

OS/390



# C/C++ Run-Time Library Reference, Volume 1



OS/390



# C/C++ Run-Time Library Reference, Volume 1

**Note**

Before using this information and the product it supports, be sure to read the general information under "Appendix B. Notices" on page 1803.

**Sixth Edition, September 1999**

This is a major revision of SC28-1663-04.

This edition applies to Version 2 Release 8 of OS/390 (5647-A01) and to all subsequent releases and modifications until otherwise indicated in new editions.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

IBM welcomes your comments. A form for readers' comments may be provided at the back of this publication, or you may address your comments to the following address:

International Business Machines Corporation  
Department 55JA, Mail Station P384  
522 South Road  
Poughkeepsie, NY 12601-5400  
United States of America

FAX (United States & Canada): 1+914+432-9405

FAX (Other Countries):

Your International Access Code +1+914+432-9405

IBMLink (United States customers only): IBMUSM10(MHVRCFS)

IBM Mail Exchange: USIB6TC9 at IBMMAIL

Internet e-mail: mhvrcfs@us.ibm.com

World Wide Web: <http://www.ibm.com/s390/os390/>

If you would like a reply, be sure to include your name, address, telephone number, or FAX number.

Make sure to include the following in your comment or note:

- Title and order number of this book
- Page number or topic related to your comment

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1996, 1999. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

# Contents

<b>Introduction</b>	xxv
IBM OS/390 C/C++ and Related Publications	xxv
Hardcopy Books	xxx
Softcopy Books	xxxi
Softcopy Examples	xxxi
OS/390 C/C++ on the World Wide Web	xxxii
C/C++ News...	xxxii
How to Read the Syntax Diagrams	xxxiii
 <b>Summary of Changes</b>	 xxxv
 <b>Volume 1 - Part 1. About IBM OS/390 C/C++</b>	 1
Changes for Version 2 Release 8	1
The C/C++ Compilers	1
The C Language	1
The C++ Language	2
Common Features of the OS/390 C and C++ Compilers	2
OS/390 C Compiler Specific Features	3
Features That Are Specific to the OS/390 C++ Compiler	4
Utilities	4
Class Libraries	5
Class Library Source	6
The Debug Tool	6
OS/390 Language Environment	6
The Program Management Binder	7
OS/390 UNIX System Services (OS/390 UNIX)	8
OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions	9
Input and Output	10
I/O Interfaces	10
File Types	11
Additional I/O Features	12
The System Programming C Facility	12
Interaction with Other IBM Products	13
Additional Features of OS/390 C/C++	14
 <b>Volume 1 - Part 2. Header Files</b>	 17
Feature Test Macros	17
aio.h	23
arpa/inet.h	23
assert.h	23
cics.h	23
collate.h	23
cpio.h	23
csp.h	24
ctest.h	24
ctype.h	24
decimal.h	24
dirent.h	24
dll.h	25
dynit.h	25

env.h	25
errno.h	25
ezbzsdc.h	28
fcntl.h	28
features.h	28
float.h	28
fmtmsg.h	29
fnmatch.h	29
__ftp.h	29
ftw.h	29
glob.h	29
grp.h	29
iconv.h	30
ims.h	30
langinfo.h	30
lc_core.h	31
leawi.h	31
libgen.h	31
limits.h	32
localdef.h	33
locale.h	33
math.h	35
memory.h	36
monetary.h	36
msgcat.h	36
mtf.h	37
ndbm.h	37
netdb.h	37
net/if.h	37
netinet/in.h	37
net/rtroute.h	37
new.h	37
nlist.h	37
nl_langinfo.h	38
nl_types.h	38
poll.h	38
pthread.h	38
pwd.h	39
re_comp.h	39
regex.h	40
regexp.h	40
rexc.h	40
search.h	40
setjmp.h	40
signal.h	41
spawn.h	42
spc.h	42
stdarg.h	42
stddef.h	42
stdio.h	43
Defined Types in stdio.h	43
Macros Defined in stdio.h	43
stdlib.h	45
string.h	46

strings.h	46
stropts.h	46
syslog.h	47
sys/file.h	47
sys/__getipc.h	47
sys/ioctl.h	47
sys/ipc.h	47
sys/mman.h	47
sys/mntent.h	47
sys/modes.h	47
sys/__messag.h	48
sys/msg.h	48
sys/ps.h	48
sys/resource.h	48
sys/sem.h	48
sys/server.h	48
sys/shm.h	48
sys/socket.h	48
sys/stat.h	48
sys/statfs.h	49
sys/statvfs.h	49
sys/time.h	49
sys/timeb.h	49
sys/times.h	49
sys/ttydev.h	49
sys/types.h	49
sys/uio.h	50
sys/un.h	50
sys/__ussos.h	50
sys/utsname.h	50
sys/wait.h	51
sys/__wlm.h	51
tar.h	51
terminat.h	51
termios.h	51
time.h	51
ucontext.h	52
uheap.h	52
unexpected.h	52
unistd.h	53
ulimit.h	53
utime.h	53
utmpx.h	54
varargs.h	54
variant.h	54
wchar.h	54
wcstr.h	55
wctype.h	56
wordexp.h	56
xti.h	56
<b>Volume 1 - Part 3. Library Functions</b>	<b>61</b>
Names	61
Standards	62

Using C Include Files from C++	64
Built-in Functions	64
IEEE Floating Point	65
External Variables	66
errno	66
daylight	67
getdate_err	67
h_errno	67
__loc1	67
loc1	67
loc2	67
locs	67
optarg	68
opterr	68
optind	68
optopt	68
signgam	68
stdin	68
stderr	68
stdout	68
t_errno	68
timezone	69
tzname	69
Functions A-O in the OS/390 C/C++ Library	70
abort() — Stop a Program	71
abs() — Calculate Integer Absolute Value	73
accept() — Accept a New Connection on a Socket	75
accept_and_recv() — Accept Connection and Receive First Message	78
access() — Determine Whether a File Can be Accessed	82
acos() — Calculate Arccosine	85
acosh() — Hyperbolic Arccosine	87
advance() — Pattern Match Given a Compiled Regular Expression	88
aio_cancel() — Cancel an Asynchronous I/O Request	90
aio_error() — Retrieve Error Status for an Asynchronous I/O Operation	92
aio_read() — Asynchronous Read from a Socket	93
aio_return() — Retrieve Status for an Asynchronous I/O Operation	96
aio_suspend() — Wait for an Asynchronous I/O Request	97
aio_write() — Asynchronous Write to a Socket	99
alarm() — Set an Alarm	102
alloca() — Allocate Storage from the Stack	105
asctime() — Convert Time to Character String	106
asin() — Calculate Arcsine	108
asinh() — Hyperbolic Arcsine	110
assert() — Verify Condition	111
atan() - atan2() — Calculate Arctangent	113
atanh() — Hyperbolic Arctangent	115
atexit() — Register Program Termination Function	116
__atoe() — ISO8859-1 to EBCDIC String Conversion	119
__atoe_l() — ISO8859-1 to EBCDIC Conversion Operation	120
atof() — Convert Character String to Double	121
atoi() — Convert Character String to Integer	122
atol() — Convert Character String to Long	123
a64l() — Convert Base-64 String Representation to Long Integer	124
basename() — Return the Last Component of a Pathname	125



bcmp()	— Compare Bytes in Memory	126
bcopy()	— Copy Bytes in Memory	127
bind()	— Bind a Name to a Socket	128
brk()	— Change Space Allocation	133
bsd_signal()	— BSD Version of signal()	135
bsearch()	— Search Arrays	137
bzero()	— Zero Out Bytes in Memory	140
calloc()	— Reserve and Initialize Storage	141
catclose()	— Close a Message Catalog Descriptor	143
catgets()	— Read a Program Message	144
catopen()	— Open a Message Catalog	146
cbrt()	— Cube Root	148
cclass()	— Return Characters in a Character Class	149
cds()	— Compare Double and Swap	151
cdump()	— Request a Main Storage Dump	152
ceil()	— Round Up to Integral Value	153
__certificate()	— Register/Deregister a Digital Certificate	154
cfgetispeed()	— Determine the Input Baud Rate	156
cfgetospeed()	— Determine the Output Baud Rate	159
cfsetispeed()	— Set the Input Baud Rate in the Termios	161
cfsetospeed()	— Set the Output Baud Rate in the Termios	163
chaudit()	— Change Audit Flags for a File by Path	165
chdir()	— Change the Working Directory	167
__check_resource_auth_np()	— Determine Access to MVS Resources	169
CheckSchEnv()	— Check WLM Scheduling Environment	172
chmod()	— Change the Mode of a File or Directory	174
chown()	— Change the Owner or Group of a File or Directory	177
chpriority()	— Change the Scheduling Priority of a Process	180
chroot()	— Change Root Directory	182
clearenv()	— Clear Environment Variables	184
clearerr()	— Reset Error and End-of-File	187
clock()	— Determine Processor Time	189
close()	— Close a File	191
closedir()	— Close a Directory	194
closelog()	— Close the Control Log	196
clrmemf()	— Clear Memory Files	197
__cnvblk()	— Convert Block	199
collequiv()	— Return a List of Equivalent Collating Elements	200
collorder()	— Return List of Collating Elements	202
collrange()	— Calculate the Range List of Collating Elements	204
colltostr()	— Return a String for a Collating Element	206
compile()	— Compile Regular Expression	208
confstr()	— Get Configurable Variables	212
connect()	— Connect a Socket	214
ConnectServer()	— Connect to WLM as a Server Manager	219
ConnectWorkMgr()	— Connect to WLM as a Work Manager	221
__console()	— Console Communication Services	223
ContinueWorkUnit()	— Continue WLM Work Unit	225
__convert_id_np()	— Convert Between DCE UUID and Userid	226
copysign()	— Copy Sign	228
cos()	— Calculate Cosine	229
cosh()	— Calculate Hyperbolic Cosine	231
creat()	— Create a New File or Rewrite an Existing One	233
CreateWorkUnit()	— Create WLM Work Unit	236

crypt()	— String Encoding Function	238
cs()	— Compare and Swap	239
csid()	— Character Set ID for Multibyte Character	240
csnap()	— Request a Condensed Dump	242
__csplist	— Retrieve CSP Parameters	243
ctdli()	— Call to DL/I	244
ctermid()	— Generate Path Name for Controlling Terminal	246
ctest()	— Start Debug Tool	248
ctime()	— Convert Time to a Character String	250
ctrace()	— Request a Traceback	252
cuserid()	— Return Character Login of the User	254
dbm_clearerr()	— Clear Database Error Indicator	255
dbm_close()	— Close a Database	256
dbm_delete()	— Delete Database Record	257
dbm_error()	— Check Database Error Indicator	258
dbm_fetch()	— Get Database Content	259
dbm_firstkey()	— Get First Key in Database	260
dbm_nextkey()	— Get Next Key in Database	262
dbm_open()	— Open a Database	264
dbm_store()	— Store Database Record	266
decabs()	— Decimal Absolute Value	268
decchk()	— Check for Valid Decimal Types	269
decfix()	— Fix Up a Nonpreferred Sign Variable	271
DeleteWorkUnit()	— Delete a WLM Work Unit	272
difftime()	— Compute Time Difference	273
dirname()	— Report the Parent Directory of a Pathname	275
DisconnectServer()	— Disconnect from WLM Server	276
div()	— Calculate Quotient and Remainder	277
dllfree()	— Free the Supplied DLL	278
dllload()	— Load the DLL and Connect it to the Application	280
dllqueryfn()	— Obtain a Pointer to a DLL Function	282
dllqueryvar()	— Obtain a Pointer to a DLL Variable	284
drand48()	— Pseudo-random Number Generator	286
dup()	— Duplicate an Open File Descriptor	288
dup2()	— Duplicate an Open File Descriptor to Another	290
dynalloc()	— Allocate a Data Set	292
dynfree()	— Deallocate a Data Set	299
dyninit()	— Initialize __dyn_t Structure	301
ecvt()	— Convert Double to String	303
encrypt()	— Encoding Function	305
endgrent()	— Group Database Entry Functions	307
endhostent()	— Work with a Host Entry	308
endnetent()	— Close Network Information Data Sets	309
endprotoent()	— Work with a Protocol Entry	310
endpwent()	— User Database Functions	311
endservent()	— Close Network Services Information Data Sets	312
endutxent()	— Close the utmpx Database	313
erand48()	— Pseudo-random Number Generator	314
erf()	— erf() - erfc()	316
__err2ad()	— Return Address of Reason Code of Last Failure	318
__errno2()	— Return Reason Code Information	319
__etoa()	— EBCDIC to ISO8859-1 String Conversion	320
__etoa_l()	— EBCDIC to ISO8859-1 Conversion Operation	321
exec Functions		322

exit() — End Program	330
_exit() — End a Process and Bypass the Cleanup	332
exp() — Calculate Exponential Function	334
expm1() — Exponential Minus One	335
extlink_np() — Create an External Symbolic Link	336
fabs() — Calculate Floating-Point Absolute Value	338
fattach() — Attach a STREAMS-based File Descriptor to a File in the File System Name Space	339
fchmod() — Change Audit Flags for a File by Descriptor	341
fchdir() — Change Working Directory	343
fchmod() — Change the Mode of a File or Directory by Descriptor	344
fchown() — Change the Owner or Group by File Descriptor	346
fclose() — Close File	348
fcntl() — Control Open File Descriptors	350
fcvt() — Convert Double to String	358
fdelrec() — Delete a VSAM Record	359
fdetach() — Detach a Name from a STREAMS-based File Descriptor	361
fdopen() — Associate a Stream with an Open File Descriptor	363
feof() — Test End-of-File Indicator	365
ferror() — Test for Read/Write Errors	367
fetch() — Get a Load Module	369
fetchep() — Share Writable Static	382
fflush() — Write Buffer to File	385
ffs() — Find First Set Bit in an Integer	387
fgetc() — Read a Character	388
fgetpos() — Get File Position	390
fgets() — Read a String from a Stream	392
fgetwc() — Get Next Wide Character	394
fgetws() — Get a Wide-Character String	396
fileno() — Get the File Descriptor from an Open Stream	399
finite() — Determine the Infinity Classification of a Floating-Point Number	401
fldata() — Retrieve File Information	402
flocate() — Locate a VSAM Record	405
floor() — Round Down to Integral Value	408
fmod() — Calculate Floating-Point Remainder	410
fmtmsg() — Display a Message in the Specified Format	412
fnmatch() — Match Filename or Pathname	415
fopen() — Open a File	417
fork() — Create a New Process	422
fortrc() — Return FORTRAN Return Code	426
fpathconf() — Determine Configurable Path Name Variables	427
fp_clr_flag() — Reset Floating-Point Exception Status Flag	430
fp_raise_xcp() — Raise a Floating-Point Exception	431
fp_read_flag() — Return the Current Floating-Point Exception Status	433
fp_read_rnd() — Determine Rounding Mode	435
fprintf() - printf() - sprintf() — Format and Write Data	436
fp_swap_rnd() — Swap Rounding Mode	446
fputc() — Write a Character	448
fputs() — Write a String	450
fputwc() — Output a Wide-Character	452
fputws() — Output a Wide-Character String	454
fread() — Read Items	456
free() — Free a Block of Storage	458

freopen() — Redirect an Open File	460
frexp() — Extract Mantissa and Exponent of the Floating-Point Value	462
fscanf() — scanf() — sscanf() — Read and Format Data	464
fseek() — Change File Position	474
fsetpos() — Set File Position	477
fstat() — Get Status Information about a File	479
fstatvfs() — Get File System Information	481
fsync() — Write Changes to Direct-Access Storage	483
ftell() — Get Current File Position	485
ftime() — Get the Date and Time	487
ftok() — Generate an Interprocess Communication (IPC) key	488
ftruncate() — Truncate a File	489
ftw() — Traverse a File Tree	491
fupdate() — Update a VSAM Record	493
fwrite() — Write Items	495
gamma() — Calculate Gamma Function	497
gcvrt() — Convert Double to String	498
getc() - getchar() — Read a Character	499
getclientid() — Get the Identifier for the Calling Application	501
__getclientid() — Get the PID Identifier for the Calling Application	503
getcontext() — Get User Context	505
getcwd() — Get Path Name of the Working Directory	508
getdate() — Convert User Format Date and Time	510
getdtablesize() — Get the File Descriptor Table Size	513
getegid() — Get the Effective Group ID	514
getenv() — Get Value of Environment Variables	515
__getenv() — Get an Environment Variable	517
geteuid() — Get the Effective User ID	519
getgid() — Get the Real Group ID	521
getgrent() — Get Group Database Entry	522
getgrgid() — Access the Group Database by ID	523
getgrnam() — Access the Group Database by Name	525
getgroups() — Get a List of Supplementary Group IDs	527
getgroupsbyname() — Get Supplementary Group IDs by User Name	529
gethostbyaddr() — Get a Host Entry by Address	531
gethostbyname() — Get a Host Entry by Name	534
gethostent() — Get the Next Host Entry	537
gethostid() — Get the Unique Identifier of the Current Host	539
gethostname() — Get the Name of the Host Processor	540
getibmopt() — Get IBM TCP/IP Image	542
getibmsockopt() — Get the Options Associated with a Bulk Mode Socket	543
__getipc() — Query Interprocess Communications	546
getitimer() — Get Value of an Interval Timer	548
getlogin() — Get the User Login Name	550
__getlogin1() — Get the User Login Name	552
getmccoll() — Get Next Collating Element from String	554
getmsg() - getpmsg() — Receive Next Message from a STREAMS File	555
getnetbyaddr() — Get a Network Entry by Address	558
getnetbyname() — Get a Network Entry by Name	560
getnetent() — Get the Next Network Entry	562
getopt() — Command Option Parsing	564
getpagesize() — Get the Current Page Size	566
getpass() — Read a String of Characters Without Echo	567
getpeername() — Get the Name of the Peer Connected to a Socket	568

getpgid()	— Get Process Group ID	570
getpgrp()	— Get the Process Group ID	571
getpid()	— Get the Process ID	573
getpmsg()	— Receive Next Message from a STREAMS File	575
getppid()	— Get the Parent Process ID	576
getpriority()	— Get Process Scheduling Priority	578
getprotobyname()	— Get a Protocol Entry by Name	580
getprotobynumber()	— Get a Protocol Entry by Number	582
getprotoent()	— Get the Next Protocol Entry	584
getpwent()	— Get User Database Entry	586
getpwnam()	— Access the User Database by User Name	587
getpwuid()	— Access the User Database by User ID	589
getrlimit()	— Control Maximum Resource Consumption	591
getrusage()	— Get Information About Resource Utilization	594
gets()	— Read a String	595
getservbyname()	— Get a Server Entry by Name	597
getservbyport()	— Get a Service Entry by Port	599
getservent()	— Get the Next Service Entry	601
getsid()	— Get Process Group ID of Session Leader	603
getsockname()	— Get the Name of a Socket	604
getsockopt()	— Get the Options Associated with a Socket	606
getstabsize()	— Get the Socket Table Size	611
getsubopt()	— Parse Suboption Arguments	612
getsyntax()	— Return LC_SYNTAX Characters	614
gettimeofday()	— Get Date and Time	616
getuid()	— Get the Real User ID	618
getutxent()	— Read Next Entry in utmpx Database	620
getutxid()	— Search by ID utmpx Database	622
getutxline()	— Search by Line utmpx Database	624
getw()	— Get a Machine Word from a Stream	626
getwc()	— Get a Wide Character	627
getwchar()	— Get a Wide Character	629
getwd()	— Get the Current Working Directory	631
getwmccoll()	— Get Next Collating Element from Wide String	632
givesocket()	— Make the Specified Socket Available	633
glob()	— Generate Pathnames Matching a Pattern	636
globfree()	— Free Storage Allocated by glob()	639
gmtime()	— Convert Time to Broken-Down UTC Time	640
grantpt()	— Grant Access to the Slave Pseudoterminal Device	642
hcreate()	— Create Hash Search Tables	643
hdestroy()	— Destroy Hash Search Tables	644
__heaprpt()	— Obtain Dynamic Heap Storage Report	645
hsearch()	— Search Hash Tables	647
htonl()	— Translate Address Host to Network Long	648
htons()	— Translate an Unsigned Short Integer into Network Byte Order	649
hypot()	— Calculate Hypotenuse	650
ibmsflush()	— Flush the Application-side Datagram Queue	652
iconv()	— Code Conversion	654
iconv_close()	— Deallocate Code Conversion Descriptor	657
iconv_open()	— Allocate Code Conversion Descriptor	658
ilogb()	— Integer Unbiased Exponent	659
index()	— Search for Character	660
inet_addr()	— Translate an Internet Address into Network Byte Order	661
inet_lnaof()	— Translate a Local Network Address into Host Byte Order	663

inet_makeaddr()	— Create an Internet Host Address	664		
inet_netof()	— Get the Network Number from the Internet Host Address	665		
inet_network()	— Get the Network Number from the Decimal Host Address	666		
inet_ntoa()	— Get the Decimal Internet Host Address	667		
initgroups()	— Initialize the Supplementary Group ID List for the Process	669		
initstate()	— Initialize Generator for Random()	670		
insque()	— Insert an Element into a Doubly-linked List	671		
ioctl()	— Control Device	672		
__ipdbcs()	— Retrieve the List of Requested DBCS Tables to Load	688		
__ipdspx()	— Retrieve the Data Set Prefix Specified	689		
__iphost()	— Retrieve the Resolver Supplied Hostname	690		
__ipmsgc()	— Determine the Case to Use for FTP Messages	691		
__ipnode()	— Retrieve the Resolver Supplied Node Name	692		
__iptcpn()	— Retrieve the Resolver Supplied Jobname or Userid	693		
isalnum()	to isxdigit()	— Test Integer Value	694	
isascii()	— Test for 7-bit US-ASCII Character	697		
isastream()	— Test a File Descriptor	702		
isatty()	— Test if Descriptor Represents a Terminal	703		
__isBFP()	— Determine Application Floating-Point Format	705		
isblank()	— Test for Blank Character Classification	706		
iscics()	— Verify Whether CICS is Running	708		
iscntrl()	— Test for Control Classification	709		
isdigit()	— Test for Hexadecimal-Digit Classification	709		
isgraph()	— Test for Graphic Classification	709		
islower()	— Test for Lowercase	709		
ismccollet()	— Identify a Multi-Character Collating Element	710		
isnan()	— Test for NaN	712		
__isPosixOn()	— Test for Posix Runtime Option	713		
isprint()	— Test for Printable Character Classification	714		
ispunct()	— Test for Punctuation Classification	714		
isspace()	— Test for Space Character Classification	714		
isupper()	— Test for Uppercase Letter Classification	714		
iswalnum()	to iswxdigit()	— Test Wide Integer Value	715	
iswblank()	— Test for Blank Character Classification	718		
iswcntrl()	— Test for Control Classification	719		
iswctype()	— Test for Character Property	720		
iswdigit()	— Test for Hexadecimal-Digit Classification	722		
iswgraph()	— Test for Graphic Classification	722		
iswlower()	— Test for Lowercase	722		
iswprint()	— Test for Printable Character Classification	722		
iswpunct()	— Test for Punctuation Classification	722		
iswspace()	— Test for Space Character Classification	722		
iswupper()	— Test for Uppercase Letter Classification	722		
iswxdigit()	— Test for Hexadecimal-Digit Classification	722		
isxdigit()	— Test for Hexadecimal-Digit Classification	722		
JoinWorkUnit()	— Join a WLM Work Unit	723		
jrand48()	— Pseudo-random Number Generator	724		
j0()	- j1()	- jn()	— Bessel Functions of the First Kind	726
kill()	— Send a Signal to a Process	728		
killpg()	— Send a Signal to a Process Group	731		
labs()	— Calculate Long Absolute Value	733		
lchown()	— Change Owner and Group of a File	734		
lcong48()	— Pseudo-random Number Initializer	736		
ldexp()	— Multiply by a Power of Two	738		

ldiv() — Compute Quotient and Remainder of Integral Division	740
LeaveWorkUnit() — Leave a WLM Work Unit	742
lfind() — Linear Search Routine	743
lgamma() — Log Gamma Function	744
__librel() — Query Release Level	746
link() — Create a Link to a File	749
listen() — Prepare the Server for Incoming Client Requests	752
localdtconv() — Date/Time Formatting Convention Inquiry	754
localeconv() — Query Numeric Conventions	756
localtime() — Convert Time and Correct for Local Time	758
lockf() — Record Locking on Files	760
log() — Calculate Natural Logarithm	762
logb() — Unbiased Exponent	763
__login() — Create a New Security Environment for Process	764
log1p() — Natural Log of $x + 1$	766
log10() — Calculate Base 10 Logarithm	767
longjmp() — Restore Stack Environment	768
_longjmp() — Non-Local Goto	771
lrand48() — Pseudo-random Number Generator	773
lsearch() — Linear Search and Update	775
lseek() — Change the Offset of a File	776
lstat() — Get Status of File or Symbolic Link	778
l64a() — Convert Long to Base-64 String Representation	782
makecontext() — Modify User Context	783
malloc() — Reserve Storage Block	786
maxcoll() — Return Maximum Collating Element	788
maxdesc() — Get Socket Numbers to Extend Beyond the Default Range	789
mblen() — Calculate Length of Multibyte Character	791
mbrlen() — Calculate Length of Multibyte Character	794
mbrtowc() — Convert a Multibyte Character to a Wide Character	797
mbsinit() — Test State Object for Initial State	800
mbsrtowcs() — Convert a Multibyte String to a Wide-Character String	802
mbstowcs() — Convert Multibyte Characters to Wide Characters	804
mbtowc() — Convert Multibyte Character to Wide Character	806
memccpy() — Copy Bytes in Memory	808
memchr() — Search Buffer	809
memcmp() — Compare Bytes	811
memcpy() — Copy Buffer	813
memmove() — Move Buffer	814
memset() — Set Buffer to Value	816
mkdir() — Make a Directory	817
mkfifo() — Make a FIFO Special File	820
mknod() — Make a Directory or File	823
mkstemp() — Make a Unique Filename	826
mktemp() — Make a Unique Filename	827
mktime() — Convert Local Time	828
__mlockall() — Lock the Address Space of a Process	831
mmap() — Map Pages of Memory	832
modf() — Extract Fractional and Integral Parts of Floating-Point Value	837
mount() — Make a File System Available	838
mprotect() — Set Protection of Memory Mapping	841
rand48() — Pseudo-random Number Generator	843
msgctl() — Message Control Operations	845
msgget() — Get Message Queue	847

msgrcv() — Message Receive Operation	850
msgsnd() — Message Send Operations	852
msgxrcv() — Extended Message Receive Operation	854
msync() — Synchronize Memory with Physical Storage	856
munmap() — Unmap Pages of Memory	858
nextafter() — Next Representable Double Float	860
nftw() — Traverse a File Tree	861
nice() — Change Priority of a Process	864
nlist() — Get Entries from a Name List	865
nl_langinfo() — Retrieve Locale Information	866
rand48() — Pseudo-random Number Generator	868
ntohl() — Translate a Long Integer into Host Byte Order	870
ntohs() — Translate an Unsigned Short Integer into Host Byte Order	871
open() — Open a File	872
opendir() — Open a Directory	877
__opendir2() — Open a Directory	880
openlog() — Open the System Control Log	882
__openMvsRel() — Get the OS/390 UNIX Release Number	884
__open_stat() — Open a File and Get File Status Information	885
__osenv() — Capture the WLM and Pthread Security Attributes	888

<b>Volume 2 - Part 3. Library Functions (continued)</b>	893
Functions P-Z in the OS/390 C/C++ Library	893
__passwd() — Verify/Change User Password	894
pathconf() — Determine Configurable Path Name Variables	896
pause() — Suspend a Process Pending a Signal	899
pclose() — Close a Pipe Stream to or from a Process	901
perror() — Print Error Message	903
__pid_affinity() — Add or Delete Process Affinity	905
pipe() — Create an Unnamed Pipe	907
poll() — Monitor Activity on File Descriptors and Message Queues	910
popen() — Initiate a Pipe Stream to or from a Process	914
pow() — Raise to Power	916
printf() — Format and Write Data	917
pthread_attr_destroy() — Destroy the Thread Attributes Object	918
pthread_attr_getdetachstate() — Get the Detach State Attribute	920
pthread_attr_getstacksize() — Get the Thread Attribute Stacksize Object	922
pthread_attr_getsynctype_np() — Get Thread Sync Type	924
pthread_attr_getweight_np() — Get Weight of Thread Attribute Object	925
pthread_attr_init() — Initialize a Thread Attribute Object	927
pthread_attr_setdetachstate() — Set the Detach State Attribute Object	929
pthread_attr_setstacksize() — Set the Stacksize Attribute Object	931
pthread_attr_setsynctype_np() — Set Thread Sync Type	933
pthread_attr_setweight_np() — Set Weight of Thread Attribute Object	934
pthread_cancel() — Cancel a Thread	936
pthread_cleanup_pop() — Remove a Cleanup Handler	939
pthread_cleanup_push() — Establish a Cleanup Handler	941
pthread_cond_broadcast() — Broadcast a Condition	943
pthread_cond_destroy() — Destroy the Condition Variable Object	945
pthread_cond_init() — Initialize a Condition Variable	947
pthread_cond_signal() — Signal a Condition	948
pthread_cond_timedwait() — Wait on a Condition Variable	950
pthread_cond_wait() — Wait on a Condition Variable	952
pthread_condattr_destroy() — Destroy Condition Variable Attribute Object	955



pthread_condattr_getkind_np() — Get Kind Attribute from a Condition Variable Attribute Object	957
pthread_condattr_init() — Initialize a Condition Attribute Object	959
pthread_condattr_setkind_np() — Set Kind Attribute from a Condition Variable Attribute Object	961
pthread_create() — Create a Thread	963
pthread_detach() — Detach a Thread	965
pthread_equal() — Compare Thread IDs	967
pthread_exit() — Exit a Thread	969
pthread_getspecific() — Get the Thread-Specific Value for a Key	971
pthread_getspecific_d8_np() — Get the Thread-Specific Value for a Key	974
pthread_join() — Wait for a Thread to End	977
pthread_join_d4_np() — Wait for a Thread to End	979
pthread_key_create() — Create Thread-Specific Data Key	981
pthread_kill() — Send a Signal to a Thread	984
pthread_mutex_destroy() — Delete a Mutex Object	986
pthread_mutex_init() — Initialize a Mutex Object	988
pthread_mutex_lock() — Wait for a Lock on a Mutex Object	990
pthread_mutex_trylock() — Attempt to Lock a Mutex Object	992
pthread_mutex_unlock() — Unlock a Mutex Object	994
pthread_mutexattr_destroy() — Destroy a Mutex Attribute Object	996
pthread_mutexattr_getkind_np() — Get Kind from a Mutex Attribute Object	998
pthread_mutexattr_getpshared() — Get the Process-Shared Mutex Attribute	1001
pthread_mutexattr_gettype() — Get Type of Mutex Attribute Object	1002
pthread_mutexattr_init() — Initialize a Mutex Attribute Object	1004
pthread_mutexattr_setkind_np() — Set Kind for a Mutex Attribute Object	1006
pthread_mutexattr_setpshared() — Set the Process-Shared Mutex Attribute	1009
pthread_mutexattr_settype() — Set Type of Mutex Attribute Object	1011
pthread_once() — Invoke a Function Once	1013
pthread_rwlock_destroy() — Destroy a Read-Write Lock Object	1016
pthread_rwlock_init() — Initialize a Read-Write Lock Object	1017
pthread_rwlock_rdlock() — Wait for a Lock on a Read-Write Lock Object	1018
pthread_rwlock_tryrdlock() — Attempt to Lock a Read-Write Lock Object for Reading	1020
pthread_rwlock_trywrlock() — Attempt to Lock a Read-Write Lock Object for Writing	1021
pthread_rwlock_unlock() — Unlock a Read-Write Lock Object	1022
pthread_rwlock_wrlock() — Wait for a Lock on a Read-Write Lock Object for Writing	1023
pthread_rwlockattr_destroy() — Destroy a Read-Write Lock Attribute Object	1024
pthread_rwlockattr_getpshared() — Get the Process-Shared Read-Write Lock Attribute	1025
pthread_rwlockattr_init() — Initialize a Read-Write Lock Attribute Object	1026
pthread_rwlockattr_setpshared() — Set the Process-Shared Read-Write Lock Attribute	1027
pthread_security_np() — Create or Delete Thread Level Security	1029
pthread_self() — Get the Caller	1033
pthread_setinr() — Set Thread's Cancelability State	1035
pthread_setinrtype() — Set Thread's Cancelability Type	1038
pthread_set_limit_np() — Set Task and Thread Limits	1041
pthread_setspecific() — Set the Thread-Specific Value for a Key	1043

pthread_tag_np() — Set and Query Thread Tag Data	1046
pthread_testintr() — Establish a Cancelability Point	1047
pthread_yield() — Release the Processor to Other Threads	1049
ptsname() — Get Name of the Slave Pseudoterminal Device	1051
putc() - putchar() — Write a Character	1052
putenv() — Change or Add an Environment Variable	1054
putmsg() - putpmsg() — Send a Message on a STREAM	1056
puts() — Write a String	1059
pututxline() — Write Entry to utmpx Database	1061
putw() — Put a Machine Word on a Stream	1063
putwc() — Output a Wide Character	1064
putwchar() — Output a Wide Character to Standard Output	1066
qsort() — Sort Array	1068
QueryMetrics() — Query WLM System Information	1070
QuerySchEnv() — Query WLM Scheduling Environment	1072
raise() — Raise Signal	1074
rand() — Generate Random Number	1077
random() — A Better Random-Number Generator	1079
read() — Read From a File or Socket	1080
readdir() — Read an Entry from a Directory	1086
__readdir2() — Read Directory Entry and Get File Information	1089
readlink() — Read the Value of a Symbolic Link	1091
readv() — Read Data on a File or Socket and Store in a Set of Buffers	1093
realloc() — Change Reserved Storage Block Size	1096
realpath() — Resolve Path Name	1099
re_comp() — Compile Regular Expression	1100
recv() — Receive Data on a Socket	1103
recvfrom() — Receive Messages on a Socket	1106
recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers	1110
re_exec() — Match Regular Expression	1115
regcmp() — Compile Regular Expression	1116
regcomp() — Compile Regular Expression	1120
regerror() — Return Error Message	1123
regex() — Execute Compiled Regular Expression	1125
regexec() — Execute Compiled Regular Expression	1126
regfree() — Free Memory for Regular Expression	1129
release() — Delete a Load Module	1130
remainder — Remainder Function	1132
remove() — Delete File	1133
remque() — Remove an Element from a Doubly-linked List	1135
rename() — Rename File	1136
rewind() — Set File Position to Beginning of File	1139
rewinddir() — Reposition a Directory Stream to the Beginning	1141
rexec() — Execute Commands One at a Time on a Remote Host	1143
rindex() — Search for Character	1145
rint() — Round to Nearest Integral Value	1146
rmdir() — Remove a Directory	1147
rpmatch() — Test for a Yes/No Response Match	1150
sbrk() — Change Space Allocation	1152
scalb() — Load Exponent	1154
scanf() — Read and Format Data	1155
seed48() — Pseudo-random Number Initializer	1156
seekdir() — Set Position of Directory Stream	1158

<code>select()</code> — Monitor Activity on Files/Sockets and Message Queues . . . . .	1159
<code>selectex()</code> — Monitor Activity on Files/Sockets and Message Queues . . .	1166
<code>semctl()</code> — Semaphore Control Operations . . . . .	1169
<code>semget()</code> — Get a Set of Semaphores . . . . .	1172
<code>semop()</code> — Semaphore Operations . . . . .	1175
<code>send()</code> — Send Data on a Socket . . . . .	1178
<code>send_file()</code> — Send File Data Over a Socket . . . . .	1181
<code>sendmsg()</code> — Send Messages on a Socket . . . . .	1185
<code>sendto()</code> — Send Data on a Socket . . . . .	1190
<code>__server_classify()</code> — Set Classify Area Field . . . . .	1194
<code>__server_classify_create()</code> — Create a Classify Area . . . . .	1197
<code>__server_classify_destroy()</code> — Delete a Classify Area . . . . .	1198
<code>__server_classify_reset()</code> — Reset a Classify Area to an Initial State . .	1199
<code>__server_init()</code> — Initialize Server . . . . .	1200
<code>__server_pwu()</code> — Process Server Work Unit . . . . .	1203
<code>setbuf()</code> — Control Buffering . . . . .	1208
<code>setcontext()</code> — Restore User Context . . . . .	1210
<code>setegid()</code> — Set the Effective Group ID . . . . .	1213
<code>setenv()</code> — Add, Delete, and Change Environment Variables . . . . .	1215
<code>seteuid()</code> — Set the Effective User ID . . . . .	1218
<code>setgid()</code> — Set the Group ID . . . . .	1220
<code>setgrent()</code> — Reset Group Database to First Entry . . . . .	1222
<code>setgroups()</code> — Set the Supplementary Group ID List for the Process . . .	1223
<code>sethostent()</code> — Open the Host Information Data Set . . . . .	1224
<code>setibmopt()</code> — Set IBM TCP/IP Image . . . . .	1225
<code>setibmsockopt()</code> — Set Options Associated with a Bulk Mode Socket . . .	1227
<code>setitimer()</code> — Set Value of an Interval Timer . . . . .	1232
<code>setjmp()</code> — Preserve Stack Environment . . . . .	1234
<code>_setjmp()</code> — Set Jump Point for a Non-Local Goto . . . . .	1237
<code>setkey()</code> — Set Encoding Key . . . . .	1239
<code>setlocale()</code> — Set Locale . . . . .	1241
<code>setlogmask()</code> — Set the Mask for the Control Log . . . . .	1250
<code>setnetent()</code> — Open the Network Information Data Set . . . . .	1251
<code>set_new_handler()</code> — Register a Function for <code>set_new_handler()</code> . . . . .	1252
<code>setpeer()</code> — Preset the Socket Peer Address . . . . .	1253
<code>setpgid()</code> — Set Process Group ID for Job Control . . . . .	1254
<code>setpgrp()</code> — Set Process Group ID . . . . .	1256
<code>setpriority()</code> — Set Process Scheduling Priority . . . . .	1257
<code>setprotoent()</code> — Open the Protocol Information Data Set . . . . .	1259
<code>setpwent()</code> — Reset User Database Search . . . . .	1260
<code>setregid()</code> — Set Real and Effective Group IDs . . . . .	1261
<code>setreuid()</code> — Set Real and Effective User IDs . . . . .	1262
<code>setrlimit()</code> — Control Maximum Resource Consumption . . . . .	1264
<code>setservent()</code> — Open the Network Services Information Data Set . . . . .	1267
<code>setsid()</code> — Create Session, Set Process Group ID . . . . .	1268
<code>setsockopt()</code> — Set Options Associated with a Socket . . . . .	1270
<code>setstate()</code> — Change Generator for <code>random()</code> . . . . .	1275
<code>set_terminate()</code> — Register a Function for <code>terminate()</code> . . . . .	1276
<code>_SET_THLIIPADDR()</code> — Set the Client's IP Address . . . . .	1277
<code>setuid()</code> — Set the Effective User ID . . . . .	1278
<code>set_unexpected()</code> — Register a Function for <code>unexpected()</code> . . . . .	1281
<code>setutxent()</code> — Reset to Start of <code>utmpx</code> Database . . . . .	1282
<code>setvbuf()</code> — Control Buffering . . . . .	1283
<code>shmat()</code> — Shared Memory Attach Operation . . . . .	1285

|

shmctl() — Shared Memory Control Operations	1287
shmdt() — Shared Memory Detach Operation	1289
shmget() — Get a Shared Memory Segment	1290
shutdown() — Shut Down All or Part of a Duplex Connection	1293
sigaction() — Examine or Change a Signal Action	1295
__sigactionset() — Examine and/or Change Signal Actions	1305
sigaddset() — Add a Signal to the Signal Mask	1312
sigaltstack() — Set and/or Get Signal Alternate Stack Context	1314
sigdelset() — Delete a Signal from the Signal Mask	1316
sigemptyset() — Initialize a Signal Mask to Exclude All Signals	1318
sigfillset() — Initialize a Signal Mask to Include All Signals	1320
sighold() — Add a Signal to a Thread	1322
sigignore() — Set Disposition to Ignore a Signal	1323
siginterrupt() — Allow Signals to Interrupt Functions	1324
sigismember() — Test If a Signal Is in a Signal Mask	1325
siglongjmp() — Restore the Stack Environment and Signal Mask	1327
signal() — Handle Interrupts	1330
__signgam() — Return signgam Reference	1335
sigpause() — Unblock a Signal and Wait for a Signal	1336
sigpending() — Examine Pending Signals	1337
sigprocmask() — Examine or Change a Thread	1339
sigrelse() — Remove a Signal from a Thread	1342
sigset() — Change a Signal Action and/or a Thread	1343
sigsetjmp() — Save Stack Environment and Signal Mask	1346
sigstack() — Set and/or Get Signal Stack Context	1349
sigsuspend() — Change Mask and Suspend the Thread	1351
sigtimedwait() — Wait for Queued Signals	1354
sigwait() — Wait for an Asynchronous Signal	1356
sigwaitinfo() — Wait for Queued Signals	1358
sin() — Calculate Sine	1360
sinh() — Calculate Hyperbolic Sine	1362
sleep() — Suspend Execution of a Thread	1364
__smf_record() — Record an SMF Record	1366
sock_debug() — Provide OE Syscall Tracing Facility	1367
sock_debug_bulk_perf0() — Produce a Report When a Socket Configured	1368
sock_do_bulkmode() — Use Bulk Mode for Messages Read by the Socket	1369
sock_do_teststor — Check for Attempt to Access Storage Outside	1370
socket() — Create a Socket	1371
socketpair() — Create a Pair of Sockets	1375
spawn() - spawnp() — Spawn a New Process	1377
__spawn2() - spawnp2() — Spawn a New Process Using Enhanced Inheritance Structure	1388
sprintf() — Format and Write Data to Buffer	1395
sqrt() — Calculate Square Root	1396
srand() — Set Seed for rand() Function	1398
srandom() — Use Seed to Initialize Generator for random()	1400
srand48() — Pseudo-random Number Initializer	1401
sscanf() — Read and Format Data from Buffer	1403
stat() — Get File Information	1404
statvfs() — Get File System Information	1408
step() — Pattern Match with Regular Expression	1411
strcasecmp() — Case-insensitive String Comparison	1413

strcat()	— Concatenate Strings	1414
strchr()	— Search for Character	1416
strcmp()	— Compare Strings	1418
strcoll()	— Compare Strings	1420
strcpy()	— Copy String	1422
strcspn()	— Compare Strings	1424
strdup()	— Duplicate a String	1426
strerror()	— Get Pointer to Runtime Error Message	1427
strfmon()	— Convert Monetary Value to String	1428
strftime()	— Convert to Formatted Time	1432
strlen()	— Determine String Length	1436
strncasecmp()	— Case-insensitive String Comparison	1437
strncat()	— Concatenate Strings	1438
strncmp()	— Compare Strings	1440
strncpy()	— Copy String	1442
strpbrk()	— Find Characters in String	1444
strptime()	— Date and Time Conversion	1445
strrchr()	— Find Last Occurrence of Character in String	1449
strspn()	— Search String	1451
strstr()	— Locate Substring	1453
strtcoll()	— Return Collating Element for String	1454
strtod()	— Convert Character String to Double	1456
strtok()	— Tokenize String	1458
strtol()	— Convert Character String to Long	1460
strtoul()	— Convert String to Unsigned Integer	1462
strxfrm()	— Transform String	1465
svc99()	— Access Supervisor Call	1467
swab()	— Copy and Swap Bytes	1471
swapcontext()	— Save and Restore User Context	1472
swprintf()	— Write Wide Characters to a Wide-Character Array	1475
swscanf()	— Read a Wide-Character String	1477
symlink()	— Create a Symbolic Link to a Path Name	1479
sync()	— Schedule File System Updates	1482
sysconf()	— Determine System Configuration Options	1483
syslog()	— Send a Message to the Control Log	1486
system()	— Execute a Command	1488
t_accept()	— Accept a Connect Request	1493
takesocket()	— Acquire a Socket from Another Program	1496
t_alloc()	— Allocate a Library Structure	1498
tan()	— Calculate Tangent	1500
tanh()	— Calculate Hyperbolic Tangent	1502
t_bind()	— Bind an Address to a Transport Endpoint	1504
tcdrain()	— Wait Until Output Has Been Transmitted	1507
tcflow()	— Suspend or Resume Data Flow on a Terminal	1509
tcflush()	— Flush Input or Output on a Terminal	1512
tcgetattr()	— Get the Attributes for a Terminal	1515
__tcgetcp()	— Get Terminal Code Page Names	1517
tcgetpgrp()	— Get the Foreground Process Group ID	1520
tcgetsid()	— Get Process Group ID for Session Leader for Controlling Terminal	1522
t_close()	— Close a Transport Endpoint	1523
t_connect()	— Establish a Connection with Another Transport User	1524
tcperror()	— Print the Error Messages of a Socket Function	1527
tcsendbreak()	— Send a Break Condition to a Terminal	1529

tcsetattr() — Set the Attributes for a Terminal	1531
__tcsetcp() — Set Terminal Code Page Names	1542
tcsetpgrp() — Set the Foreground Process Group ID	1546
__tcsettables() — Set Terminal Code Page Names and Conversion Tables	1549
tdelete() — Binary Tree Delete	1555
telldir() — Current Location of Directory Stream	1557
tempnam() — Generate a Temporary File Name	1558
terminate() — Terminate After Failures in C++ Error Handling	1560
t_error() — Produce Error Message	1561
tfind() — Binary Tree Find Node	1563
t_free() — Free a Library Structure	1564
t_getinfo() — Get Protocol-specific Service Information	1566
t_getprotaddr — Get the Protocol Addresses	1568
t_getstate() — Get the Current State	1569
time() — Determine Current Time	1570
times() — Get Process and Child Process Times	1572
tinit() — Attach and Initialize Subtasks	1575
t_listen() — Listen for a Connect Indication	1577
t_look() — Look at the Current Event on a Transport Endpoint	1579
tmpfile() — Create Temporary File	1582
tmpnam() — Produce Temporary File Name	1584
toascii() — Translate Integer to a 7-bit ASCII Character	1586
tolower() - toupper() — Convert Character Case	1592
_tolower() — Translate Uppercase Characters to Lower Case	1593
t_open() — Establish a Transport Endpoint	1594
t_optmgmt() — Manage Options for a Transport Endpoint	1596
_toupper() — Translate Lowercase Characters to Upper-case	1603
towlower() - towupper() — Convert Wide Character Case	1604
t_rcv() — Receive Data or Expedited Data Sent Over a Connection	1605
t_rcvconnect() — Receive the Confirmation from a Connect Request	1607
t_rcvdis() — Retrieve Information from Disconnect	1609
t_rcvrel() — Acknowledge Receipt of an Orderly Release Indication	1611
t_rcvudata — Receive a Data Unit	1612
t_rcvuderr — Receive a Unit Data Error Indication	1613
truncate() — Truncate a File to a Specified Length	1614
tsched() — Schedule MTF Subtask	1615
tsearch() — Binary Tree Search	1617
t_snd() — Send Data or Expedited Data Over a Connection	1619
t_snddis() — Send User-initiated Disconnect Request	1621
t_sndrel() — Initiate an Orderly Release	1623
t_sndudata — Send a Data Unit	1624
t_strerror() — Produce an Error Message String	1625
t_sync() — Synchronize Transport Library	1626
tsyncro() — Wait for MTF Subtask Termination	1628
tterm() — Terminate Subtasks	1630
ttynam() — Get the Name of a Terminal	1632
ttyslot() — Find the Slot in the utmpx File of the Current User	1634
t_unbind() — Disable a Transport Endpoint	1635
twalk() — Binary Tree Walk	1636
tzset() — Set the Time Zone	1637
ualarm() — Set the Interval Timer	1640
__ucreate() — Create a Heap Using User-Provided Storage	1641
__ufree() — Return Storage to a User-Created Heap	1643

__uheapreport() — Produce a Storage Report for a User-Created Heap	1644
ulimit() — Get/Set Process File Size Limits	1645
__umalloc() — Allocate Storage from a User-Created Heap	1646
umask() — Set and Retrieve File Creation Mask	1647
umount() — Remove a Virtual File System	1649
uname() — Display Current Operating System Name	1652
unexpected() — Handle Exception Not Listed in Exception Specification	1654
ungetc() — Push Character onto Input Stream	1655
ungetwc() — Push a Wide Character onto a Stream	1658
unlink() — Remove a Directory Entry	1660
unlockpt() — Unlock a Pseudoterminal Master/Slave Pair	1662
usleep() — Suspend Execution for an Interval	1663
utime() — Set File Access and Modification Times	1664
utimes() — Set File Access and Modification Times	1667
__utmpxname() — Change the utmpx Database Name	1669
va_arg() - va_end() - va_start() — Access Function Arguments	1670
valloc() — Page-Aligned Memory Allocator	1675
vfork() — Create a New Process	1676
vfprintf() — Format and Print Data to Stream	1679
vprintf() — Format and Print Data to stdout	1681
vsprintf() — Format and Print Data to Buffer	1683
vswprintf() — Write Wide Characters	1685
wait() — Wait for a Child Process to End	1687
waitid() — Wait for Child Process to Change State	1690
waitpid() — Wait for a Specific Child Process to End	1692
wait3() — Wait for Child Process to Change State	1696
wcrtomb() — Convert a Wide Character to a Multibyte Character	1697
wcscat() — Append to Wide-Character String	1699
wcschr() — Search for Wide-Character Substring	1701
wscmp() — Compare Wide-Character Strings	1703
wscoll() — Language Collation String Comparison	1705
wscpy() — Copy Wide-Character String	1707
wscspn() — Find Offset of First Wide-Character Match	1709
wcsftime() — Format Date and Time	1711
wcsid() — Character Set ID for Wide Character	1713
wcslen() — Calculate Length of Wide-Character String	1715
wcsncat() — Append to Wide-Character String	1716
wcsncmp() — Compare Wide-Character Strings	1718
wcsncpy() — Copy Wide-Character String	1720
wcspbrk() — Locate First Wide Characters in String	1722
wcsrchr() — Locate Last Wide Character in String	1724
wcsrtombs() — Convert Wide-Character String to Multibyte String	1726
wcsspn() — Search for Wide Characters in a String	1729
wcsstr() — Locate a Wide Character Sequence	1731
wctod() — Convert Wide-Character String to a Double Floating-Point	1733
wctok() — Break a Wide-Character String into Tokens	1735
wctol() — Convert a Wide-Character String to a Long Integer	1738
wcstombs() — Convert Wide-Character String to Multibyte Character String	1740
wcstoul() — Convert a Wide-Character String to an Unsigned Long Integer	1742
wcswcs() — Locate Wide-Character Substring in Wide Character String	1744
wcswidth() — Determine the Display Width of a Wide-Character String	1746
wcsxfrm() — Transform a Wide-Character String	1747

wctob() — Convert Wide Character to Byte	1749
wctomb() — Convert Wide Character to Multibyte Character	1751
wctype() — Obtain Handle for Character Property Classification	1753
wcwidth() — Determine the Display Width of a Wide Character	1754
w_getmntent() — Get Information on Mounted File Systems	1756
w_getpsent() — Get Process Data	1759
w_ioctl(), __w_pioctl() — Control of Devices	1762
wmemchr — Locate Wide Character	1764
wmemcmp — Compare Wide Character	1766
wmemcpy — Copy Wide Character	1768
wmemmove — Move Wide Character	1770
wmemset — Set Wide Character	1772
wordexp() — Perform Shell Word Expansions	1774
wordfree() — Perform Shell Word Expansions	1778
__w_pioctl() —Control of Devices	1779
write() — Write Data on a File or Socket	1780
writew() — Write Data on a File or Socket Socket from an Array	1785
__wsinit() — Reinitialize Writeable Static	1788
w_statfs() — Get the File System Status	1789
w_statvfs() — Get the File System Status	1791
y0() - y1() - yn() — Bessel Functions of the Second Kind	1793
Library Functions for the System Programming C Facilities	1795
<b>Appendix A. C/C++ Macros</b>	1797
<b>Appendix B. Notices</b>	1803
Programming Interface Information	1804
Standards	1805
Trademarks	1806
<b>Glossary</b>	1807
<b>Index</b>	1827



---

## Figures

1. Libraries in OS/390 Language Environment . . . . .	7
2. Overlap of C Standards and Extensions . . . . .	64
3. Program Flow of a Fetchable Module . . . . .	371
4. Program Flow of fetchep() . . . . .	384
5. Format Specification for fprintf(), printf(), and sprintf() . . . . .	437
6. Syntax of Conversion Specification for fscanf(), scanf(), and sscanf() . . . . .	466



---

# Introduction

---

## IBM OS/390 C/C++ and Related Publications

This section summarizes the content of the IBM OS/390 C/C++ publications and shows where to find related information in other publications.

---

*Table 1 (Page 1 of 4). OS/390 C/C++ Publications*

Book Title and Number	Key Sections/Chapters in the Book
<i>OS/390 C/C++ Programming Guide</i> , SC09-2362	<p>Guidance information for:</p> <ul style="list-style-type: none"><li>• C/C++ input and output</li><li>• Debugging OS/390 C programs that use input/output</li><li>• Using linkage specifications in C++</li><li>• Combining C and assembler</li><li>• Creating and using DLLs</li><li>• Using threads in an OS/390 UNIX® application</li><li>• Using threads in an OS/390 UNIX application</li><li>• Reentrancy</li><li>• Using the decimal data type in C and C++</li><li>• Handling exceptions, error conditions, and signals</li><li>• Optimizing code</li><li>• Optimizing your C/C++ code with Interprocedural Analysis</li><li>• Network communications under OS/390 UNIX</li><li>• Interprocess communications using OS/390 UNIX</li><li>• Structuring a program that uses C++ templates</li><li>• Using environment variables</li><li>• Using System Programming C facilities</li><li>• Library functions for the System Programming C facilities</li><li>• Using runtime user exits</li><li>• Using the OS/390 C multitasking facility</li><li>• Using other IBM products with OS/390 C/C++ (CICS, CSP, DWS, DB2, GDDM, IMS, ISPF, QMF)</li><li>• Direct-to-SOM support under OS/390 C/C++</li><li>• Internationalization: locales and character sets, code set conversion utilities, mapping variant characters</li><li>• POSIX character set</li><li>• Code point mappings</li><li>• Locales supplied with OS/390 C/C++</li><li>• Charmap files supplied with OS/390 C/C++</li><li>• Examples of charmap and locale definition source files</li><li>• Converting code from code character set IBM-1047</li><li>• Using built-in functions</li><li>• Programming considerations for OS/390 UNIX C/C++</li></ul>

Table 1 (Page 2 of 4). OS/390 C/C++ Publications

Book Title and Number	Key Sections/Chapters in the Book
<i>OS/390 C/C++ User's Guide</i> , SC09-2361	<p>Guidance information for:</p> <ul style="list-style-type: none"> <li>• OS/390 C/C++ examples</li> <li>• Compiler options</li> <li>• Binder options and control statements</li> <li>• Specifying OS/390 Language Environment runtime options</li> <li>• Compiling, IPA Linking, binding, and running OS/390 C/C++ programs</li> <li>• Using precompiled headers</li> <li>• Utilities (Object Library, DLL Rename, CXXFILT, DSECT Conversion, Code Set and Locale, ar and make, BPXBATCH)</li> <li>• Diagnosing problems</li> <li>• Cataloged procedures and REXX EXECs supplied by IBM</li> <li>• Error messages and return codes</li> </ul>
<i>OS/390 C/C++ Language Reference</i> , SC09-2360	<p>Reference information for:</p> <ul style="list-style-type: none"> <li>• The C and C++ Languages</li> <li>• Lexical elements of OS/390 C and OS/390 C++</li> <li>• Declarations, expressions and operators</li> <li>• Implicit type conversions</li> <li>• Functions and statements</li> <li>• Preprocessor directives</li> <li>• C++ classes, class members, and friends</li> <li>• C++ overloading, special member functions, and inheritance</li> <li>• C++ templates and exception handling</li> <li>• OS/390 C and OS/390 C++ compatibility</li> </ul>
<i>OS/390 C/C++ Run-Time Library Reference</i> , SC28-1663	<p>Reference information for:</p> <ul style="list-style-type: none"> <li>• C header files</li> <li>• C Library functions</li> </ul>
<i>OS/390 C Curses</i> , SC28-1907	<p>Reference information for:</p> <ul style="list-style-type: none"> <li>• Curses concepts</li> <li>• Key data types</li> <li>• General rules for characters, renditions, and window properties</li> <li>• General rules of operations and operating modes</li> <li>• Use of macros</li> <li>• Restrictions on block-mode terminals</li> <li>• Curses functional interface</li> <li>• Contents of headers</li> <li>• The terminfo database</li> </ul>
<i>OS/390 C/C++ Compiler and Run-Time Migration Guide</i> , SC09-2359	<p>Guidance and reference information for:</p> <ul style="list-style-type: none"> <li>• Common migration questions</li> <li>• Application executable program compatibility</li> <li>• Source program compatibility</li> <li>• Input and output operations compatibility</li> <li>• Class library migration considerations</li> <li>• Changes between releases of OS/390</li> <li>• C/370 V1 to V2 compiler changes</li> <li>• Other migration considerations</li> </ul>

Table 1 (Page 3 of 4). OS/390 C/C++ Publications

Book Title and Number	Key Sections/Chapters in the Book
<i>OS/390 C/C++ Reference Summary</i> , SX09-1313	<p>Summary tables for:</p> <ul style="list-style-type: none"> <li>• Character set, trigraphs, digraphs, and keywords</li> <li>• Escape sequences, storage classes</li> <li>• Predefined and derived types, type qualifiers</li> <li>• Operator precedence, redirection symbols</li> <li>• fprintf() format, type characters, and flag characters</li> <li>• fscanf() format and type characters</li> <li>• __amrc structure</li> <li>• Hardware exceptions and signals</li> <li>• Compiler return codes</li> <li>• Compiler options</li> <li>• #pragma directives</li> <li>• Library functions</li> <li>• Utilities</li> </ul>
<i>OS/390 C/C++ IBM Open Class Library User's Guide</i> , SC09-2363	<p>Guidance information for:</p> <ul style="list-style-type: none"> <li>• Using the Complex Mathematics Class Library: Review of complex numbers, header files, constructing complex objects, mathematical operators for complex, friend functions for complex, handling complex mathematics errors</li> <li>• Using the I/O Stream Class Library: Introduction, getting started, advanced topics, and manipulators</li> <li>• Using the Collection Class Library: Overview, instantiating and using, element and key functions, tailoring a collection implementation, polymorphic use of collections, support for notifications, exception handling, tutorials, problem solving, compatibility with previous releases, thread safety</li> <li>• Using the Application Support Class Library: Introduction, String classes, Exception and Trace classes, Date and Time classes, controlling threads and protecting data, the IBM Open Class notification framework, Binary Coded Decimal classes</li> </ul>
<i>OS/390 C/C++ IBM Open Class Library Reference</i> , SC09-2364	<p>Reference information for:</p> <ul style="list-style-type: none"> <li>• Complex Mathematics Class Library</li> <li>• I/O Stream Class Library</li> <li>• Collection Class Library</li> <li>• Application Support Class Library</li> </ul>
<i>OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference</i> , SC09-2366	<p>Guidance and reference information for:</p> <ul style="list-style-type: none"> <li>• C++ SOM (RRBC-enabled) versions of Collection and Application Support Class Libraries</li> <li>• Cross-language SOM version of the Collection Class Library</li> </ul>
<i>Debug Tool User's Guide and Reference</i> , SC09-2137	<p>Guidance and reference information for:</p> <ul style="list-style-type: none"> <li>• Preparing to debug programs</li> <li>• Debugging programs</li> <li>• Using Debug Tool in different environments</li> <li>• Language-specific information</li> <li>• Debug Tool reference</li> </ul>
<i>Debug Tool Reference Summary</i> , SX26-3840	Summary information for Debug Tool commands

Table 1 (Page 4 of 4). OS/390 C/C++ Publications

Book Title and Number	Key Sections/Chapters in the Book
APAR and BOOKS files (Shipped with Program materials)	<p>Partitioned data set CBC.SCBCDOC on the product tape contains the members, APAR and BOOKS, which provide additional information for using the IBM OS/390 C/C++ licensed program, including:</p> <ul style="list-style-type: none"> <li>• Isolating reportable problems</li> <li>• Keywords</li> <li>• Preparing an Authorized Program Analysis Report (APAR)</li> <li>• Problem identification worksheet</li> <li>• Maintenance on OS/390</li> <li>• Late changes to OS/390 C/C++ publications</li> </ul>

**Note:** For complete and detailed information on linking and running with OS/390 Language Environment and using the OS/390 Language Environment runtime options, refer to the *OS/390 Language Environment Programming Guide*, SC28-1939. For complete and detailed information on using interlanguage calls, refer to *OS/390 Language Environment Writing Interlanguage Applications*, SC28-1943.

The following table lists the OS/390 C/C++ and related publications. The table groups the publications according to the tasks they describe.

Table 2 (Page 1 of 3). Publications by Task

Tasks	Books
Planning, preparing, and migrating to OS/390 C/C++	<ul style="list-style-type: none"> <li>• <i>OS/390 C/C++ Compiler and Run-Time Migration Guide</i>, SC09-2359</li> <li>• <i>OS/390 Language Environment Customization</i>, SC28-1941</li> <li>• <i>OS/390 UNIX System Services Planning</i>, SC28-1890</li> <li>• <i>OS/390 Planning for Installation</i>, GC28-1726</li> <li>• OS/390 Task Atlas, available on the OS/390 Library page on the World Wide Web (<a href="http://www.s390.ibm.com/os390/bkserv">http://www.s390.ibm.com/os390/bkserv</a>)</li> </ul>
Installing	<ul style="list-style-type: none"> <li>• OS/390 Program Directory</li> <li>• <i>OS/390 Planning for Installation</i>, GC28-1726</li> <li>• <i>OS/390 Language Environment Customization</i>, SC28-1941</li> </ul>
Coding programs	<ul style="list-style-type: none"> <li>• <i>OS/390 C/C++ Run-Time Library Reference</i>, SC28-1663</li> <li>• <i>OS/390 C/C++ Language Reference</i>, SC09-2360</li> <li>• <i>OS/390 C/C++ Reference Summary</i>, SX09-1313</li> <li>• <i>OS/390 C/C++ Programming Guide</i>, SC09-2362</li> <li>• <i>OS/390 Language Environment Concepts Guide</i>, GC28-1945</li> <li>• <i>OS/390 Language Environment Programming Guide</i>, SC28-1939</li> <li>• <i>OS/390 Language Environment Programming Reference</i>, SC28-1940</li> <li>• <i>OS/390 C/C++ IBM Open Class Library User's Guide</i>, SC09-2363</li> <li>• <i>OS/390 C/C++ IBM Open Class Library Reference</i>, SC09-2364</li> <li>• <i>OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference</i>, SC09-2366</li> </ul>

Table 2 (Page 2 of 3). Publications by Task

Tasks	Books
Coding and binding programs with interlanguage calls	<ul style="list-style-type: none"> <li>• <i>OS/390 C/C++ Programming Guide</i>, SC09-2362</li> <li>• <i>OS/390 C/C++ Language Reference</i>, SC09-2360</li> <li>• <i>OS/390 Language Environment Programming Guide</i>, SC28-1939</li> <li>• <i>OS/390 Language Environment Writing Interlanguage Applications</i>, SC28-1943</li> <li>• <i>DFSMS/MVS Program Management</i>, SC26-4916</li> </ul>
Compiling, binding, and running programs	<ul style="list-style-type: none"> <li>• <i>OS/390 C/C++ User's Guide</i>, SC09-2361</li> <li>• <i>OS/390 Language Environment Programming Guide</i>, SC28-1939</li> <li>• <i>OS/390 Language Environment Debugging Guide and Run-Time Messages</i>, SC28-1942</li> <li>• <i>DFSMS/MVS Program Management</i>, SC26-4916</li> <li>• OS/390 Messages Database, available on the OS/390 Library page in the World Wide Web (<a href="http://www.s390.ibm.com/os390/bkserv">http://www.s390.ibm.com/os390/bkserv</a>)</li> </ul>
Compiling and binding applications in the OS/390 UNIX environment	<ul style="list-style-type: none"> <li>• <i>OS/390 C/C++ User's Guide</i>, SC09-2361</li> <li>• <i>OS/390 UNIX System Services User's Guide</i>, SC28-1891</li> <li>• <i>OS/390 UNIX System Services Command Reference</i>, SC28-1892</li> <li>• <i>DFSMS/MVS Program Management</i>, SC26-4916</li> </ul>
Compiling and binding SOM applications with OS/390 SOMobjects	<ul style="list-style-type: none"> <li>• <i>OS/390 SOMobjects Programmer's Guide</i>, GC28-1859</li> <li>• <i>OS/390 C/C++ Programming Guide</i>, SC09-2362</li> <li>• <i>OS/390 C/C++ User's Guide</i>, SC09-2361</li> </ul>
Debugging programs	<ul style="list-style-type: none"> <li>• README file</li> <li>• <i>Debug Tool User's Guide and Reference</i>, SC09-2137</li> <li>• <i>Debug Tool Reference Summary</i>, SX26-3840</li> <li>• <i>OS/390 C/C++ User's Guide</i>, SC09-2361</li> <li>• <i>OS/390 C/C++ Programming Guide</i>, SC09-2362</li> <li>• <i>OS/390 Language Environment Programming Guide</i>, SC28-1939</li> <li>• <i>OS/390 Language Environment Debugging Guide and Run-Time Messages</i>, SC28-1942</li> <li>• <i>OS/390 UNIX System Services Messages and Codes</i>, SC28-1908</li> <li>• <i>OS/390 UNIX System Services User's Guide</i>, SC28-1891</li> <li>• <i>OS/390 UNIX System Services Command Reference</i>, SC28-1892</li> <li>• <i>OS/390 UNIX System Services Programming Tools</i>, SC28-1904</li> </ul>
Using shells and utilities in the OS/390 UNIX environment	<ul style="list-style-type: none"> <li>• <i>OS/390 C/C++ User's Guide</i>, SC09-2361</li> <li>• <i>OS/390 UNIX System Services Command Reference</i>, SC28-1892</li> <li>• <i>OS/390 UNIX System Services Messages and Codes</i>, SC28-1908</li> </ul>
Using sockets library functions in the OS/390 UNIX environment	<ul style="list-style-type: none"> <li>• <i>OS/390 C/C++ Run-Time Library Reference</i>, SC28-1663</li> </ul>

Table 2 (Page 3 of 3). Publications by Task

Tasks	Books
Porting a UNIX Application to OS/390	<ul style="list-style-type: none"> <li>• <i>OS/390 UNIX System Services Porting Guide</i> This guide contains useful information about supported header files and C functions, sockets in an OS/390 UNIX environment, process management, compiler optimization tips, and suggestions for improving the application's performance after it has been ported. The <i>Porting Guide</i> is available as a PDF file which you can download, or as web pages which you can browse, at the following URL: <a href="http://www.s390.ibm.com/unix/bpxa1por.html">http://www.s390.ibm.com/unix/bpxa1por.html</a></li> </ul>
Working in the OS/390 UNIX System Services Parallel Environment	<ul style="list-style-type: none"> <li>• <i>OS/390 UNIX System Services Parallel Environment: Operation and Use</i>, SC33-6697</li> <li>• <i>OS/390 UNIX System Services Parallel Environment: MPI Programming and Subroutine Reference</i>, SC33-6696</li> </ul>
Performing diagnosis and submitting an Authorized Program Analysis Report (APAR)	<ul style="list-style-type: none"> <li>• <i>OS/390 C/C++ User's Guide</i>, SC09-2361</li> <li>• CBC.SCBCDOC(APAR) on OS/390 C/C++ product tape</li> </ul>
Quick reference	<ul style="list-style-type: none"> <li>• <i>OS/390 C/C++ Reference Summary</i>, SX09-1313</li> </ul>
Multimedia Tutorial	<ul style="list-style-type: none"> <li>• For a new way of learning C++ programming, you can order the CD-ROM <i>Experience C++: A Multimedia Tutorial</i>, SK2T-1158. This tutorial runs in DOS.</li> </ul>

**Note:** For information on using the prelinker, see the appendix on prelinking and linking OS/390 C/C++ programs in the *OS/390 C/C++ User's Guide*. As of Release 4, this appendix contains information that was previously in the chapter on prelinking and linking OS/390 C/C++ programs in the *OS/390 C/C++ User's Guide*. It also contains prelinker information that was previously in the *OS/390 C/C++ Programming Guide*.

## Hardcopy Books

The following OS/390 C/C++ books are available in hardcopy:

- *OS/390 C/C++ Run-Time Library Reference*, SC28-1663
- *OS/390 C/C++ User's Guide*, SC09-2361
- *OS/390 C/C++ Programming Guide*, SC09-2362
- *OS/390 C/C++ Reference Summary*, SX09-1313
- *OS/390 C/C++ IBM Open Class Library User's Guide*, SC09-2363
- *OS/390 C Curses*, SC28-1907
- *OS/390 C/C++ Compiler and Run-Time Migration Guide*, SC09-2359
- *Debug Tool User's Guide and Reference*, SC09-2137

You can purchase these books on their own, or as part of a set. You receive the *OS/390 C/C++ Compiler and Run-Time Migration Guide*, SC09-2359 at no charge. Feature code 8009 includes the remaining books.



---

## Softcopy Books

All of the OS/390 C/C++ publications (except for the *OS/390 C/C++ Reference Summary*) are available in softcopy book format. The books are available on the tape that accompanies the OS/390 product, and on a CD-ROM called the *IBM Online Library Omnibus Edition: OS/390 Collection*, SK2T-6700.

To read the softcopy books, the BookManager Read (Program 5684-062, 5695-046) licensed program must be available on your operating system. BookManager Read provides access to online information as an alternative to hard copy documents. You can read, search, make notes, and select sections of text to print.

Also available are BookManager Read/DOS (Program 73F6-022) for the DOS operating system, and BookManager Read/2 (Program 73F6-023) for the OS/2 operating system. With these products, you can download online books to your workstation and read them.

If your system has BookManager Read installed, you can enter the command BOOKMGR to start BookManager and display a list of books available to you. If you know the name of the book that you want to view, you can use the OPEN command to open the book directly.

**Note:** If your workstation does not have graphics capability, BookManager Read cannot correctly display some characters, such as arrows and brackets.

You can also browse the books on the World Wide Web by clicking on "The Library" link on the OS/390 home page. The URL for this page is:

<http://www.s390.ibm.com/os390/index.html>

---

## Softcopy Examples

Most of the larger examples in the following books are available in machine-readable form:

- *OS/390 C/C++ Language Reference*, SC09-2360
- *OS/390 C/C++ User's Guide*, SC09-2361
- *OS/390 C/C++ Programming Guide*, SC09-2362
- *OS/390 C/C++ IBM Open Class Library User's Guide*, SC09-2363
- *OS/390 C/C++ IBM Open Class Library Reference*, SC09-2364
- *OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference*, SC09-2366

In the following books, a label on an example indicates that the example is distributed in softcopy. The label is the name of a member in the data sets CBC.SCBCSAM or CBC.SCLBSAM. The labels have the form CBCxyyy or CLBxyyy, where x refers to a publication:

- R and X refer to the *OS/390 C/C++ Language Reference*, SC09-2360
- G refers to the *OS/390 C/C++ Programming Guide*, SC09-2362
- U refers to the *OS/390 C/C++ User's Guide*, SC09-2361
- A refers to the *OS/390 C/C++ IBM Open Class Library User's Guide*, SC09-2363

Examples labelled as CBCxyyy appear in the *OS/390 C/C++ Language Reference*, the *OS/390 C/C++ Programming Guide*, and the *OS/390 C/C++ User's Guide*. Examples labelled as CLBxyyy appear in the *OS/390 C/C++ IBM Open Class Library User's Guide*.

An exception applies to the example names for the Collection Class Library which do not follow a naming convention. These examples are in the *OS/390 C/C++ IBM Open Class Library Reference*, SC09-2364 and in the *OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference*, SC09-2366. For the *OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference*, SC09-2366, the label refers to a member name in the data set *CBC.SCLBXSM*.

---

## OS/390 C/C++ on the World Wide Web

Additional information on OS/390 C/C++ is available on the World Wide Web. The URL for the OS/390 C/C++ home page is:

<http://www.software.ibm.com/ad/c390/index.html>

This page contains late-breaking information about the OS/390 C/C++ product, including the compiler, the class libraries, and utilities. It also contains a tutorial on the source level interactive debugger. There are links to other useful information, such as the OS/390 C/C++ information library and the libraries of other OS/390 elements that are available on the Web. The OS/390 C/C++ home page also contains information on active Beta programs, samples that you can download, C/370 product newsletters, and links to other related Web sites.

---

## C/C++ News...

IBM also publishes the *C/370 Compiler Newsletter*. This free newsletter keeps subscribers up to date on the latest product releases. It also provides coding hints and tips, questions and answers, and news about C/370 products and IBM OS/390 C/C++.

To take advantage of this free publication, send your name, full mailing address, and phone number, as follows:

- Send a message electronically to the following network ID :

- Internet: [inetc370@ca.ibm.com](mailto:inetc370@ca.ibm.com)
- IBMMAIL: [ibmmail\(caibmrxz\)](mailto:ibmmail(caibmrxz)@ca.ibm.com)

- Mail your request to:

EDITOR, C/370 Compiler Newsletter  
IBM Canada Ltd. Laboratory  
9/604/895/TOR  
895 Don Mills Road  
NORTH YORK ONTARIO CANADA M3C 1W3

---

## How to Read the Syntax Diagrams

This book describes the syntax for commands, directives, and statements, using the following structure:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

A double right arrowhead indicates the beginning of a command, directive, or statement. A single right arrowhead indicates that it is continued on the next line. In the following diagrams, "statement" represents a command, directive, or statement.

►►—statement—►

The following indicates a continuation; the opposing arrowheads indicate the end of a command, directive, or statement.

►—statement—◄◄

Diagrams of syntactical units other than complete commands, directives, or statements look like this:

►—statement—►

- Required items are on the horizontal line (the main path).

►►—statement—*required\_item*—◄◄

- Optional items are below the main path.

►►—statement—  
└─*optional\_item*—◄◄

- If you can choose from two or more items, they are vertical in a stack.

If you *must* choose one of the items, one item of the stack is on the main path.

►►—statement—  
└─*required\_choice1*—  
└─*required\_choice2*—◄◄

If choosing one of the items is optional, the entire stack is below the main path.

►►—statement—  
└─*optional\_choice1*—  
└─*optional\_choice2*—◄◄

- An arrow that returns to the left above the main line indicates an item that you can repeat.

►►—statement—  
└─*repeatable\_item*—◄◄

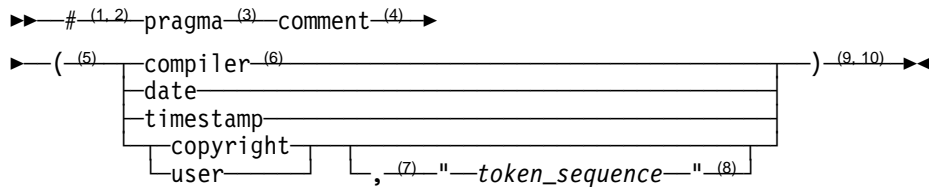
A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

- Keywords are not italicized, and should be entered exactly as shown (for example, `pragma`). You must spell keywords exactly as shown in the syntax diagram. Variables are in lowercase italics (in hardcopy), for example, *identifier*. They represent user-supplied names or values.

- If the syntax diagram shows punctuation marks, parentheses, arithmetic operators, or other nonalphanumeric characters, you must enter them as part of the syntax.

**Note:** You do not always require the white space between tokens. You should, however, include at least one blank space between tokens unless otherwise specified.

The following syntax diagram example shows the syntax for the `#pragma comment` directive.



**Notes:**

- <sup>1</sup> This is the start of the syntax diagram.
- <sup>2</sup> The symbol `#` must appear first.
- <sup>3</sup> The keyword `pragma` must follow the `#` symbol.
- <sup>4</sup> The keyword `comment` must follow the keyword `pragma`.
- <sup>5</sup> An opening parenthesis must follow the keyword `comment`.
- <sup>6</sup> The comment type must be entered only as one of the following: `compiler`, `date`, `timestamp`, `copyright`, or `user`.
- <sup>7</sup> If the comment type is `copyright` or `user`, and an optional character string is following, a comma must be present after the comment type.
- <sup>8</sup> A character string must follow the comma. The character string must be enclosed in double quotation marks.
- <sup>9</sup> A closing parenthesis is required.
- <sup>10</sup> This is the end of the syntax diagram.

The following examples of the `#pragma comment` directive are syntactically correct according to the diagram above:

```

#pragma comment(date)
#pragma comment(user)
#pragma comment(copyright,"This text will appear in the module")

```

---

# Summary of Changes

## Summary of Changes for SC28-1663-05 OS/390 Version 2 Release 8

The following items are new in this book:

- Added feature test macros:
  - `_XOPEN_SOURCE 500`
  - `__LIBASCII`
- Added functions:
  - `__osenv()`
  - `_SET_THLIIPADDR()`
  - `__w_pioctl()`

This book includes terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

## Summary of Changes for SC28-1663-04 OS/390 Version 2 Release 7

The following items are new in this book:

- Added functions:
  - `aio_cancel()`
  - `aio_error()`
  - `aio_read()`
  - `aio_return()`
  - `aio_suspend()`
  - `aio_write()`
  - `__certificate()`
  - `copysign()`
  - `finite()`
  - `fp_clr_flag()`
  - `fp_raise_xcp()`
  - `fp_read_flag()`
  - `fp_read_rnd()`
  - `fp_swap_rnd()`
  - `__isBFP()`
  - `pthread_mutexattr_getpshared()`
  - `pthread_mutexattr_gettype()`
  - `pthread_mutexattr_setpshared()`
  - `pthread_mutexattr_settype()`
  - `pthread_rwlockattr_destroy()`
  - `pthread_rwlockattr_getpshared()`
  - `pthread_rwlockattr_init()`
  - `pthread_rwlockattr_setpshared()`
  - `pthread_rwlock_destroy()`
  - `pthread_rwlock_init()`

- pthread\_rwlock\_rdlock()
- pthread\_rwlock()\_tryrdlock
- pthread\_rwlock\_trywrlock()
- pthread\_rwlock\_wrlock()
- pthread\_rwlock\_unlock()
- send\_file()
- sigtimedwait()
- sigwaitinfo()
- \_\_ucreate()
- \_\_ufree()
- \_\_uheapreport()
- \_\_umalloc()

**Summary of Changes  
for SC28-1663-03  
OS/390 Version 2 Release 6**

The following items are new in this book:

- Added functions:
  - accept\_and\_recv()
  - alloca()
  - \_\_login()
  - \_\_opendir2()
  - \_\_open\_stat()
  - \_\_pid\_affinity()
  - \_\_readdir2()
  - \_\_sigactionset()

---

## Volume 1 - Part 1. About IBM OS/390 C/C++

The C/C++ feature of the IBM OS/390 licensed program provides support for C and C++ application development on the OS/390 platform. The C/C++ feature is based on the C/C++ for MVS/ESA product.

IBM OS/390 C/C++ includes:

- A C compiler (referred to as the OS/390 C compiler)
- A C++ compiler (referred to as the OS/390 C++ compiler)
- A set of C++ class libraries
- Application Support Class and Collection Class Library source
- A mainframe interactive Debug Tool (optional)
- A set of utilities for C/C++ application development

IBM offers the C language on other platforms, such as the AIX, IBM Operating System/2 (OS/2), IBM Operating System/400 Version 3 (OS/400), Sun Solaris, VM/ESA, VSE/ESA, and Windows® operating systems. The AIX, OS/2, OS/400, Sun Solaris, and Windows operating systems also offer the C++ language.

---

### Changes for Version 2 Release 8

The Language Environment C/C++ Run-Time library has made the following changes for this release:

- Added code pages to support the euro, the monetary unit of the European Monetary Union (EMU).
- Added support for Unicode through UTF-8. Interoperability of UTF-8 (ASCII) and Unicode (EBCDIC) data are supported through data converters to and from UTF-8 and UCS-2.
- Added VSAM Record Level Sharing support for the sharing of VSAM data at the record level, using the locking and caching functions of the coupling facility hardware.

---

### The C/C++ Compilers

The following sections describe the C and C++ languages and the OS/390 C/C++ compilers.

#### The C Language

The C language is a general purpose, versatile, and functional programming language, which allows a programmer to create applications quickly and easily. C provides high-level control statements and data types as do other structured programming languages. It also provides many of the benefits of a low-level language.

## The C++ Language

The C++ language is based on the C language, but incorporates support for object-oriented concepts. Refer to *OS/390 C/C++ Language Reference* for a detailed description of the differences between OS/390 C++ and OS/390 C.

The C++ language introduces classes, which are user-defined data types that may contain data definitions and function definitions. You can use classes from established class libraries, develop your own classes, or derive new classes from existing classes by adding data descriptions and functions. New classes can inherit properties from one or more classes. Not only do classes describe the data types and functions available, but they can also hide (encapsulate) the implementation details from user programs. An object is an instance of a class.

The C++ language also provides templates and other features that include access control to data and functions, and better type checking and exception handling. It also supports polymorphism and the overloading of operators.

## Common Features of the OS/390 C and C++ Compilers

The C or C++ compilers offer many features to help your work:

- Optimization support.
  - Algorithms to take advantage of S/390 architecture to get better optimization for speed and use of computer resources through the OPTIMIZE and IPA compile-time options.
  - The OPTIMIZE compile-time option to instruct the compiler to optimize the machine instructions it generates, to produce faster-running object code, thereby optimizing application performance at run time.
  - Interprocedural Analysis (IPA), to perform optimizations across compilation units, thereby optimizing application performance at run time.
  - The precompiled header facility, to save information from one compilation unit for use in another or to reuse information when re-compiling the source compilation unit, thereby improving performance at compile time.
- DLLs (dynamic link libraries) to reduce application size, and dynamically link to exported variables and functions at run time.

IBM OS/390 C/C++ provides support for generating DLLs in a way similar to the way OS/2 generates DLLs. DLLs allow a function reference or a variable reference in one executable to use a definition located in another executable at run time. You can use both load-on-reference and load-on-demand DLLs. When your program calls a DLL function, or references a DLL, IBM OS/390 C/C++ provides a load-on-reference DLL. Your application code explicitly controls load-on-demand DLLs at the source level.

You can use DLLs to split applications into smaller modules and improve system memory usage. DLLs also offer more flexibility for building, packaging, and redistributing applications.

- Full program reentrancy.

With reentrancy, many users can simultaneously run a program. A reentrant program uses less storage if it is stored in the LPA (link pack area) or ELPA (extended link pack area) and simultaneously run by multiple users. It also reduces processor I/O when the program starts up, and improves program per-



formance by reducing the transfer of data to auxiliary storage. OS/390 C programmers can design programs that are naturally reentrant. For those programs that are not naturally reentrant, C programmers can use constructed reentrancy. To do this, compile programs with the RENT option and use the program management binder supplied with OS/390, or the OS/390 Language Environment Prelinker (prelinker) and program management binder. The OS/390 C++ compiler always ensures that C++ programs are reentrant.

- Locale-based internationalization support derived from the IEEE POSIX 1003.2-1992 standard. Also derived from the X/Open CAE Specification, System Interface Definitions, Issue 4 and Issue 4 Version 2. This allows programmers to use locales to specify language/country characteristics for their applications.
- The ability to call and be called by other languages such as assembler, COBOL, PL/1, and Fortran, to enable programmers to integrate OS/390 C/C++ code with existing applications.
- Exploitation of OS/390 and OS/390 UNIX technology.

OS/390 UNIX is an IBM implementation of the open operating system environment, as defined in the XPG4 and POSIX standards.

- When used with OS/390 UNIX and OS/390 Language Environment, support for the following standards at the system level:
  - A subset of the extended multibyte and wide character functions as defined by the Programming Language C Amendment 1. This is ISO/IEC 9899:1990/Amendment 1:1994(E)
  - ISO/IEC 9945-1:1990(E)/IEEE POSIX 1003.1-1990
  - A subset of IEEE POSIX 1003.1a, Draft 6, July 1991
  - IEEE Portable Operating System Interface (POSIX) Part 2, P1003.2
  - A subset of IEEE POSIX 1003.4a, Draft 6, February 1992 (the IEEE POSIX committee has renumbered POSIX.4a to POSIX.1c)
  - X/Open CAE Specification, System Interfaces and Headers, Issue 4 Version 2
  - A subset of IEEE 754-1985 (R1990) IEEE Standard for Binary Floating-Point Arithmetic (ANSI), as applicable to the S/390 environment.
  - X/Open CAE Specification, Network Services, Issue 4
- Year 2000 support.

## OS/390 C Compiler Specific Features

In addition to the features common to OS/390 C/C++, the OS/390 C compiler provides you with the following capabilities:

- The ability to write portable code that conforms to the following standards:
  - All elements of the ISO standard ISO/IEC 9899:1990 (E)
  - ANSI/ISO 9899:1990[1992] (formerly ANSI X3.159-1989 C)
  - X/Open Specification Programming Language Issue 3, Common Usage C
  - FIPS-160

- System programming capabilities, which allow you to use OS/390 C in place of assembler
- Additional optimization capabilities through the `INLINE` compile-time option
- Extensions of the standard definitions of the C language to provide programmers with support for the OS/390 environment, such as fixed-point (packed) decimal data support

## Features That Are Specific to the OS/390 C++ Compiler

In addition to the features common to OS/390 C/C++, the OS/390 C++ compiler provides you with the following:

- An implementation based on the definition of the language that is contained in the Draft Proposal International Standard for Information Systems— Programming Language C++ (X3J16/92-00091). The OS/390 C++ compiler also conforms to a subset of the C++ ANSI/ISO (Draft) Standard (X3J16/93-0062).
- System Object Model (SOM) support, through the SOM Interface Definition Language (IDL) compiler available with OS/390 SOMobjects. You can use the IDL compiler and associated emitters to create language-specific bindings that define the interface to a SOM object. This enables OS/390 C++ programs to share SOM objects with other languages. In addition, SOM enables release-to-release binary compatibility.

With Direct-to-SOM (DTS) support in the OS/390 C++ compiler, you can generate SOM objects directly from C++ code. You do not need to create and process the IDL first. You can write virtually the same code you do when creating C++ objects.

**Note:** The OS/390 C++ compiler no longer supports IDL generation through the IDL compile-time option. This option instructed the compiler to generate IDL. Mixed-language or distributed object applications used IDL. If you need IDL for your applications, you should now code it yourself instead of generating it through the IDL compile option.

- C++ template support and exception handling consistent with VisualAge C++ product implementations.

---

## Utilities

The OS/390 C/C++ compilers provide the following utilities:

- The Object Library Utility to update partitioned data set (PDS) libraries of object modules and Interprocedural Analysis (IPA) object modules
- The DLL Rename Utility to make selected DLLs a unique component of the applications with which they are packaged
- The CXXFILT Utility to map OS/390 C++ mangled names to the original source
- The localedef Utility to read the locale definition file and produce a locale object that the locale-specific library functions can use
- The DSECT Conversion Utility to convert descriptive assembler DSECTs into OS/390 C/C++ data structures
- The C/C++ Model Tool to provide online help for C/C++ `#pragma` directives and runtime library functions. These functions are other than the C Curses functions, and are at the level that is supplied in OS/390 Release 2

---

## Class Libraries

IBM OS/390 C/C++ provides a base set of class libraries, called C/C++ IBM Open Class, which is consistent with that available in other members of the VisualAge C++ product family. These class libraries are:

- The I/O Stream Class Library

The I/O Stream Class Library lets you perform input and output (I/O) operations independent of physical I/O devices or data types that are used. You can code sophisticated I/O statements easily and clearly, and define input and output for your own data types. You can improve the maintainability of programs that use input and output by using the I/O Stream Class Library.

- The Complex Mathematics Class Library

The Complex Mathematics Class Library lets you manipulate and perform standard arithmetic on complex numbers. Scientific and technical fields use complex numbers.

- The Application Support Class Library

The Application Support Class Library provides the basic abstractions that are needed during the creation of most C++ applications, including String, Date, and Time.

The Application Support Class library is available in a C++ SOM version as well as the regular C++ native version.

- The Collection Class Library

The Collection Class Library implements a wide variety of classical data structures such as stack, tree, list, hash table, and so on. Most programs use collections. You can develop programs without having to define every collection. Programmers can start programming by using a high level of abstraction, and later replace an abstract data type with the appropriate concrete implementation. Each abstract data type has a common interface for all of its implementations. The Collection Class Library provides programmers with a consistent set of building blocks from which they can derive application objects. The library design exploits features of the C++ language such as exception handling and template support.

The Collection Class Library is available in a C++ SOM and a cross-language SOM version, as well as the regular C++ native version.

All of the libraries that are described above are thread-safe, except the cross-language SOM version of the Collection Class Library.

All of the libraries that are described above are available in both static and DLL formats. OS/390 C/C++ packages the Application Support Class and Collection Class libraries together in a single DLL. For compatibility, separate side-decks are available for the Application Support Class and Collection Class libraries, in addition to the side-deck available for the combined library.

**Note:** Retroactive to OS/390 Version 1 Release 3, the IBM Open Class Library is licensed with the base operating system. This enables applications to use this library at run time without having to license the OS/390 C/C++ compiler feature(s) or to use the DLL Rename Utility.

## Class Library Source

The Class Library Source consists of the following:

- Application Support Class Library source code
- Collection Class Library source code (C++ native and C++ SOM only)
- Instructions for building the Application Support Class and Collection Class Libraries in C++ native (static and DLL) versions
- Instructions for building the Application Support Class and Collection Class Libraries in C++ SOM (static and DLL) versions
- Class Library Language Environment message file source
- Instructions for building the Class Library Language Environment message files

---

## The Debug Tool

IBM OS/390 C/C++ supports program development by using a mainframe interactive Debug Tool. This optionally available tool allows you to debug applications in their native host environment, such as CICS/ESA, IMS/ESA, DB2, and so on. The Debug Tool provides the following support and function:

- Step mode
- Breakpoints
- Monitor
- Frequency analysis
- Dynamic patching

You can record the debug session in a log file, and replay the session. You can also use the Debug Tool to help capture test cases for future program validation or to further isolate a problem within an application.

You can specify either data sets or hierarchical file system (HFS) files as source files.

---

## OS/390 Language Environment

IBM OS/390 C/C++ exploits the C/C++ runtime environment and library of runtime services available with OS/390 Language Environment (formerly Language Environment for MVS & VM, Language Environment/370 and LE/370).

OS/390 Language Environment consists of four language-specific runtime libraries, and Base Routines and Common Services; see Figure 1 on page 7. OS/390 Language Environment establishes a common runtime environment and common runtime services for language products, user programs, and other products.

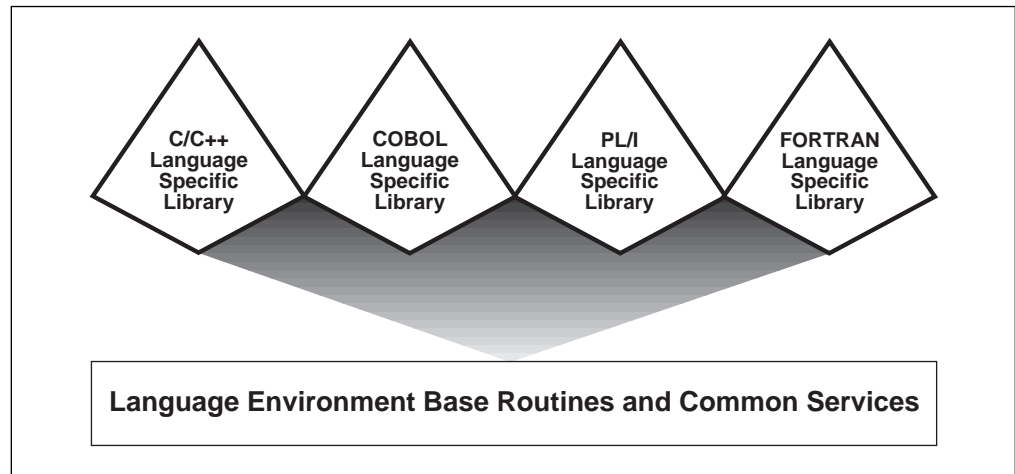


Figure 1. Libraries in OS/390 Language Environment

The common execution environment is composed of data items and services that are included in library routines available to an application that runs in the environment. The OS/390 Language Environment provides a variety of services:

- Services that satisfy basic requirements common to most applications. These include support for the initialization and termination of applications, allocation of storage, interlanguage communication (ILC), and condition handling.
- Extended services that are often needed by applications. OS/390 C/C++ contains these functions within a library of callable routines, and include interfaces to operating system functions and a variety of other commonly used functions.
- Runtime options that help in the execution, performance, and diagnosis of your application.
- Access to operating system services; OS/390 UNIX services are available to an application programmer or program through the OS/390 C/C++ language bindings.
- Access to language-specific library routines, such as the OS/390 C/C++ library functions.

## The Program Management Binder

The binder provided with OS/390 combines the object modules, load modules, and program objects comprising an OS/390 application. It produces a single output program object or load module that you can load for execution. The binder supports all C and C++ code, provided that you store the output program in a PDSE (Partitioned Data Set Extended) member or an HFS file.

If you cannot use a PDSE member or HFS file, and your program contains C++ code, or C code that is compiled with any of the RENT, LONGNAME, DLL or IPA compile-time options, you must use the prelinker.

Using the binder without using the prelinker has the following advantages:

- Faster rebinds when recompiling and rebinding a few of your source files
- Rebinding at the single compile unit level of granularity (except when you use the IPA compile-time option)

- Input of object modules, load modules, and program objects
- Improved long name support:
  - Long names do not get converted into prelinker generated names
  - Long names appear in the binder maps, enabling full cross-referencing
  - Variables do not disappear after prelink
  - Fewer steps in the process of producing your executable program

The prelinker provided with OS/390 Language Environment combines the object modules comprising an OS/390 C/C++ application and produces a single object module. You can link-edit the object module into a load module (which is stored in a PDS), or bind it into a load module or a program object stored in a PDS, or a PDSE or HFS file.

---

## OS/390 UNIX System Services (OS/390 UNIX)

OS/390 UNIX provides capabilities under OS/390 to make it easier to implement or port applications in an open, distributed environment. OS/390 UNIX Services are available to OS/390 C/C++ application programs through the C/C++ language bindings available with OS/390 Language Environment.

Together, the OS/390 UNIX Services, OS/390 Language Environment, and OS/390 C/C++ compilers provide an application programming interface that supports industry standards.

OS/390 UNIX provides support for both existing OS/390 applications and new OS/390 UNIX applications:

- C programming language support as defined by ISO/ANSI C
- C++ programming language support
- C language bindings as defined in the IEEE 1003.1 and 1003.2 standards; subsets of the draft 1003.1a and 1003.4a standards; X/Open CAE Specification: System Interfaces and Headers, Issue 4, Version 2, which provides standard interfaces for better source code portability with other conforming systems; and X/Open CAE Specification, Network Services, Issue 4, which defines the X/Open UNIX descriptions of sockets and X/Open Transport Interface (XTI)
- OS/390 UNIX Extensions that provide OS/390-specific support beyond the defined standards
- The OS/390 UNIX Shell and Utilities feature, which provides:
  - A shell, based on the Korn Shell and compatible with the Bourne Shell
  - Tools and utilities that conform to the *X/Open Single UNIX Specification*, also known as *X/Open Portability Guide (XPG) Version 4, Issue 2*, and provide OS/390 support. The following utilities are included:

<b>ar</b>	Creates and maintains library archives
<b>BPXBATCH</b>	Allows you to submit batch jobs that run shell commands, scripts, or OS/390 C/C++ executable files in HFS files from a shell session
<b>c89</b>	Compiles, assembles, and binds OS/390 UNIX C applications

- |               |  |
|---------------|--|
| <b>gencat</b> | Merges the message text source files Messagefile (usually *.msg) into a formatted message Catalogfile (usually *.cat)  |
| <b>lex</b>    | Automatically writes large parts of a lexical analyzer based on a description that is supplied by the programmer   |
| <b>make</b>   | Helps you manage projects containing a set of interdependent files, such as a program with many OS/390 C/C++ source and object files, keeping all such files up to date with one another |
| <b>yacc</b>   | Allows you to write compilers and other programs that parse input according to strict grammar rules  |
- Support for other utilities such as:
- |                  |   |
|------------------|---|
| <b>c++</b>       | Compiles, assembles, and binds OS/390 UNIX C++ applications                                       |
| <b>mkcatdefs</b> | Preprocesses a message source file for input to the gencat utility                                |
| <b>runcat</b>    | Invokes mkcatdefs and pipes the message catalog source data (the output from mkcatdefs) to gencat |
| <b>dspcat</b>    | Displays all or part of a message catalog   |
| <b>dspmsg</b>    | Displays a selected message from a message catalog  |
- The OS/390 UNIX Debugger feature, which provides the dbx interactive symbolic debugger for OS/390 UNIX applications
  - OS/390 UNIX, which provides access to a hierarchical file system (HFS), with support for the POSIX.1 and XPG4 standards
  - OS/390 C/C++ I/O routines, which support using HFS files, standard OS/390 data sets, or a mixture of both
  - Application threads (with support for a subset of POSIX.4a)
  - Support for OS/390 C/C++ DLLs

OS/390 UNIX offers program portability across multivendor operating systems, with support for POSIX.1, POSIX.1a (draft 6), POSIX.2, POSIX.4a (draft 6), and XPG4.2.

To application developers who have worked with other UNIX environments, the OS/390 UNIX Shell and Utilities are a familiar environment for C/C++ application development. If you are familiar with existing MVS development environments, you may find that the OS/390 UNIX environment can enhance your productivity. Refer to the *OS/390 UNIX System Services User's Guide* for more information on the Shell and Utilities.

---

## OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions

Most OS/390 UNIX C functions are available at all times. However, to use some OS/390 UNIX C functions, you must run an OS/390 C/C++ program on a system where the OS/390 UNIX kernel is available and active. In some situations, you must also specify the POSIX(ON) runtime option. This is required for the POSIX.4a threading functions, and the system and signal handling functions where the behavior is different between POSIX/XPG4 and ANSI. Refer to the *OS/390 C/C++*

*Run-Time Library Reference* for more information about requirements for each function.

You can invoke an OS/390 C/C++ program that uses OS/390 UNIX C functions using the following methods:

- Directly from the OS/390 UNIX Shell.
- From another program, or from the OS/390 UNIX Shell, using one of the `exec` family of functions, or the BPXBATCH utility from TSO or MVS batch.
- Using the POSIX `system()` call.
- Directly through TSO or MVS batch without the use of the intermediate BPXBATCH utility. In some cases, you may require the POSIX(0N) runtime option.

---

## Input and Output

The C/C++ runtime library that supports the OS/390 C/C++ compiler supports different input and output (I/O) interfaces, file types, and access methods. The C++ I/O Stream Class Library provides additional support.

### I/O Interfaces

The C/C++ runtime library supports the following I/O interfaces:

#### C Stream I/O

This is the default and the ANSI-defined I/O method. This method processes all input and output by character.

#### Record I/O

The library can also process your input and output by record. A record is a set of data that is treated as a unit. It can also process VSAM data sets by record. Record I/O is an OS/390 C/C++ extension to the ANSI standard.

#### TCP/IP Sockets I/O

OS/390 UNIX provides support for an enhanced version of an industry-accepted protocol for client/server communication that is known as *sockets*. A set of C language functions provides support for OS/390 UNIX sockets. OS/390 UNIX sockets correspond closely to the sockets that are used by UNIX applications that use the Berkeley Software Distribution (BSD) 4.3 standard (also known as OE sockets). The slightly different interface of the X/Open CAE Specification, Networking Services, Issue 4, is supplied as an additional choice. This interface is known as X/Open Sockets.

The OS/390 UNIX socket application program interface (API) provides support for both UNIX domain sockets and Internet domain sockets. UNIX domain sockets, or *local sockets*, allow interprocess communication within OS/390 independent of TCP/IP. Local sockets behave like traditional UNIX sockets and allow processes to



communicate with one another on a single system. With Internet sockets, application programs can communicate with others in the network using TCP/IP.

In addition, the C++ I/O Stream Library supports formatted I/O in C++. You can code sophisticated I/O statements easily and clearly, and define input and output for your own data types. This helps improve the maintainability of programs that use input and output.

## File Types

In addition to conventional files, such as sequential files and partitioned data sets, the C/C++ runtime library supports the following file types:

### Virtual Storage Access Method (VSAM) Data Sets

OS/390 C/C++ has native support for three types of VSAM data organization:

- Key-sequenced data sets (KSDS). Use KSDS to access a record through a key within the record. A key is one or more consecutive characters that are taken from a data record that identifies the record.
- Entry-sequenced data sets (ESDS). Use ESDS to access data in the order it was created (or in the reverse order).
- Relative-record data sets (RRDS). Use RRDS for data in which each item has a particular number (for example, a telephone system with a record associated with each number).

For more information on how to perform I/O operations on these VSAM file types, see the *OS/390 C/C++ Programming Guide*.

### Hierarchical File System Files

When you are running under MVS, TSO (batch and interactive), or IMS environments, OS/390 C/C++ recognizes a Hierarchical File System (HFS) file. The name specified on the `fopen()` or `freopen()` call has to conform to certain rules (described in the *OS/390 C/C++ Programming Guide*). You can create regular HFS files, special character HFS files, or FIFO HFS files. You can also create links or directories.

### Memory Files

Memory files are temporary files that reside in memory. For improved performance, you can direct input and output to memory files rather than to devices. Since memory files reside in main storage and only exist while the program is executing, you primarily use them as work files. You can access memory files across load modules through calls to non-POSIX `system()` and `C fetch()`; they exist for the life of the root program. Standard streams can be redirected to memory files on a non-POSIX `system()` call using command line redirection.

## Hiperspace Expanded Storage

Large memory files can be placed in Hiperspace expanded storage to free up some of your home address space for other uses. Hiperspace expanded storage or high performance space is a range of up to 2 gigabytes of contiguous virtual storage space. A program can use this storage as a buffer (1 gigabyte =  $2^{30}$  bytes).

## Additional I/O Features

IBM OS/390 C/C++ provides additional I/O support through the following features:

- User error handling for serious I/O failures (SIGIOERR)
- Improved sequential data access performance through enablement of the DFSMS/MVS support for 31-bit sequential data buffers and sequential data striping on extended format data sets
- Full support of PDS/Es on OS/390 — including support for multiple members opened for write
- Overlapped I/O support under OS/390 (NCP, BUFNO)
- Multibyte character I/O functions
- Fixed-point (packed) decimal data type support in formatted I/O functions
- Support for multiple volume data sets that span more than one volume of DASD or tape
- Support for Generation Data Group I/O

---

## The System Programming C Facility

The System Programming C (SP C) facility allows you to build applications that require no dynamic loading of OS/390 Language Environment libraries. It also allows you to tailor your application to better utilize the low-level services available on your operating system. SP C offers a number of advantages:

- You can develop applications that you can execute in a customized environment rather than with OS/390 Language Environment services. Note that if you do not use OS/390 Language Environment services, only some built-in functions and a limited set of C/C++ runtime library functions are available to you.
- You can substitute the OS/390 C language in place of assembler language when writing system exit routines, by using the interfaces that are provided by SP C.
- SP C lets you develop applications featuring a user-controlled environment, in which an OS/390 C environment is created once and used repeatedly for C function execution from other languages.
- You can utilize co-routines, by using a two-stack model to write application service routines. In this model, the application calls on the service routine to perform services independently of the user. The application is then suspended when control is returned to the user application.

---

## Interaction with Other IBM Products

When you use OS/390 C/C++, you can write programs that utilize the power of other IBM products and subsystems:

- Cross System Product (CSP)

Cross System Product/Application Development (CSP/AD) is an application generator that provides ways to interactively define, test, and generate application programs to improve productivity in application development. Cross System Product/Application Execution (CSP/AE) takes the generated program and executes it in a production environment.

**Note:** You cannot compile CSP applications with the OS/390 C++ compiler. However, your OS/390 C++ program can use interlanguage calls (ILC) to call OS/390 C programs that access CSP.

- Customer Information Control System (CICS)

You can use the CICS/ESA Command-Level Interface to write C/C++ application programs. The CICS Command-Level Interface provides data, job, and task management facilities that are normally provided by the operating system.

**Note:** Code preprocessed with CICS/ESA versions prior to V4 R1 is not supported for OS/390 C++ applications. OS/390 C++ code preprocessed on CICS/ESA V4 R1 cannot run under CICS/ESA V3 R3.

- DATABASE 2 (DB2)

DB2 programs manage data that is stored in relational data bases. The IBM DATABASE 2 licensed program runs on OS/390.

You can access the data by using a structured set of queries that are written in Structured Query Language (SQL). The DB2 program uses SQL statements that are embedded in the program. The SQL translator (DB2 preprocessor) translates the embedded SQL into host language statements that perform the requested functions. The OS/390 C/C++ compilers compile the output of the SQL translator. The DB2 program processes a request, and processing returns to the application.

- Data Window Services (DWS)

The Data Window Services (DWS) part of the Callable Services Library allows your OS/390 C or OS/390 C++ program to manipulate temporary data objects that are known as TEMPSPACE and VSAM linear data sets.

- Information Management System (IMS)

The Information Management System/Enterprise Systems Architecture (IMS/ESA) product provides support for hierarchical databases.

- Interactive System Productivity Facility (ISPF)

OS/390 C/C++ provides access to the Interactive System Productivity Facility (ISPF) Dialog Management Services. A dialog is the interaction between a person and a computer. The dialog interface contains display, variable, message, and dialog services as well as other facilities that are used to write interactive applications.

- Graphical Data Display Manager (GDDM)

GDDM provides a comprehensive set of functions to display and print applications most effectively:

- A windowing system that the user can tailor to display selected information
- Support for presentation and keyboard interaction
- Comprehensive graphics support
- Fonts — including support for double-byte character set (DBCS)
- Business image support
- Saving and restoring graphics pictures
- Support for many types of display terminals, printers, and plotters
- Query Management Facility (QMF)

OS/390 C supports the Query Management Facility (QMF), a query and report writing facility, which allows you to write applications through a callable interface. You can create applications to perform a variety of tasks, such as data entry, query building, administration aids, and report analysis.

---

## Additional Features of OS/390 C/C++

Feature	Description
Multibyte Character Support	OS/390 C/C++ supports multibyte characters for those national languages such as Japanese whose characters cannot be represented by a single byte.
Wide Character Support	Multibyte characters can be normalized by OS/390 C library functions and encoded in units of one length. These normalized characters are called wide characters. Conversions between multibyte and wide characters can be performed by string conversion functions such as <code>wcstombs()</code> , <code>mbstowcs()</code> , <code>wcsrtombs()</code> , and <code>mbsrtowcs()</code> , as well as the family of wide-character I/O functions. Wide-character data can be represented by the <code>wchar_t</code> data type.
Extended Precision Floating-Point Numbers	<p>OS/390 C/C++ provides three S/370 floating-point number data types: single precision (32 bits), declared as <code>float</code>; double precision (64 bits), declared as <code>double</code>; and extended precision (128 bits), declared as <code>long double</code>.</p> <p>Extended precision floating-point numbers give greater accuracy to mathematical calculations.</p> <p>As of Release 6, OS/390 C/C++ also supports IEEE 754 floating-point representation. By default, <code>float</code>, <code>double</code>, and <code>long double</code> values are represented in IBM S/390 floating point format. However, the IEEE 754 floating-point representation is used if you specify the <code>FL0AT(IEEE754)</code> compile option. For details on this support, see the description of the <code>FL0AT</code> option in the <i>OS/390 C/C++ User's Guide</i>.</p>
Command Line Redirection	You can redirect the standard streams <code>stdin</code> , <code>stderr</code> , and <code>stdout</code> from the command line or when calling programs using the <code>system()</code> function.
National Language Support	OS/390 C/C++ provides message text in either American English or Japanese. You can dynamically switch between the two languages.
Locale Definition Support	OS/390 C/C++ provides a locale definition utility that supports the creation of separate files of internationalization data, or locales. Locales can be used at run time to customize the behavior of an application to national language, culture, and coded character set (code page) requirements. Locale-sensitive library functions, such as <code>isdigit()</code> , use this information.
Coded Character Set (Code page) Support	The OS/390 C/C++ compiler can compile C/C++ source written in different EBCDIC code pages. In addition, the <code>iconv</code> utility converts data or source from one code page to another.

Feature	Description
Selected Built-in Library Functions	Selected library functions, such as string and character functions, are built into the compiler to improve performance execution. Built-in functions are compiled into the executable, and no calls to the library are generated.
Multitasking Facility (MTF)	Multitasking is a mode of operation where your program performs two or more tasks at the same time. OS/390 C provides a set of library functions that perform multitasking. These functions are known as the Multitasking Facility (MTF). MTF uses the multitasking capabilities of OS/390 to allow a single OS/390 C application program to use more than one processor of a multiprocessing system simultaneously.
Packed Structures and Unions	OS/390 C provides support for packed structures and unions. Structures and unions may be packed to reduce the storage requirements of a OS/390 C program.
Fixed-point (Packed) Decimal Data	<p>OS/390 C supports fixed-point (packed) decimal as a native data type for use in business applications. The packed data type is similar to the COBOL data type COMP-3 or the PL/I data type FIXED DEC, with up to 31 digits of precision.</p> <p>The Application Support Class Library provides the Binary Coded Decimal Class for C++ programs.</p>
Long Name Support	For portability, external names can be mixed case and up to 1024 characters in length. For C++, the limit applies to the mangled version of the name.
System Calls	You can call commands or executable modules using the <code>system()</code> function under OS/390, OS/390 UNIX, and TSO. You can also use the <code>system()</code> function to call EXECs on OS/390 and TSO, or Shell scripts using OS/390 UNIX.
Exploitation of ESA	Support for OS/390, IMS/ESA, Hiperspace expanded storage, and CICS/ESA allows you to exploit the features of the ESA.
Exploitation of hardware	<p>Use the ARCHITECTURE compiler option to select the minimum level of machine architecture on which your program will run. ARCH(3) enables support for IEEE 754 Binary Floating-Point instructions. ARCH(2) instructs the compiler to generate faster instruction sequences available only on newer machines.</p> <p>Use the TUNE compiler option to optimize your application for a selected machine architecture. Tune(3) optimizes your application for the new G5 processor. TUNE(2) optimizes your application for other architectures. For information on which machines and architectures support the above options, refer to the ARCHITECTURE and TUNE compiler information in the <i>OS/390 C/C++ User's Guide</i>.</p>



---

## Volume 1 - Part 2. Header Files

This part describes each header file, explains its contents, and lists the functions that use the file. The function descriptions are described in “Volume 1 - Part 3. Library Functions” on page 61.

The header files provided with the OS/390 C/C++ Library contain macro and constant definitions, type definitions, and function declarations. Some functions require definitions and declarations from header files to work correctly. The inclusion of header files is optional, as long as the necessary statements from the header files are coded directly into the source.

Use the `#include` directive to select header files to include with your application, for example, `#include <stdio.h>`.

For information about the `#include` directive, see *OS/390 C/C++ Language Reference* and *OS/390 C/C++ User's Guide*.

---

### Feature Test Macros

Many of the symbols that are defined in headers are “protected” by a feature test macro. These “protected” symbols are invisible to the application unless the user defines the feature test macro with `#define`, using either of the following methods:

- In the source code before including any header files.
- On the compilation command.

Note that the `LANGLVL` compiler option does not define or undefine these macros.

Table 3 summarizes the relationships between the feature test macros and the standards. ‘Yes’ indicates that a feature test macro makes visible the symbols related to a standard.

Feature test macros that do not apply to POSIX standards are not listed in this table.

Table 3. Feature Test Macros and Standards

Feature Test Macro	POSIX .1	POSIX .1a	POSIX .2	POSIX .4a	XPG4 .2	XPG4 .2 Ext
_POSIX_SOURCE	Yes					
_POSIX1_SOURCE 1	Yes					
_POSIX1_SOURCE 2	Yes	Yes				
_POSIX_C_SOURCE 1	Yes					
_POSIX_C_SOURCE 2	Yes		Yes			
_XOPEN_SOURCE	Yes		Yes		Yes	
_XOPEN_SOURCE _EXTENDED 1	Yes		Yes		Yes	Yes
_OPEN_SYS	Yes	Yes	Yes	Yes		
_OPEN_SYS_IPC_EXTENSIONS	Yes		Yes		Yes	
_OPEN_SYS_PTY_EXTENSIONS	Yes	Yes	Yes		Yes	Yes
_OPEN_THREADS	Yes	Yes		Yes		
_OPEN_SOURCE 1	Yes	Yes	Yes	Yes		
_OPEN_SOURCE 2 or _ALL_SOURCE	Yes	Yes	Yes	Yes	Yes	Yes
_OE_SOCKETS	Yes	Yes				
_OPEN_SYS SOCK_EXT	Yes	Yes		Yes	Yes	
_ALL_SOURCE_NO_THREADS	Yes	Yes	Yes		Yes	Yes
_OPEN_SOURCE 3	Yes	Yes	Yes	Yes	Yes	Yes

The following feature test macros are supported:

- `_POSIX_SOURCE`

When defined to any value with `#define`, it indicates that symbols required by POSIX.1 are made visible. Additional symbols can be made visible if POSIX.1 explicitly allows the symbol to appear in the header in question.

- `_POSIX1_SOURCE`

- When defined to 1, it has the same meaning as `_POSIX_SOURCE`.
- When defined to 2, both the POSIX.1a symbols and the POSIX.1 symbols are made visible. Additional symbols can be made visible if POSIX.1a explicitly allows the symbol to appear in the header in question.

- `_POSIX_C_SOURCE`

- When defined to 1, it indicates that symbols required by POSIX.1 are made visible. Additional symbols can be made visible if POSIX.1 explicitly allows the symbol to appear in the header in question.
- When defined to 2, both the POSIX.1 and POSIX.2 symbols are made visible. Additional symbols can be made visible if POSIX.2 explicitly allows the symbol to appear in the header in question.

- `_OPEN_SYS`

When defined to 1, this indicates that symbols required by POSIX.1, POSIX.1a, POSIX.2 are made visible. Any symbols defined by the `_OPEN_THREADS` macro are also made visible. Additional symbols can be made visible if any of these standards explicitly allows the symbol to appear in the header in question or if the symbol is defined to be an OS/390 UNIX services extension.

- `_OPEN_THREADS`



When defined to 1, this indicates that symbols required by POSIX.1, POSIX.1a, and POSIX.4a are made visible.

- `_XOPEN_SOURCE`

Defines the functionality defined in the XPG/4 standard dated July 1992.

- `_XOPEN_SOURCE_EXTENDED`

When defined to 1, this defines the functionality defined in the XPG/4 standard plus the set of “Common APIs for UNIX-based Operating Systems”, April, 1994, draft.

- `_XOPEN_SOURCE 500`

Makes available certain key functions that are associated with UNIX 98. Use of this feature test macro allows users to access UNIX 98 functions and structures as they are made available in OS/390 UNIX System Services.

- `_OPEN_SYS_IPC_EXTENSIONS`

Defines OS/390 UNIX services extensions to the X/Open InterProcess Communications functions. When `_OPEN_SYS_IPC_EXTENSIONS` is defined, the POSIX.1, POSIX.1a, and the XPG4 symbols are visible. This macro should be used in conjunction with `_XOPEN_SOURCE`.

- `_OPEN_SYS_PTY_EXTENSIONS`

Defines OS/390 UNIX services extensions to the X/Open Pseudo TTY functions. When `_OPEN_SYS_PTY_EXTENSIONS` is defined, the POSIX.1, POSIX.1a, XPG4, and XPG4.2 symbols are visible. This macro should be used in conjunction with `_XOPEN_SOURCE_EXTENDED 1`.

- `_OPEN_SOURCE`

When defined to 1, this defines all of the functionality that was available on OS/390 UNIX services in MVS 5.1. This macro is equivalent to specifying `_OPEN_SYS`.

When defined to 2, this defines all of the functionality that is available on OS/390 UNIX services in MVS 5.2.2, including XPG4, XPG4.2, and all of the OS/390 UNIX extensions.

- `_ALL_SOURCE`

Defines all of the functionality that is available with OS/390 UNIX services, including XPG4, XPG4.2, and all of the OS/390 UNIX MVS extensions. In addition, defining `_ALL_SOURCE` makes visible a number of symbols which are not permitted under ANSI, POSIX or XPG4, but which are provided as an aid to porting C-language applications to OS/390 UNIX services.

- `_OPEN_DEFAULT`

When defined to 0, and if no other feature test macro is defined, then all symbols will be visible. If in addition to `_OPEN_DEFAULT` only POSIX and/or XPG4 feature test macros are defined, then only the symbols so requested will be visible. Otherwise, additional symbols (e.g., those visible when the `LNLVL(EXTENDED)` compiler options specified), may be exposed.

When defined to 1, this provides the base level of OS/390 UNIX services functionality, which is POSIX.1, POSIX.1a and POSIX.2.

- `_OE_SOCKETS`

Defines a BSD-like socket interface for the function prototypes and structures involved. This can be used with `_XOPEN_SOURCE_EXTENDED 1` and the XPG4.2 socket interfaces will be replaced with the BSD-like interfaces.

- `_OPEN_MSGQ_EXT`

Defines an interface which enables use of `select()`, `selectex()` and `poll()` to monitor message and file descriptors.

- `_MSE_PROTOS`

The `_MSE_PROTOS` feature test macro does the following:

1. Selects behavior for a multibyte extension support (MSE) function declared in `wchar.h` as specified by ISO/IEC 9899:1990/Amendment 1:1994 instead of behavior for the function as defined by CAE Specification, System Interfaces and Headers, Issue 4, July 1992 (XPG4), and
2. Exposes declaration of an MSE function declared in `wchar.h` which is specified by ISO/IEC 9899:1990/Amendment 1:1994 but not by XPG4.

- `_ALL_SOURCE_NO_THREADS`

Provides the same function as `_ALL_SOURCE`, except it does not expose threading services (`_OPEN_THREADS`).

- `_VARARG_EXT_`

Allows users of the `va_arg`, `va_end`, and `va_start` macros to define the `va_list` type differently.

- `_OPEN_SYS_DIR_EXT`

Defines the interface and function prototypes for `__opendir2()` and `__readdir2()`.

- `_OPEN_SYS_SOCK_EXT`

Defines the interface for function prototypes and structures for the extended sockets and bulk mode support.

- `_OPEN_SYS_SOCK_EXT2`

Defines the interface and function prototype for `__accept_and_recv()`. sockets and bulk mode support.

- `_SHARE_EXT_VARS`

Provides access to POSIX and XPG4 external variables of an application from a dynamically loaded module such as a DLL.

- `_SHR_ENVIRON`

If you have declared `char **environ` in your program and want to access the environment variable array from a dynamically loaded module such as a DLL, define the `_SHR_ENVIRON` feature test macro and include `stdlib.h` in the program source.

- `_SHR_TIMEZONE`

To share access to the `timezone` external variable from a dynamically loaded module such as a DLL, define the `_SHR_TIMEZONE` feature test macro and include `time.h` in your program source. To avoid namespace pollution when `_SHR_TIMEZONE` is defined, the `timezone` variable must be referred to as `_timezone`.

- `_SHR_TZNAME`

To share access to the `tzname` external variable from dynamically loaded module such as a DLL, define the `_SHR_TZNAME` feature test macro and include `time.h` in your program source.

- `_SHR_DAYLIGHT`

To share access to the `daylight` external variable from a dynamically loaded module such as a DLL, define the `_SHR_DAYLIGHT` feature test macro and include `time.h` in your program source.

- `_SHR__LOC1`

To share access to the `__loc1` external variable from a dynamically loaded module such as a DLL, define the `_SHR__LOC1` feature test macro and include `libgen.h` in your program source.

- `_SHR_SIGNGAM`

To share access to the `signgam` external variable from a dynamically loaded module such as a DLL, define the `_SHR_SIGNGAM` feature test macro and include `math.h` in your program source.

- `_SHR_H_ERRNO`

To share access to the `h_errno` external variable from a dynamically loaded module such as a DLL, define the `_SHR_H_ERRNO` feature test macro and include `netdb.h` in your program source.

- `_SHR_T_ERRNO`

To share access to the `t_errno` external variable from a dynamically loaded module such as a DLL, define the `_SHR_T_ERRNO` feature test macro and include `xti.h` in your program source.

- `_SHR_LOC1`

To share access to the `loc1` external variable from a dynamically loaded module such as a DLL, define the `_SHR_LOC1` feature test macro and include `regex.h` in your program source.

- `_SHR_LOC2`

To share access to the `loc2` external variable from a dynamically loaded module such as a DLL, define the `_SHR_LOC2` feature test macro and include `regex.h` in your program source.

- `_SHR_LOCS`

To share access to the `locs` external variable from a dynamically loaded module such as a DLL, define the `_SHR_LOCS` feature test macro and include `regex.h` in your program source.

- `_SHR_OPTARG`

To share access to the `optarg` external variable from a dynamically loaded module such as a DLL, define the `_SHR_OPTARG` feature test macro and include `unistd.h` or `stdio.h` in your program source.

- `_SHR_OPTIND`

To share access to the `optind` external variable from a dynamically loaded module such as a DLL, define the `_SHR_OPTIND` feature test macro and include `unistd.h` or `stdio.h` in your program source.

- `_SHR_OPTOPT`

To share access to the `optopt` external variable from a dynamically loaded module such as a DLL, define the `_SHR_OPTOPT` feature test macro and include `unistd.h` or `stdio.h` in your program source.

- `_SHR_OPTERR`

To share access to the `opterr` external variable from a dynamically loaded module such as a DLL, define the `_SHR_OPTERR` feature test macro and include `unistd.h` or `stdio.h` in your program source.

- `_LONGMAP`

Programs that need to compile with `_LONGMAP` defined:

- Compiled with the `LONGNAME` C/C++ compiler option
- **And** use POSIX functions
- **And** use the Prelinker, outside of the OE (shell) environment (i.e., do not use the OE Prelinker option as does `c89`).

Programs that do not need to compile with `_LONGMAP` defined:

- Compiled with the `LONGNAME` C/C++ compiler option
- **And** use POSIX functions
- **And** use the Binder support for C/C++ compiled programs. This normally means using `SCEELKEX` with the Binder.

- `__LIBASCII`

Provides an ASCII-like environment for the following C/C++ functions:

```
access(), argvtoascii(), argvtoebcdic(), asctime(), atof(),
atoi(), atol(), chdir(), chmod(), chown(), creat(), ctime(),
dllload(), dllqueryfn(), dynalloc()
ecvt(), execv(), execve(), execvp(),
fcvt(), fdopen(), fopen(),
freopen(), ftok(), gcvt(),
getcwd(), getenv(), getgrnam(), gethostbyaddr(),
gethostbyname(), gethostname(),
getlogin(), getopt(), getpass(), getpwnam(), getpwuid(),
getservbyname(), getwd(), inetaddr(), inet_ntoa(),
isalnum(), isalpha(), iscntrl(),
isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(),
isupper(), isxdigit(), link(), localeconv(), mbstowcs(), mbtowc(),
mkdir(), mknod(), mktemp(), nl_langinfo(), open(), opendir(),
perror(), popen(), ptsname(), putenv(),
readdir(), regcomp(), remove(), rename(), rexec(), rmdir(), scanf(),
setenv(), setkey(), setlocale(), setvbuf(), sprintf(), sscanf(),
stat(), statvfs(), strcasecmp(), strerror(), strncasecmp(),
strtod(), strtol(), strtoul(), system(), tempnam(), tmpnam(),
toascii(), toebcdic(), tolower(), toupper(), uname(), unlink(), utime(),
utimes()
```

For each application program using one or more of these functions, where the input/output is ASCII add the following feature test macro:

- `#define __LIBASCII`
- Recompile using the `-D__STRING_CODE_SET__="ISO8859-1"` option to cause the compiler to generate all strings defined in the source program in ASCII rather than EBCDIC format.

**Note:** The libascii functions are as thread-safe as the run-time library with the exception, of the getopt() function. The libascii getopt() function is not thread-safe. The second argument is changed for a short period of time from EBCDIC to ASCII and then back to EBCDIC.

•

---

## aio.h

The aio.h header file contains definitions for asynchronous I/O operations. It declares these functions:

aio_read()	aio_write()	aio_cancel()
aio_suspend()	aio_error()	aio_return()

---

## arpa/inet.h

The arpa/inet.h header file contains definitions for internet operations.

---

## assert.h

The assert.h header file defines the assert() macro that allows you to insert diagnostics into your code. You must include assert.h when you use assert().

---

## cics.h

The cics.h header file declares the iscics() function, which verifies whether CICS is running.

---

## collate.h

The collate.h header includes declarations of functions that allow retrieval of information regarding the current locale's collating properties. It declares these functions:

cclass()	collequiv()	collorder()	collrange()	colltostr()
getmccoll()	getwmccoll()	ismccollet()	maxcoll()	strtcoll()

For more information about the effect of locale, see setlocale(), locale.h, or look up the individual functions in this book. For still more information, see the chapter, "Internationalization: Locales and Character Sets" in the *OS/390 C/C++ Programming Guide*.

---

## cpio.h

The cpio.h header file contains CPIO archive values.

---

### csp.h

The csp.h header file declares the `__csplist` macro, which obtains the CSP parameter list.

These macros are *not* supported under OS/390 UNIX services and they are not supported for C++ applications.

---

### ctest.h

The ctest.h header file contains declarations for the functions that involve debugging and diagnostics. The diagnostic functions are:

<code>cdump()</code>	<code>csnap()</code>	<code>ctest()</code>	<code>ctrace()</code>
----------------------	----------------------	----------------------	-----------------------

---

### ctype.h

The ctype.h header file declares functions used in character classification. The functions declared are:

<code>isalnum()</code>	<code>isalpha()</code>	<code>isblank()</code>	<code>iscntrl()</code>	<code>isdigit()</code>
<code>isgraph()</code>	<code>islower()</code>	<code>isprint()</code>	<code>ispunct()</code>	<code>isspace()</code>
<code>isupper()</code>	<code>isxdigit()</code>	<code>_tolower()</code>	<code>_toupper()</code>	

`_XOPEN_SOURCE`

<code>isascii()</code>	<code>toascii()</code>	<code>tolower()</code>	<code>toupper()</code>
------------------------	------------------------	------------------------	------------------------

For more information about the effect of locale, see `setlocale()`, `locale.h`, or look up the individual functions in this book. For still more information, see the chapter, “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*.

---

### decimal.h

The decimal.h header file is not supported under OS/390 C++ applications.

The decimal.h header file contains declarations for those built-in functions that perform fixed-point decimal operations. The functions declared are:

<code>decabs()</code>	<code>decchk()</code>	<code>decfix()</code>
-----------------------	-----------------------	-----------------------

The header file also contains definitions of constants that specify the ranges of the decimal data types.

---

### dirent.h

The dirent.h header file contains constants, prototypes, and typedef definitions for POSIX directory access functions. It declares the following functions.

`_POSIX_SOURCE`

<code>closedir()</code>	<code>opendir()</code>	<code>__opendir2()</code>	<code>readdir()</code>	<code>__readdir2()</code>	<code>rewinddir()</code>
-------------------------	------------------------	---------------------------	------------------------	---------------------------	--------------------------

`_XOPEN_SOURCE`

`seekdir()`                      `telldir()`

This header file can be used by C++ POSIX(OFF) functions. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information.

## dll.h

The `dll.h` header file declares the following functions:

`dllload()`                      `dllqueryvar()`                      `dllqueryfn()`                      `dllfree()`

Use this header file when using these functions to import functions and variables from a DLL.

## dynit.h

The `dynit.h` header file contains information for dynamic allocation routines. Specifically, it contains declarations of the `dynalloc()` and `dynfree()` functions, the definition of the `dyninit()` macro, declarations of related structures, and definitions of related constants.

## env.h

The `env.h` header file is used to declare the `setenv()` and `clearenv()` functions, which are used in POSIX programs to set and clear environment variables. The `env.h` header file requires the `_POSIX1_SOURCE 2` feature test macro.

## errno.h

The `errno.h` header file defines the symbolic constants that are returned in the external variable `errno`.

*Table 4 (Page 1 of 3). Definitions in `errno.h`*

<code>EACCES</code>	Permission denied
<code>EADDRINUSE</code>	Address in use
<code>EADDRNOTAVAIL</code>	Address not available
<code>EADV</code>	advertise error
<code>EAFNOSUPPORT</code>	Address family not supported
<code>EAGAIN</code>	Resource temporarily unavailable
<code>EALREADY</code>	Connection already in progress
<code>EBADF</code>	Bad file descriptor
<code>EBADMSG</code>	Bad message
<code>EBUSY</code>	Resource busy
<code>ECHILD</code>	No child processes
<code>ECOMM</code>	Communication error on send
<code>ECONNABORTED</code>	Connection aborted
<code>ECONNREFUSED</code>	Connection refused
<code>ECONNRESET</code>	Connection reset
<code>EDEADLK</code>	Resource deadlock avoided
<code>EDESTADDRREQ</code>	Destination address required
<code>EDOM</code>	Domain error
<code>EDOTDOT</code>	Cross mount point (not an error)

Table 4 (Page 2 of 3). Definitions in *errno.h*


---

EDQUOT	Reserved
EEXIST	File exists
EFAULT	Bad address
EFBIG	File too large
EHOSTDOWN	Host is down
EHOSTUNREACH	Destination host can not be reached
EIMBADCALL	A bad socket-call constant in IUCV header
EIMBADPARM	Other IUCV header error
EIMCANCELLED	Request cancelled
EIBMCONFLICT	Conflicting call outstanding on socket
EIBMIUCVERR	Request failed due to IUCV error
EIBMSOCKINUSE	Assigned socket number already in use
EIBMSOCKOUTOFRANGE	Assigned socket number out of range
EIDRM	Identifier removed
EILSEQ	Illegal byte sequence
EINPROGRESS	Connection in progress
EINTR	Interrupted function call
EINTRNODATA	Function call interrupted before any data received
EINVAL	Invalid argument
EIO	Input/output error
EISCON	Socket is already connected
EISDIR	Is a directory
ELEMSGERR	Message file was not found in the hierarchical file system
ELENOFORK	Language Environment member language cannot tolerate a fork()
ELOOP	A loop exists in symbolic links encountered during resolution of the path argument
EMFILE	Too many open files
EMLINK	Too many links
EMSGSIZE	Message too long
EMULTIHOP	Multihop is not allowed
EMVSBADCHAR	Bad character in environment variable name
EMVSCATLG	Catalog obtain error
EMVSCVAF	Catalog Volume Access Facility error
EMVSDYNALC	Dynamic allocation error
EMVSERR	An MVS internal error
EMVSEXPIRE	Password has expired
EMVSINITIAL	Process initialization err
EMVSNORTL	Access to the OS/390 UNIX services version of the C RTL is denied
EMVSNOTUP	OS/390 UNIX services are not active
EMVSPARM	Bad parameters were passed to the service
EMVSPASSWORD	Password is invalid
EMVSPATHOPTS	Access mode argument conflicts with PATHOPTS parameter
EMVSPFSFILE	PDSE/X encountered a permanent file error
EMVSPFSPERM	PDSE/X encountered a system error
EMVSSAF2ERR	SAF/RACF error
EMVSSAFEXTRERR	SAF/RACF extract error
EMVSTODNOTSET	System TOD clock not set
ENAMETOOLONG	File name too long
ENETDOWN	The local interface to use or reach the destination
ENETRESET	Network dropped connection on reset
ENETUNREACH	Network unreachable
ENFILE	Too many open files in system
ENOBUFS	No buffer space available
ENODATA	No message available



Table 4 (Page 3 of 3). Definitions in *errno.h*


---

ENODEV	No such device
ENOENT	No such file or directory
ENOEXEC	Exec format error
ENOLINK	The link has been severed
ENOLCK	No locks available
ENOMEM	Not enough space
ENOMSG	No message of desired type
ENONET	Machine is not on the network
ENOPROTOPT	Protocol not available
ENOREUSE	The socket cannot be reused
ENOSPC	No space left on device
ENOSR	No stream resource
ENOSYS	Function not implemented
ENOSTR	Not a stream
ENOTBLK	Block device required
ENOTDIR	Not a directory
ENOTEMPTY	Directory not empty
ENOTSOCK	Descriptor does not refer to a socket
ENOTTY	Inappropriate I/O control operation
ENXIO	No such device or address
EOFFLOADboxDOWN	Offload box down
EOFFLOADboxERROR	Offload box error
EOFFLOADboxRESTART	Offload box restarted
EOPNOTSUPP	Operation not supported on socket
EOVERFLOW	Value too large to be stored in data type
EPERM	Operation not permitted
EPFNOSUPPORT	Protocol family not supported
EPIPE	Broken pipe
EPROCLIM	Too many processes
EPROTO	Protocol error
EPROTONOSUPPORT	Protocol not supported
EPROTOTYPE	The socket type is not supported by the protocol
ERANGE	Range error
EREMCHG	Remote address changed
EREMOTE	Too many levels of remote in path
EROFS	Read-only file system
ERREMOTE	Object is remote
ESHUTDOWN	Cannot send after socket shutdown
ESOCKTNOSUPPORT	Socket type not supported
ESPIPE	Invalid seek
ESRCH	No such process
ESRMNT	srmount error
ESTALE	The file handle is stale
ETIME	Stream ioctl() timeout
ETIMEDOUT	Socket not connected
ETOOMANYREFS	Too many references: cannot splice
ETXTBSY	Text file busy
EUSERS	Too many users
EWouldBLOCK	Problem on non-blocking socket
EXDEV	A link to a file on another file system was attempted
E2BIG	Argument list too long

---

The *errno.h* header file also defines *errno*, which is a modifiable `1value` having type `int`. If you intend to test the value of *errno* after library function calls, first set it to 0, because the library functions do not reset the value to 0.

strerror() or perror() functions can be used to print the description of the message associated with a particular errno.

To test for the explicit error, use the macro names defined in errno.h, rather than specific values of these macros. Doing so will ensure future compatibility and portability.

errno.h also declares the \_\_errno2() prototype.

---

### ezbzsdc.h

The ezbzsdc.h header file contains structures for getting sysplex fully qualified domain name.

---

### fcntl.h

The fcntl.h header file declares the following POSIX functions for creating, opening, rewriting, and manipulating files.

\_POSIX\_SOURCE

creat()                  fcntl()                  open()                  \_\_open\_stat()

---

### features.h

The features.h header file contains definitions for feature test macros. For information on feature test macros, see “Feature Test Macros” on page 17.

---

### float.h

The float.h header file contains definitions of constants listed in ANSI 2.2.4.2.2, that describe the characteristics of the internal representations of the three floating-point data types, float, double, and long double. The definitions are:

---

*Table 5 (Page 1 of 2). Definitions in float.h*

Constant	Description
FLT_ROUNDS	The rounding mode for floating-point addition.
FLT_RADIX	The radix for OS/390 C applications, FLT_RADIX, is defined to be 16.
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	The number of hexadecimal digits stored to represent the significant of a fraction.
FLT_DIG DBL_DIG LDBL_DIG	The number of decimal digits, $q$ , such that any floating-point number with $q$ decimal digits can be rounded into a floating-point number with $p$ radix FLT_RADIX digits and back again, without any change to the $q$ decimal digits.
FLT_MIN_EXP DBL_MIN_EXP LDBL_MIN_EXP	The minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized floating-point number.

---

Table 5 (Page 2 of 2). Definitions in float.h

Constant	Description
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	The minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers.
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	The maximum integer such that FLT_RADIX raised to that power minus 1 is a representable finite floating-point number.
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	The maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers.
FLT_MAX DBL_MAX LDBL_MAX	The maximum representable finite floating-point number.
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	The difference between 1.0 and the least value greater than 1.0 that is representable in the given floating-point type.
FLT_MIN DBL_MIN LDBL_MIN	The minimum normalized positive floating-point number.

---

## fmtmsg.h

The fmtmsg.h header file contains message display structures.

---

## fnmatch.h

The fnmatch.h header file contains filename matching types.

---

## \_\_ftp.h

The \_\_ftp.h header file contains definitions for FTP resolver functions.

---

## ftw.h

The ftw.h header file contains file tree traversal constants.

---

## glob.h

The glob.h header file contains pathname pattern matching types.

---

## grp.h

The grp.h header file declares functions used to access group databases.

\_POSIX\_SOURCE

getgrgid()            getgrnam()

\_OPEN\_SYS

initgroups()          setgroups()

---

`_XOPEN_SOURCE_EXTENDED` 1

`setgrent()`                      `endgrent()`                      `getgrent()`

---

## iconv.h

The `iconv.h` header file declares the `iconv_open()`, `iconv()`, and `iconv_close()` functions that deal with code conversion.

---

## ims.h

The `ims.h` header file declares the `ctdli()` function that invokes IMS facilities. The function is *not* supported from an OS/390 UNIX services program running POSIX(ON).

---

## langinfo.h

The `langinfo.h` header file contains the declaration for the `nl_langinfo()` function. The header file also defines the macros that, in turn, define constants used to identify the information queried in the current locale. The following macros are defined:

*Table 6 (Page 1 of 2). Item values defined in langinfo.h*

Item Name	Description
<code>ABDAY_1</code>	Abbreviated first day of the week
<code>ABDAY_2</code>	Abbreviated second day of the week
<code>ABDAY_3</code>	Abbreviated third day of the week
<code>ABDAY_4</code>	Abbreviated fourth day of the week
<code>ABDAY_5</code>	Abbreviated fifth day of the week
<code>ABDAY_6</code>	Abbreviated sixth day of the week
<code>ABDAY_7</code>	Abbreviated seventh day of the week
<code>ABMON_1</code>	Abbreviated first month
<code>ABMON_2</code>	Abbreviated second month
<code>ABMON_3</code>	Abbreviated third month
<code>ABMON_4</code>	Abbreviated fourth month
<code>ABMON_5</code>	Abbreviated fifth month
<code>ABMON_6</code>	Abbreviated sixth month
<code>ABMON_7</code>	Abbreviated seventh month
<code>ABMON_8</code>	Abbreviated eighth month
<code>ABMON_9</code>	Abbreviated ninth month
<code>ABMON_10</code>	Abbreviated tenth month
<code>ABMON_11</code>	Abbreviated eleventh month
<code>ABMON_12</code>	Abbreviated twelfth month
<code>ALT_DIGITS</code>	String of semicolon separated alternative symbols for digits
<code>AM_STR</code>	String for morning
<code>CODESET</code>	Current encoded character set of the process
<code>CREDIT_SIG</code>	Credit sign
<code>CRNCYSTR</code>	Currency symbol
<code>D_FMT</code>	String for formatting date
<code>D_T_FMT</code>	String for formatting date and time
<code>DAY_1</code>	Name of the first day of the week
<code>DAY_2</code>	Name of the second day of the week
<code>DAY_3</code>	Name of the third day of the week
<code>DAY_4</code>	Name of the fourth day of the week
<code>DAY_5</code>	Name of the fifth day of the week
<code>DAY_6</code>	Name of the sixth day of the week

Table 6 (Page 2 of 2). Item values defined in *langinfo.h*

Item Name	Description
DAY_7	Name of the seventh day of the week
DEBIT_SIGN	Debit sign
ERA	String of semicolon separated era segments
ERA_D_FMT	String for era date format
ERA_D_T_FMT	String for era date and time format
ERA_T_FMT	String for era time format
MON_1	Name of the first month
MON_2	Name of the second month
MON_3	Name of the third month
MON_4	Name of the fourth month
MON_5	Name of the fifth month
MON_6	Name of the sixth month
MON_7	Name of the seventh month
MON_8	Name of the eighth month
MON_9	Name of the ninth month
MON_10	Name of the tenth month
MON_11	Name of the eleventh month
MON_12	Name of the twelfth month
NOEXPR	Negative response expression
NOSTR	Negative response string
PM_STR	String for afternoon
RADIXCHAR	Radix character
T_FMT	String for formatting time
T_FMT_AMPM	String for formatting time in 12-hour clock format
THOUSEP	Separator for thousands
YESEXPR	Affirmative response expression
YESSTR	affirmative response string

The *langinfo.h* header file also defines the type `n_l_item`, which is the type of the above constants.

For more information about the effect of locale, see `setlocale()`, *locale.h*, or look up the individual functions in this book. For still more information, see the chapter, “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*.

---

## lc\_core.h

The *lc\_core.h* header file contains locale-related data structures.

---

## leawi.h

The *leawi.h* header file contains internal macros. The header file is required for applications that use Language Environment Application Writer Interfaces (LE AWI).

---

## libgen.h

The *libgen.h* header file contains definitions for pattern matching functions.

## limits.h

The limits.h header file contains symbolic names that represent standard values for limits on resources, such as the maximum value for an object of type char.

*Table 7 (Page 1 of 2). Definitions of Resource Limits*

BC_BASE_MAX	32767
BC_DIM_MAX	32768
BC_SCALE_MAX	32767
BC_STRING_MAX	2048
CHAR_BIT	8
CHAR_MAX	127 (_CHAR_SIGNED)
CHAR_MAX	255
CHAR_MIN	(-128) (_CHAR_SIGNED)
CHAR_MIN	0
COLL_WEIGHTS_MAX	2
__DIR_NAME_MAX	256
EXPR_NEST_MAX	32
INT_MAX	2147483647
INT_MIN	(-2147483647 - 1)
LINE_MAX	2048
LONG_MAX	2147483647
LONGLONG_MAX	(9223372036854775807LL)
LONG_MIN	(-2147483647L - 1)
LONGLONG_MIN	(-LONGLONG_MAX - 1)
MB_LEN_MAX	4
NGROUPS_MAX	300
_POSIX_ARG_MAX	4096
_POSIX_CHILD_MAX	6
_POSIX_DATAKEYS_MAX	32
_POSIX_LINK_MAX	8
_POSIX_MAX_CANON	255
_POSIX_MAX_INPUT	255
_POSIX_NAME_MAX	14
_POSIX_NGROUPS_MAX	0
_POSIX_OPEN_MAX	16
_POSIX_PATH_MAX	255
_POSIX_PIPE_BUF	512
_POSIX_SSIZE_MAX	32767
_POSIX_STREAM_MAX	8
POSIX_SYMLINK_MAX	24
_POSIX_TZNAME_MAX	3
_POSIX2_BC_BASE_MAX	99
_POSIX2_BC_DIM_MAX	2048
_POSIX2_BC_SCALE_MAX	99
_POSIX2_BC_STRING_MAX	1000
_POSIX2_COLL_WEIGHTS_MAX	2
_POSIX2_EXPR_NEST_MAX	32
_POSIX2_LINE_MAX	2048
_POSIX2_RE_DUP_MAX	255
RE_DUP_MAX	255
SCHAR_MAX	127
SCHAR_MIN	(-128)
SHRT_MAX	32767
SHRT_MIN	(-32768)
SSIZE_MAX	2147483647
UCHAR_MAX	255
UINT_MAX	4294967295
ULONG_MAX	4294967295U

Table 7 (Page 2 of 2). Definitions of Resource Limits

ULONGLONG_MAX	(18446744073709551615ULL)
USHRT_MAX	65535

## localedef.h

The localedef.h header file defines data structures for locale objects which are loaded by setlocale(). The data structures in localedef.h are not a supported programming interface.

## locale.h

The locale.h header file contains declarations for the localdtconv(), localeconv(), and setlocale() library functions, which are useful for changing the C locale when you are creating applications for international users.

The locale.h file declares the lconv structure. Table 8 below shows the elements of the lconv structure and the defaults for the C locale.

Table 8 (Page 1 of 2). Elements of the lconv Structure

Element	Purpose of Element	Default
char *decimal_point	Decimal-point character used to format nonmonetary quantities.	"."
char *thousands_sep	Character used to separate groups of digits to the left of the decimal-point character in formatted nonmonetary quantities.	""
char *grouping	String indicating the size of each group of digits in formatted nonmonetary quantities. The value of each character in the string determines the number of digits in a group. A value of CHAR_MAX indicates that there are no further groupings. 0 indicates that the previous element is to be used for the remainder of the digits.	""
char *int_curr_symbol	International currency symbol for the current locale. The first three characters contain the alphabetic international currency symbol. The fourth character (usually a space) is the character used to separate the international currency symbol from the monetary quantity.	""
char *currency_symbol	Local currency symbol of the current locale.	""
char *mon_decimal_point	Decimal-point character used to format monetary quantities.	"."
char *mon_thousands_sep	Separator for digits in formatted monetary quantities.	""

Table 8 (Page 2 of 2). Elements of the *Iconv* Structure

Element	Purpose of Element	Default
char *mon_grouping	String indicating the size of each group of digits in formatted monetary quantities. The value of each character in the string determines the number of digits in a group. A value of CHAR_MAX indicates that there are no further groupings. 0 indicates that the previous element is to be used for the remainder of the digits.	""
char *positive_sign	String indicating the positive sign used in monetary quantities.	""
char *negative_sign	String indicating the negative sign used in monetary quantities.	""
char int_frac_digits	The number of displayed digits to the right of the decimal place for internationally formatted monetary quantities.	UCHAR_MAX
char frac_digits	Number of digits to the right of the decimal place in monetary quantities.	UCHAR_MAX
char p_cs_precedes	1 if the currency_symbol precedes the value for a nonnegative formatted monetary quantity; 0 if it does not.	UCHAR_MAX
char p_sep_by_space	1 if the currency_symbol is separated by a space from the value of a nonnegative formatted monetary quantity; 0 if it does not; 2 if a space separates the symbol and the sign string—if adjacent.	UCHAR_MAX
char n_cs_precedes	1 if the currency_symbol precedes the value for a negative formatted monetary quantity; 0 if it does not.	UCHAR_MAX
char n_sep_by_space	1 if the currency_symbol is separated by a space from the value of a negative formatted monetary quantity; 0 if it does not; 2 if a space separates the symbol and the sign string—if adjacent.	UCHAR_MAX
char p_sign_posn	Value indicating the position of the positive_sign for a nonnegative formatted monetary quantity.	UCHAR_MAX
char n_sign_posn	Value indicating the position of the negative_sign for a negative formatted monetary quantity.	UCHAR_MAX
char *left_parenthesis	Negative-valued monetary symbol.	""
char *right_parenthesis	Negative-valued monetary symbol.	""
char *debit_sign	Debit_sign characters.	""
char *credit_sign	Credit_sign characters.	""

The `locale.h` file declares the `dtconv` structure:



```

struct dtconv {
    char *abbrev_month_names[12]; /* Abbreviated month names */
    char *month_names[12];       /* full month names */
    char *abbrev_day_names[7];   /* Abbreviated day names */
    char *day_names[7];          /* full day names */
    char *date_time_format;      /* date and time format */
    char *date_format;           /* date format */
    char *time_format;           /* time format */
    char *am_string;             /* AM string */
    char *pm_string;             /* PM string */
    char *time_format_ampm;      /* long date format */
};

```

The information in each element of `dtconv` is equivalent to calling `nl_langinfo()` with these keywords:

Keyword	Element of <code>dtconv</code>
<code>abmon</code>	<code>abbrev_month_names</code>
<code>mon</code>	<code>month_names</code>
<code>abday</code>	<code>abbrev_day_names</code>
<code>day</code>	<code>day_names</code>
<code>d_t_fmt</code>	<code>date_time_format</code>
<code>d_fmt</code>	<code>date_format</code>
<code>t_fmt</code>	<code>time_format</code>
<code>am_pm</code>	<code>am_string</code>
<code>am_pm</code>	<code>pm_string</code>
<code>t_fmt_ampm</code>	<code>time_format_ampm</code>

The `locale.h` file also contains additional definitions for the IBM OS/390 C/C++ compiler, including the `LC_SYNTAX` extension and the following macros:

<code>LC_ALL</code>	<code>LC_COLLATE</code>	<code>LC_CTYPE</code>	<code>LC_MONETARY</code>
<code>LC_NUMERIC</code>	<code>LC_TIME</code>	<code>LC_TOD</code>	<code>NULL</code>

The aspects of a program related to national language or to cultural characteristics (such as time zone, currency symbols, and sorting order of characters) can be customized at run time using different locales, to suit users' requirements at those locales. The methods for doing so are discussed in the internationalization chapter of *OS/390 C/C++ Programming Guide*.

---

## math.h

The `math.h` header file contains function declarations for all the floating-point math functions:

No feature test macro required.

acos()	asin()	atan()	atan2()	ceil()
cos()	cosh()	exp()	floor()	fmod()
frexp()	ldexp()	log()	log10()	modf()
pow()	sin()	sinh()	sqrt()	tan()
tanh()				

\_XOPEN\_SOURCE

erf()	erfc()	gamma()	hypot()	isnan()
j0()	j0()	j1()	lgamma()	yn()
y0()	y1()			

\_XOPEN\_SOURCE\_EXTENDED 1

acosh()	asinh()	atanh()	cbrt()	expm1()
ilogb()	logb()	log1p()	nextafter()	remainder()
rint()	scalb()			

The header file includes declarations for the built-in functions `abs()` and `fabs()`. For information about built-in functions, see “Built-in Functions” on page 64.

The `math.h` header file declares the macro `HUGE_VAL`, which expands to a positive `double` expression.

For all mathematical functions, a *domain error* occurs when an input argument is outside the range of values allowed for that function. If a domain error occurs, `errno` is set to the value of `EDOM`.

A range error occurs if the result of the function cannot be represented in a `double` value. If the magnitude of the result is too large (overflow), the function returns the positive or negative value of the macro `HUGE_VAL`, and sets `errno` to `ERANGE`. If the result is too small (underflow), the function returns 0.

---

## memory.h

The `memory.h` header file contains declarations for memory operations.

---

## monetary.h

The `monetary.h` header file contains the declaration for the `strfmon()` function.

For more information about the effect of locale, see `setlocale()`, `locale.h`, or look up the individual functions in this book. For still more information, see the chapter, “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*.

---

## msgcat.h

The `msgcat.h` header file contains message catalog structures and definitions. The data structures in `msgcat.h` are not a supported programming interface.

---

## mtf.h

The mtf.h header file contains declarations for the multitasking facility (MTF) functions:

tinit()                      tsched()                      tsyncro()                      tterm()

tsched() is a built-in function.

This header file also contains definitions of macros for certain return values from the above functions.

This header file is supported only under OS/390 C applications. The functions are *not* supported under OS/390 UNIX services.

---

## ndbm.h

The ndbm.h header file contains definitions for ndbm database operations.

---

## netdb.h

The netdb.h header file contains definitions for network database operations.

---

## net/if.h

The net/if.h header file contains network interface structures and definitions.

---

## netinet/in.h

The inetnet/in.h header file contains definitions for the internet protocol family.

---

## net/rtroute.h

The net/rtroute.h header file contains network routing structures and definitions.

---

## new.h

The new.h header file declares the set\_new\_handler() function, which is used for OS/390 C++ exception handling (try, throw, and catch). This header file also declares array and non-array version of the allocation operator new and the deallocation operator delete.

---

## nlist.h

The nlist.h header file declares the nlist() function.

---

### nl\_langinfo.h

The nl\_langinfo.h header file also contains the declarations for all the macros which are declared in “langinfo.h” on page 30.

---

### nl\_types.h

The nl\_types.h header file contains the following definitions.

No feature test macro required.

nl\_langinfo()

\_XOPEN\_SOURCE

catclose()                  catgets()                  catopen()

For more information about the effect of locale, see setlocale(), locale.h, or look up the individual functions in this book. For still more information, see the chapter, “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*.

---

### poll.h

The poll.h header file contains definitions for the poll() function.

---

### pthread.h

The pthread.h header file defines the timespec structure. It also contains declarations for the following thread functions:

\_OPEN\_THREADS

pthread\_attr\_destroy()  
pthread\_attr\_getstacksize()  
pthread\_attr\_setdetachstate()  
pthread\_cancel()  
pthread\_cleanup\_push()  
pthread\_condattr\_init()  
pthread\_cond\_destroy()  
pthread\_cond\_signal()  
pthread\_cond\_wait()  
pthread\_detach()  
pthread\_exit()  
pthread\_join()  
pthread\_kill()  
pthread\_mutexattr\_getpshared()  
pthread\_mutexattr\_init()  
pthread\_mutexattr\_settype()  
pthread\_mutex\_init()  
pthread\_mutex\_trylock()  
pthread\_once()  
pthread\_rwlockattr\_getpshared()  
pthread\_rwlockattr\_setpshared()  
pthread\_rwlock\_init()

pthread\_attr\_getdetachstate()  
pthread\_attr\_init()  
pthread\_attr\_setstacksize()  
pthread\_cleanup\_pop()  
pthread\_condattr\_destroy()  
pthread\_cond\_broadcast()  
pthread\_cond\_init()  
pthread\_cond\_timedwait()  
pthread\_create()  
pthread\_equal()  
pthread\_getspecific()  
pthread\_key\_create()  
pthread\_mutexattr\_destroy()  
pthread\_mutexattr\_gettype()  
pthread\_mutexattr\_setpshared()  
pthread\_mutex\_destroy()  
pthread\_mutex\_lock()  
pthread\_mutex\_unlock()  
pthread\_rwlockattr\_destroy()  
pthread\_rwlockattr\_init()  
pthread\_rwlock\_destroy()  
pthread\_rwlock\_rdlock()

```
pthread_rwlock_tryrdlock()
pthread_rwlock_unlock()
pthread_setintr()
pthread_setspecific()
pthread_tag_np()
```

```
pthread_rwlock_trywrlock()
pthread_self()
pthread_setintrtype()
pthread_testintr()
pthread_yield()
```

## **\_OPEN\_SYS**

```
pthread_attr_getsynctype_np()
pthread_attr_setweight_np()
pthread_condattr_setkind_np()
pthread_mutexattr_getkind_np()
pthread_set_limit_np()
pthread_security_np()
```

```
pthread_attr_getweight_np()
pthread_condattr_getkind_np()
pthread_join_d4_np()
pthread_mutexattr_setkind_np()
pthread_attr_setsynctype_np()
```

Furthermore, pthread.h defines these symbols:

<code>__COND_DEFAULT</code>	<code>__COND_NODEBUG</code>
<code>__DETACHED</code>	<code>__HEAVY_WEIGHT</code>
<code>__MEDIUM_WEIGHT</code>	<code>__MUTEX_NODEBUG</code>
<code>__MUTEX_NONRECURSIVE</code>	<code>__MUTEX_RECURSIVE</code>
<code>__UNDETACHED</code>	<code>NO_PRIO_INHERIT</code>
<code>PRIO_INHERIT</code>	<code>PTHREAD_DEFAULT_SCHED</code>
<code>PTHREAD_INHERIT_SCHED</code>	<code>PTHREAD_INTR_ASYNCHRONOUS</code>
<code>PTHREAD_INTR_CONTROLLED</code>	<code>PTHREAD_INTR_DISABLE</code>
<code>PTHREAD_INTR_ENABLE</code>	<code>PTHREAD_ONCE_INIT</code>
<code>PTHREAD_SCOPE_GLOBAL</code>	<code>PTHREAD_SCOPE_LOCAL</code>
<code>SCHED_FIFO</code>	<code>SCHED_OTHER</code>
<code>SCHED_RR</code>	<code>PRIO_PROTECT</code>

Furthermore, pthread.h defines these macros:

<code>PTHREAD_MUTEX_DEFAULT</code>	<code>PTHREAD_MUTEX_ERRORCHECK</code>
<code>PTHREAD_MUTEX_NORMAL</code>	<code>PTHREAD_MUTEX_INITIALIZER</code>
<code>PTHREAD_MUTEX_RECURSIVE</code>	<code>PTHREAD_RWLOCK_INITIALIZER</code>

---

## **pwd.h**

The pwd.h header file declares functions that access the user database through a password structure. The header file also defines the passwd structure.

### **\_POSIX\_SOURCE**

```
getpwnam()
```

```
getpwuid()
```

### **\_XOPEN\_SOURCE\_EXTENDED 1**

```
endpwent()
```

```
getpwent()
```

```
setpwent()
```

---

## **re\_comp.h**

The re\_comp.h header file contains regular expression matching functions for re\_comp().

---

### regex.h

The regex.h header file contains definitions for the following regular expression functions.

regcomp()                      regerror()                      regexec()                      regfree()

The regex.h header file declares the regex\_t type, which can store a compiled regular expression.

The regex.h header file declares the following macros:

- Values of the *cflags* parameter of the regcomp() function: REG\_EXTENDED, REG\_ICASE, REG\_NEWLINE, REG\_NOSUB
- Values of the *eflags* parameter of the regexec() function: REG\_NOTBOL, REG\_NOTEOL
- Values of the *errcode* parameter of the regerror() function: REG\_\*

---

### regexp.h

The regexp.h header file contains regular expression declarations.

---

### rexec.h

The rexec.h header file declares the rexec() function.

---

### search.h

The search.h header file contains definitions for searching tables.

---

### setjmp.h

The setjmp.h header file contains function declarations for longjmp() and setjmp(), which use the system stack to affect the program state. It also defines one buffer type, jmp\_buf, that the setjmp() and longjmp() functions use to save and restore the program state.

\_POSIX\_SOURCE

setjmp.h declares functions siglongjmp() and sigsetjmp() and defines a buffer type sigjmp\_buf used by siglongjmp() and sigsetjmp().

\_XOPEN\_SOURCE\_EXTENDED 1

setjmp.h declares the functions \_longjmp() and \_setjmp(). See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information about using POSIX support.

## signal.h

The signal.h header file defines the following values.

- Functions:

raise()	signal()
---------	----------

- Macros:

SIG_DFL	SIG_ERR	SIG_IGN	SIG_PROMOTE
---------	---------	---------	-------------

- Signals:

SIGABND	SIGABRT	SIGFPE	SIGILL	SIGINT
SIGIOERR	SIGSEGV	SIGTERM	SIGUSR1	SIGUSR2

- The type `sig_atomic_t`, which is the largest integer type the processor can load or store automatically in the presence of asynchronous interrupts.

The following functions are supported only in a POSIX program. You must specify the POSIX(ON) run-time option for these functions. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information.

kill()	sigaction()	__sigactionset()	sigaddset()	sigdelset()
sigemptyset()	sigfillset()	sigismember()	siglongjmp()	sigpending()
sigprocmask()	sigsuspend()	sigtimedwait()	sigwait()	sigwaitinfo()

The following values are available in OS/390 UNIX services only:

- Signals:

SIGALRM	SIGCHLD	SIGCONT	SIGHUP	SIGKILL
SIGPIPE	SIGQUIT	SIGSTOP	SIGTTIN	SIGTTOU
SIGTSTP	SIGIO	SIGTRAP	SIGCLD	SIGDCE

- The structures `sigaction`, `__sigactionset_t`, `__sigactionset_s`, `sigset_t`, and `pid_t`.
- *options* arguments for `sigprocmask()`: `SIG_BLOCK`, `SIG_UNBLOCK`, and `SIG_SETMASK`.
- Flags for the `sa_flags` field, available in OS/390 UNIX services only: `SA_NOCLDSTOP` and `_SA_OLD_STYLE`.

The following values are available under `_XOPEN_SOURCE_EXTENDED 1` only:

- Signals:

SIGBUS	SIGPOLL	SIGPROF	SIGSYS	SIGURG
SIGXCPU	SIGXFSZ	SIGVTALRM	SIGWINCH	

- Functions:

bsd_signal()	killpg()	sigaltstack()	sighold()	sigignore()
siginterrupt()	sigpause()	sigrelse()	sigset()	sigstack()

---

### spawn.h

The spawn.h header file contains spawn() constants and inheritance structure.

---

### spc.h

This header file is supported only for OS/390 C applications.

The spc.h header file contains declarations for the functions available in the system programming environment, as described in “Using the System Programming C Facility” in *OS/390 C/C++ Programming Guide*. The functions are:

edcxregs	edcxusr	edcxusr2	__xhotc()	__xhotl()
__xhott()	__xhotu()	__xregs()	__xsacc()	__xsrvc()
__xusr()	__xusr2()	__24malc()	__4kmalc()	

The spc.h header file also declares these functions, used for the allocation of storage and writing of strings, (which are described in Volume 1 - Part 3. Library Functions):

calloc()	free()	malloc()	realloc()	sprintf()
----------	--------	----------	-----------	-----------

---

### stdarg.h

The stdarg.h header file defines macros used to access arguments in functions with variable-length argument lists:

va_arg()	va_start()	va_end()
----------	------------	----------

The stdarg.h header file also defines the structure `va_list`.

---

### stddef.h

The stddef.h header file contains definitions of the commonly used pointers, variables, and types, from the typedef statements, as listed below:

`ptrdiff_t` The signed integral type of the result of subtracting two pointers

`size_t` typedef for the type of the value returned by *sizeof*. OS/390 C/C++ defines `size_t` as an unsigned int

`wchar_t` typedef for a wide-character constant. OS/390 C/C++ defines `wchar_t` as an unsigned short.

`stddef.h` defines the macros `NULL` and `offsetof`. `NULL` is a pointer that never points to a data object. The `offsetof` macro expands to the number of bytes between a structure member and the start of the structure. The `offsetof` macro has the form `offsetof(structure_type, member)`

For more information about the effect of locale, see `setlocale()`, `locale.h`, or look up the individual functions in this book. For still more information, see the chapter, “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*.



## stdio.h

The stdio.h header file declares functions that deal with standard input and output. One of these functions, fdopen(), is supported only in a POSIX program. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information.

The stdio.h header file also declares these functions:

clearerr()	clrmemf()	fclose()	fdelrec()	feof()
ferror()	fflush()	fgetc()	fgetpos()	fgets()
fldata()	flocate()	fopen()	fprintf()	fputc()
fputs()	fread()	freopen()	fscanf()	fseek()
fsetpos()	ftell()	fupdate()	fwrite()	getc()
getchar()	gets()	perror()	printf()	putc()
putchar()	puts()	remove()	rename()	rewind()
scanf()	setbuf()	setvbuf()	sprintf()	sscanf()
svc99()	tmpfile()	tmpnam()	ungetc()	vfprintf()
vprintf()	vsprintf()			

## Defined Types in stdio.h

The FILE type is defined in stdio.h. Stream functions use a pointer to the FILE type to get access to a given stream. The system uses the information in the FILE structure to maintain the stream. The C standard streams stdin, stdout, and stderr are also defined in stdio.h.

The type fpos\_t is defined in stdio.h for use with fgetpos() and fsetpos().

The types \_\_S99parms, \_\_S99rbx\_t, and \_\_S99emparms\_t are defined in stdio.h for use with the svc99() function.

The type fldata\_t is defined in stdio.h for use with the fldata() function.

The types \_\_amrc\_type and \_\_amrc2\_type are defined in stdio.h for use in determining error information when I/O functions fail.

## Macros Defined in stdio.h

You can use these macros as constants in your programs, but you should not alter their values.

BUFSIZ	Specifies the buffer size to be used by the setbuf() library function when you are allocating buffers for stream I/O. This value is the expected size of the user's buffer supplied to setbuf(). If a larger buffer is required, for example, if blocksize is larger than BUFSIZ, or if special buffer attributes are required, OS/390 C/C++ applications will not use the user's buffer.
EOF	The value returned by an I/O function when the end of the file (or in some cases, an error) is found.
FOPEN_MAX	The maximum number of files that can be open simultaneously.
FILENAME_MAX	The maximum number of characters in a filename. Can be used in the size specification of an array (for example, to hold the filename returned by fldata()).

<code>L_tmpnam</code>	The size of the longest temporary name that can be generated by the <code>tmpnam()</code> function.
<code>L_ctermid</code>	Maximum size of a character array for <code>ctermid()</code> output. This macro is supported only in a POSIX program. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information.
<code>NULL</code>	A pointer which never points to a data object.
<code>TMP_MAX</code>	The maximum number of unique file names that can be generated by the <code>tmpnam()</code> function.

The macros `SEEK_CUR`, `SEEK_END`, and `SEEK_SET` expand to integral constant expressions and can be used as the third argument to `fseek()`.

The macros `_IOFBF`, `_IOLBF`, and `_IONBF` expand to integral constant expressions with distinct values suitable for use as the third argument to the `setvbuf()` function.

The following macros expand to integer constant expressions suitable for interpreting values returned by `fldata()`, in the `fldata_t` structure.

<code>__APPEND</code>	<code>__BINARY</code>	<code>__DISK</code>	<code>__DUMMY</code>
<code>__ESDS</code>	<code>__ESDS_PATH</code>	<code>__HFS</code>	<code>__HIPERSPACE</code>
<code>__KSDS</code>	<code>__KSDS_PATH</code>	<code>__MEMORY</code>	<code>__MSGFILE</code>
<code>__NORLS</code>	<code>__NOTVSAM</code>	<code>__OTHER</code>	<code>__PRINTER</code>
<code>__READ</code>	<code>__RECORD</code>	<code>__RLS</code>	<code>__RSDS</code>
<code>__TAPE</code>	<code>__TDQ</code>	<code>__TERMINAL</code>	<code>__TEXT</code>
<code>__UPDATE</code>	<code>__WRITE</code>		

The following macros expand to integral constant expressions suitable for use as the fourth argument to the `flocate()` function.

<code>__KEY_EQ</code>	<code>__KEY_EQ_BWD</code>	<code>__KEY_FIRST</code>	<code>__KEY_GE</code>
<code>__KEY_LAST</code>	<code>__RBA_EQ</code>	<code>__RBA_EQ_BWD</code>	

The following macros expand to integral constant expressions suitable for use as the argument to the function `clrmemf()`.

<code>__CURRENT</code>	<code>__CURRENT_LOWER</code>	<code>__LOWER</code>
------------------------	------------------------------	----------------------

The following macros expand to integral constant expressions suitable for use to determine the last operation reported in the `__amrc_type` structure. All these macros are described in the *OS/390 C/C++ Programming Guide*.

__BSAM_BLDL	__BSAM_CLOSE	__BSAM_CLOSE_T
__BSAM_NOTE	__BSAM_OPEN	__BSAM_POINT
__BSAM_READ	__BSAM_STOW	__BSAM_WRITE
__C_CANNOT_EXTEND	__C_DBCS_SI_TRUNCATE	__C_DBCS_SO_TRUNCATE
__C_DBCS_TRUNCATE	__C_DBCS_UNEVEN	__C_FCBCHECK
__C_TRUNCATE	__CELMSGF_WRITE	__CICS_WRITEQ_TD
__HSP_CREATE	__HSP_DELETE	__HSP_EXTEND
__HSP_READ	__HSP_WRITE	__INTERCEPT_READ
__INTERCEPT_WRITE	__IO_CATALOG	__IO_DEVTYPE
__IO_INIT	__IO_LOCATE	__IO_OBTAIN
__IO_RDJFCB	__IO_RENAME	__IO_SCRATCH
__IO_TRKCALC	__IO_UNCATALOG	__LFS_CLOSE
__LFS_FSTAT	__LFS_LSEEK	__LFS_OPEN
__LFS_READ	__LFS_WRITE	__NOSEEK_REWIND
__OS_CLOSE	__OS_OPEN	__QSAM_FREEPOOL
__QSAM_GET	__QSAM_PUT	__QSAM_RELSE
__QSAM_TRUNC	__SVC99_ALLOC	__SVC99_ALLOC_NEW
__SVC99_UNALLOC	__TGET_READ	__TPUT_WRITE
__VSAM_CLOSE	__VSAM_ENDREQ	__VSAM_ERASE
__VSAM_GENCB	__VSAM_GET	__VSAM_MODCB
__VSAM_OPEN_ESDS	__VSAM_OPEN_ESDS_PATH	__VSAM_OPEN_FAIL
__VSAM_OPEN_KSDS	__VSAM_OPEN_KSDS_PATH	__VSAM_OPEN_RRDS
__VSAM_POINT	__VSAM_PUT	__VSAM_SHOWCB
__VSAM_TESTCB		

---

## stdlib.h

The stdlib.h header file contains declarations for the following functions:

abort()	abs()[1]	alloca()[1]	atexit()	atof()
atoi()	atol()	bsearch()	calloc()	cds()[1]
clearenv()	cs()[1]	csid()	div()	exit()
fetch()[2]	fetchep()[2]	free()	getenv()	labs()
ldiv()	__librel()	malloc()	mblen()	mbstowcs()
mbtowc()	qsort()	rand()	realloc()	release()[2]
rpmatch()	setenv()	srand()	strtod()	strtol()
strtoul()	system()	wcsid()	wcstombs()	wctomb()

[1] Built-in function.

[2] Not supported under C++ applications.

Two type definitions are added to stdlib.h for the Compare and Swap functions cs() and cds(). The structures defined are \_\_cs\_t and \_\_cds\_t.

The type size\_t is declared in the header file. It is used for the type of the value returned by sizeof. The type wchar\_t is declared and used for a wide character constant. For more information on the types size\_t and wchar\_t, see “stddef.h” on page 42.

The stdlib.h declares div\_t and ldiv\_t, which define the structure types that are returned by div() and ldiv().

The stdlib.h file also contains definitions for the following macros:

NULL                      The NULL pointer constant (also defined in stddef.h).

EXIT_SUCCESS	Used by the <code>atexit()</code> function.
EXIT_FAILURE	Used by the <code>atexit()</code> function.
RAND_MAX	Expands to an integer representing the largest number that the <code>RAND</code> function can return.
MB_CUR_MAX	<p>Expands to an integer representing the maximum number of bytes in a multibyte character. This value is dependent on the current locale.</p> <p>If <code>MB_CUR_MAX</code> is set to 1, multibyte functions will behave as if all multibyte characters are one byte long; wide-character functions are <i>not</i> supported and full DBCS support is <i>not</i> provided. If <code>MB_CUR_MAX</code> is 4, all DBCS support provided by the library is enabled.</p>

---

## string.h

The `string.h` header file declares the string manipulation functions and their built-in versions:

No feature test macro required

<code>memchr()[1]</code>	<code>memcmp()[1]</code>	<code>memcpy()[1]</code>	<code>memmove()</code>	<code>memset()[1]</code>
<code>strcat()[1]</code>	<code>strchr()[1]</code>	<code>strcmp()[1]</code>	<code>strcoll()</code>	<code>strcpy()[1]</code>
<code>strcspn()</code>	<code>strerror()</code>	<code>strlen()[1]</code>	<code>strncat()[2]</code>	<code>strncmp()[2]</code>
<code>strncpy()[2]</code>	<code>strpbrk()</code>	<code>strrchr()[1]</code>	<code>strspn()</code>	<code>strstr()</code>
<code>strtok()</code>	<code>strxfrm()</code>			

[1] Built-in function.

[2] Built-in function for 3.1 and above compiler.

`_XOPEN_SOURCE`

`memccpy()`

`_XOPEN_SOURCE_EXTENDED 1`

`strdup()`

The `string.h` header file also defines the macro `NULL` and the type `size_t`. For more information see “`stddef.h`” on page 42.

---

## strings.h

The `strings.h` header file contains definitions for string operations.

---

## stropts.h

The `stropts.h` header file declares the following functions:

<code>fattach()</code>	<code>fdetach()</code>	<code>getmsg()</code>	<code>getpmsg()</code>
<code>ioctl()</code>	<code>isastream()</code>	<code>putmsg()</code>	<code>putpmsg()</code>

---

## syslog.h

The syslog.h header file contains definitions for system error logging.

---

## sys/file.h

The sys/file.h header file defines file manipulation constants.

---

## sys/\_\_getipc.h

The sys/\_\_getipc.h header file contains definitions to get interprocess communication information.

---

## sys/ioctl.h

The sys/ioctl.h header file contains system I/O definitions and structures.

---

## sys/ipc.h

The sys/ipc.h header file contains definitions for the interprocess communication access structure.

---

## sys/mman.h

The sys/mman.h header file contains memory management declarations.

---

## sys/mntent.h

This header file is supported only in an \_OPEN\_SYS program. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information.

The sys/mntent.h header file declares the w\_getmntent() function and it defines the structures w\_mnth and w\_mntent, along with some related constants.

---

## sys/modes.h

The sys/modes.h header file contains several macro definitions:

- Defined constant masks and bits for values of type mode\_t, such as the st\_mode field of the stat struct
- A defined constant mask for the st\_genvalue field of the stat struct
- Function-like macros for testing values of the st\_mode field of the stat struct.

Under OS/390 C support, these definitions are included in sys/stat.h to make sys/stat.h conform with POSIX.

---

### sys/\_\_messag.h

The sys/\_\_messag.h header file contains definitions for the \_\_console() function.

---

### sys/msg.h

The sys/msg.h header file contains definitions for message queue structures.

---

### sys/ps.h

The sys/ps.h header file declares the w\_getpsent() function that provides process data and defines the structure w\_psproc along with some related constants.

It requires the \_OPEN\_SOURCE 1 feature test macro.

---

### sys/resource.h

The sys/resource.h header file contains definitions for XSI resource operations.

---

### sys/sem.h

The sys/sem.h header file contains definitions for the semaphore facility.

---

### sys/server.h

The sys/server.h header file contains definitions for using Work Load Manager services.

---

### sys/shm.h

The sys/shm.h header file contains definitions for the shared memory facility.

---

### sys/socket.h

The sys/socket.h header file contains sockets definitions.

---

### sys/stat.h

The sys/stat.h header file declares the following functions related to OS/390 UNIX files and their access:

chaudit()	chmod()	creat()	fchaudit()	fchmod()
fstat()	lstat()	mkdir()	mkfifo()	mknod()
mount()	__open_stat()	stat()	umask()	umount()

---

## sys/statfs.h

The sys/statfs.h header file declares the `w_statfs()` function that provides file system status and the `w_statfs` structure. It requires the `_OPEN_SYS` feature test macro.

---

## sys/statvfs.h

The sys/statvfs.h header file contains definitions for file system status.

---

## sys/time.h

The sys/time.h header file contains definitions for time types.

---

## sys/timeb.h

The sys/timeb.h header file contains additional definitions for date and time.

---

## sys/times.h

The sys/times.h header file declares the `times()` function that gets processor times for use by processes. It requires the `_POSIX_SOURCE` feature test macro.

---

## sys/ttydev.h

The sys/ttydev.h header file defines constants used by the terminal I/O functions.

---

## sys/types.h

The sys/types.h header file defines a collection of *typedef* symbols and structures.

Table 9. `_POSIX_SOURCE`

---

<code>dev_t</code>	Device numbers
<code>gid_t</code>	Group IDs
<code>ino_t</code>	File serial numbers
<code>mode_t</code>	Some file attributes
<code>nlink_t</code>	Link counts
<code>off_t</code>	File sizes
<code>pid_t</code>	Process IDs and process group ids
<code>size_t</code>	Unsigned integer
<code>ssize_t</code>	Signed arithmetic
<code>uid_t</code>	User IDs
<code>time_t</code>	Time values
<code>clock_t</code>	Time values
<code>sigset_t</code>	Signal set
<code>cc_t</code>	<code>cc_t</code>
<code>tty control chars</code>	
<code>speed_t</code>	tty baud rate
<code>tcflag_t</code>	tty modes
<code>mtm_t</code>	Mount requests
<code>rdev_t</code>	Device numbers

Table 10. *\_OPEN\_THREADS*

pthread_t	Identify a thread
pthread_attr_t	Identify a thread attribute object
pthread_mutex_t	Mutexes
pthread_mutexattr_t	Identify a mutex attribute object
pthread_cond_t	Condition variables
pthread_condattr_t	Identify a condition attribute object
pthread_key_t	Thread-specific data keys
pthread_once_t	Dynamic package initialization

Table 11. *\_XOPEN\_SOURCE*

key_t	Interprocess communications
-------	-----------------------------

Table 12. *\_XOPEN\_SOURCE\_EXTENDED 1*

id_t	General identifier, can contain a pid_t or a gid_t
useconds_t	Microseconds
sa_family_t	Address family
in_port_t	AF_INET port

Table 13. *\_OE\_SOCKETS or \_ALL\_SOURCE*

u_char	Unsigned char
u_int	Unsigned int
ushort	Unsigned short
u_short	Unsigned short
u_long	Unsigned long

Table 14. *\_OE\_SOCKETS or \_XOPEN\_SOURCE\_EXTENDED 1*

in_addr_t	Internet address
ip_addr_t	Internet address
caddr_t	Used for message data pointer

---

## sys/uio.h

The sys/uio.h header file contains definitions for vector I/O operations.

---

## sys/un.h

The sys/un.h header file contains definitions for UNIX-domain sockets.

---

## sys/\_\_ussos.h

The sys/\_\_ussos.h header file contains the `_SET_THLIPADDR()` macro, which sets a client's IP address for security facility authorization (SAF).

---

## sys/utsname.h

The sys/utsname.h header file declares the `utsname` structure and the `uname()` function, which returns the name of the current operating system. It requires the `_POSIX_SOURCE` feature test macro.



---

## sys/wait.h

The sys/wait.h header file declares the following functions, used for holding processes.

\_POSIX\_SOURCE

wait()                      waitpid()

\_XOPEN\_SOURCE\_EXTENDED 1

waitid()                    wait3()

---

## sys/\_\_wlm.h

The sys/\_\_wlm.h header file contains definitions for Work Load Manager functions.

---

## tar.h

The tar.h header file contains definitions for the tar utility.

---

## terminat.h

The terminat.h header file, which is used for OS/390 C++ exception handling, declares the terminate() and set\_terminate() functions.

---

## termios.h

The termios.h header file contains constants, prototypes, and typedef definitions of POSIX terminal I/O functions. It includes the \_\_termcp structure, and declares the following functions:

cfgetispeed()	cfgetospeed()	cfsetispeed()	cfsetospeed()	tcdrain()
tcflow()	tcflush()	tcgetattr()	__tcgetcp()	tcgetsid()
tcsendbreak()	tcsetattr()	__tcsetcp()	__tcsettables()	

These functions are supported only in a POSIX program. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information.

In addition, this header file declares the w\_ioctl() function, which is callable by both C and C++ applications.

---

## time.h

The time.h header file declares the time and date functions:

asctime()	clock()	ctime()	difftime()	gmtime()
localtime()	mktime()	strptime()	strptime()	time()
tzset()[1]				

[1] These functions are supported only in a POSIX program. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information.

The `time.h` header file also provides:

- A structure `timespec` containing the following members:

<code>time_tv_sec;</code>	seconds
<code>long tv_nsec;</code>	nanoseconds

- A structure `tm` containing the components of a calendar time. See “`gmtime()` — Convert Time to Broken-Down UTC Time” on page 640 for a list of the members of the `tm` structure. This structure is used by the functions `asctime()`, `gmtime()`, `localtime()`, `mktime()`, `strftime()`, and `strptime()`.
- A macro `CLOCKS_PER_SEC` equal to the number per second of the value returned by the `clock()` function.
- Types `clock_t`, `time_t`, and `size_t`.
- The `NULL` pointer constant. For more information on `NULL` and the type `size_t`, see “`stddef.h`” on page 42.
- The macro `CLK_TCK`, which is the number of clock ticks per second, is provided for compatibility with earlier releases. It was used in connection with the return value of the `clock()` function. This use has been superseded by the `CLOCKS_PER_SEC` macro.

`CLK_TCK` is also used by the OS/390 UNIX services `times()` function. In this context, we recommend using the OS/390 UNIX services `sysconf()` function, which will eventually supersede `CLK_TCK`. (See “`sys/times.h`” on page 49 and “`unistd.h`” on page 53.)

The time functions are affected by the current locale selected. The `LC_CTYPE` category affects the behavior of the `strftime()`, `strptime()`, and `wcsftime()` functions. The `LC_TOD` category affects the behavior of the `gmtime()`, `mktime()`, and `localtime()` functions.

---

## ucontext.h

The `ucontext.h` header file contains the prototypes and definitions needed by the following functions:

<code>getcontext()</code>	<code>setcontext()</code>	<code>makecontext()</code>	<code>swapcontext()</code>
---------------------------	---------------------------	----------------------------	----------------------------

---

## uheap.h

The `uheap.h` header file contains the prototypes and definitions needed by the following functions:

<code>ucreate()</code>	<code>umalloc()</code>	<code>ufree()</code>	<code>uheapreport()</code>
------------------------	------------------------	----------------------	----------------------------

---

## unexpected.h

The `unexpected.h` header file, which is used for OS/390 C++ exception handling, declares the `unexpected()` and `set_unexpected()` functions.

---

## unistd.h

The unistd.h header file declares a number of functions.

access()	__certificate()	chdir()	chown()
close()	dup()	dup2()	extlink_np()
fchown()	fpathconf()	fsync()	ftruncate()
getcwd()	__isPosixOn()	link()	__login()
lseek()	__openMvsRel()	pathconf()	__pid_affinity()
read()	readlink()	rmdir()	symlink()
sysconf()	unlink()	write()	__wsinit()

The functions in a second group declared in unistd.h are supported only in a POSIX program. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information.

alarm()	ctermid()	execl()	execle()
execvp()	execv()	execve()	execvp()
_exit()	fork()	getegid()	geteuid()
getgid()	getgroups()	getgroupsbyname()	getlogin()
getpid()	getppid()	isatty()	pause()
pipe()	setegid()	seteuid()	setpgid()
setsid()	sleep()	tcgetpgrp()	tcsetpgrp()
ttyname()			

The unistd.h header file defines the following macros, which are constants that map to the standard streams' file descriptors:

STDIN_FILENO	STDOUT_FILENO	STDERR_FILENO
--------------	---------------	---------------

The unistd.h defines the following symbols, each of which stands for a configuration variable:

__CERTIFICATE_REGISTER 2	__CERTIFICATE_DEREGISTER 3
__LOGIN_CREATE	__LOGIN_USERID
_PC_CHOWN_RESTRICTED	_PC_LINK_MAX
_PC_MAX_CANON	_PC_MAX_INPUT
_PC_NAME_MAX	_PC_NO_TRUNC
_PC_PATH_MAX	_PC_PIPE_BUF
_PC_VDISABLE	

---

## ulimit.h

The ulimit.h header file contains definitions for ulimit commands.

---

## utime.h

The utime.h header file declares the utimbuf structure and the utime() function, which is used to set file access and modification times.

The utime() function is supported only in a POSIX program. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information.

---

### utmpx.h

The utmpx.h header file contains user accounting database definitions.

---

### varargs.h

The varargs.h header file contains definitions for handling variable argument lists.

---

### variant.h

The variant.h header file declares the getsyntax() function, which returns LC\_SYNTAX characters. It also contains the declaration of the variant structure:

```
struct variant {
    char    *codeset;           /* code set of the current locale */
    char    backslash;          /* encoding of \ */
    char    right_bracket;      /* encoding of ] */
    char    left_bracket;       /* encoding of [ */
    char    right_brace;        /* encoding of } */
    char    left_brace;         /* encoding of { */
    char    circumflex;         /* encoding of ^ */
    char    tilde;              /* encoding of ~ */
    char    exclamation_mark;   /* encoding of ! */
    char    number_sign;        /* encoding of # */
    char    vertical_line;      /* encoding of | */
    char    dollar_sign;        /* encoding of $ */
    char    commercial_at;      /* encoding of @ */
    char    grave_accent;       /* encoding of ` */
};

struct variant *getsyntax(void);
```

For more information about the effect of locale, see setlocale(), locale.h, or look up the individual functions in this book. For still more information, see the chapter, “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*.

---

### wchar.h

The wchar.h header file contains the declaration for the supported subset of the ISO/C Multibyte Support extensions defined in ISO/IEC 9899:1990/Amendment 1:1993(E) extensions. The following functions are declared in wchar.h:

fgetwc()	fgetws()	fputwc()	fputws()	getwc()
getwchar()	mbrlen()	mbrtowc()	mbsinit()	mbsrtowcs()
putwc()	putwchar()	swprintf()	swscanf()	ungetwc()
vswprintf()	wcrtomb()	wscat()	wcschr()	wcscmp()
wscoll()	wscpy()	wscspn()	wcsftime()	wcslen()
wcsncat()	wcsncmp()	wcsncpy()	wcspbrk()	wcsrchr()
wcsrtombs()	wcsspn()	wcsstr()	wcstod()	wcstok()
wcstol()	wcstoul()	wcswidth()	wcsxfrm()	wctob()
wcwidth()				

You don't need to include stdio.h and stdarg.h to use the header file.

The header file `wchar.h` contains definitions of the following types:

<code>mbstate_t</code>	Conversion-state information needed when converting between sequences of multibyte characters and wide characters.
<code>size_t</code>	typedef for the type of the value returned by <i>sizeof</i> .
<code>wchar_t</code>	typedef for a wide-character constant.
<code>wint_t</code>	An integral type unchanged by integral promotions that can hold any value corresponding to members of the extended character set, as well as WEOF (see below).
<code>FILE</code>	The <code>FILE</code> structure type is defined in both <code>stdio.h</code> and <code>wchar.h</code> . Stream functions use a pointer to the <code>FILE</code> type to get access to a given stream. The system uses the information in the <code>FILE</code> structure to maintain the stream. The C standard streams <code>stdin</code> , <code>stdout</code> , and <code>stderr</code> are also defined in <code>stdio.h</code> .
<code>va_list</code>	This type is defined in both <code>stdarg.h</code> and <code>wchar.h</code> .

The header file `wchar.h` also contains definitions of the following constants:

<b>NULL</b>	A pointer that never points to a data object.
<b>WEOF</b>	Expands to a constant expression of type <code>wint_t</code> , whose value does not correspond to any member of the extended character set. It indicates EOF.

You can perform wide-character input/output on the streams described in the ISO/IEC 9899:1990 standard, subclause 7.9.2. This standard expands the definition of a stream to include an *orientation* for both text and binary streams. For more information about DBCS orientation, see the section on Double-Byte Character Sets in the *OS/390 C/C++ Programming Guide*.

The wide-character string functions are also declared in `wcstr.h` for compatibility with previous releases of C/370, although `wcstr.h` may be withdrawn in the future.

For more information about the effect of locale, see `setlocale()`, `locale.h`, or look up the individual functions in this book. For still more information, see the chapter, “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*.

---

## wcstr.h

The `wcstr.h` header file declares the following multibyte functions:

<code>wcscat()</code>	<code>wcschr()</code>	<code>wscmp()</code>	<code>wscpy()</code>	<code>wcscspn()</code>
<code>wcslen()</code>	<code>wcsncat()</code>	<code>wcsncmp()</code>	<code>wcsncpy()</code>	<code>wcspbrk()</code>
<code>wcsrchr()</code>	<code>wcsspn()</code>	<code>wcswcs()</code>		

`wcstr.h` also defines the types `size_t`, `NULL`, `wchar_t`, and `wint_t`.

The wide-character string functions are also declared in `wchar.h` for compatibility with previous releases of C/370. `wcstr.h` may be withdrawn in future releases of the OS/390 C/C++ product.

---

## wctype.h

The wctype.h function declares functions that deal with the wide-character set.

iswalnum()	iswalpha()	iswblank()	iswcntrl()	iswctype()
iswdigit()	iswgraph()	iswlower()	iswprint()	iswpunct()
iswspace()	iswupper()	iswxdigit()	towlower()	towupper()
wctype()				

The wctype.h defines the types `wint_t` and `wctype_t`. The wctype.h defines the macro `WEOF`, which expands to a constant expression of type `wint_t`, whose value does not correspond to any member of the extended character set. The macro `WEOF` indicates EOF.

---

## wordexp.h

The wordexp.h header file contains definitions for word expansion types.

---

## xti.h

The xti.h header file declares the following under the `_XOPEN_SOURCE_EXTENDED 1` feature test macro:

Symbolic constants

*Table 15 (Page 1 of 4). Symbolic constants defined in xti.h*

Symbolic Constant	Description
TBADADDR	Incorrect addr format
TBADOPT	Incorrect option format
TACCES	Incorrect permissions
TBADF	Illegal transport fd
TNOADDR	Couldn not allocate addr
TOUTSTATE	Out of state
TBADSEQ	Bad call sequence number
TSYSERR	System error
TLOOK	Event requires attention
TBADDATA	Illegal amount of data
TBUFOVFLW	Buffer not large enough
TFLOW	Flow control
TNODATA	No data
TNODIS	Discon_ind not found on queue
TNOUDERR	Unitdata error not found
TBADFLAG	Bad flags
TNOREL	No ord rel found on queue
TNOTSUPPORT	Primitive not supported
TSTATECHNG	State currently changing
TNOSTRUCTYPE	Unknown struct-type requested
TBADNAME	Invalid transport name
TBADQLEN	Qlen is zero
TADDRBUSY	Address in use
TINDOUT	Outstanding connect indications
TPROVMISMATCH	Transport provider mismatch
TRESQLEN	Resfd specified to accept w/qlen>0
TRESADDR	Resfd not bound to same addr as fd
TQFULL	Incoming connection queue full
TPROTO	XTI protocol error

Table 15 (Page 2 of 4). Symbolic constants defined in xti.h

Symbolic Constant	Description
T_LISTEN	Connection indication received
T_CONNECT	Connect confirmation received
T_DATA	Normal data received
T_EXDATA	Expedited data received
T_DISCONNECT	Disconnect received
T_UDERR	Data gram error indication
T_ORDREL	Orderly release indication
T_GODATA	Sending normal data is possible
T_GOEXDATA	Sending expedited data is possible
T_EVENTS	Event mask
T_MORE	More data
T_EXPEDITED	Expedited data
T_NEGOTIATE	Set opts
T_CHECK	Check opts
T_DEFAULT	Get default opts
T_SUCCESS	Successful
T_FAILURE	Failure
T_CURRENT	Current opts
T_PARTSUCCESS	Partial success
T_READONLY	Read-only
T_NOTSUPPORT	Not supported
T_BIND	Struct t_bind
T_OPTMGMT	Struct t_optmgmt
T_CALL	Struct t_call
T_DIS	Struct t_discon
T_UNITDATA	Struct t_unitdata
T_UDERROR	Struct t_uderr
T_INFO	Struct t_info
T_ADDR	Address
T_OPT	Options
T_UDATA	User data
T_ALL	All the above
T_COTS	Connection oriented transport
T_COTS_ORD	Connection oriented w/ orderly release
T_CLTS	Connectionless transport service
T_SENDZERO	Supports 0-length TSDUs
T_UNINIT	Uninitialized
T_UNBND	Unbound
T_IDLE	Idle
T_OUTCON	Outgoing connection pending
T_INCON	Incoming connection pending
T_DATAXFER	Data transfer
T_OUTREL	Outgoing release pending
T_INREL	Incoming release pending
T_NOSTATES	Number of states
T_YES	True
T_NO	False
T_UNUSED	Unused value
T_NULL	Null value
T_ABSREQ	Absolute requirement
T_INFINITE	Infinity
T_INVALID	Invalid value
T_UNSPEC	Unspecified value (applicable to char, short, long, etc.)
T_ALLOPT	Process all options
XTI_GENERIC	Generic XTI options level
XTI_DEBUG	Enable debugging

Table 15 (Page 3 of 4). Symbolic constants defined in *xti.h*

Symbolic Constant	Description
XTI_LINGER	Linger on close if data present
XTI_RCVBUF	Receive buffer size
XTI_RCVLOWAT	Receive low-water mark
XTI_SNDBUF	Send buffer size
XTI_SNDLOWAT	Send low-water mark
T_CLASS0	ISO transport class 0
T_CLASS1	ISO transport class 1
T_CLASS2	ISO transport class 2
T_CLASS3	ISO transport class 3
T_CLASS4	ISO transport class 4
T_PRITOP	ISO top priority
T_PRIHIGH	ISO high priority
T_PRIMID	ISO medium priority
T_PRILOW	ISO low priority
T_PRIDFLT	ISO default priority
T_NOPROTECT	ISO no protection
T_PASSIVEPROTECT	ISO passive protection
T_ACTIVEPROTECT	ISO active protection
T_LTPDUFLT	ISO TPDU default length
ISO_TP	ISO options class
TCO_THROUGHPUT	ISO throughput option
TCO_TRANSDEL	ISO transit delay option
TCO_RESERRORRATE	ISO residual error rate option
TCO_TRANSFAILPROB	ISO transfer failure probability option
TCO_ESTFAILPROB	ISO connection establishment failure probability option
TCO_RELFAILPROB	ISO connection release failure probability option
TCO_ESTDELAY	ISO connection establishment delay option
TCO_RELDELAY	ISO connection release delay option
TCO_CONNRRESIL	ISO connection resilience delay
TCO_PROTECTION	ISO protection option
TCO_PRIORITY	ISO priority option
TCO_EXPD	ISO expedited data
TCL_TRANSDEL	ISO CLTS transit delay option
TCL_RESERRORRATE	ISO CLTS residual error rate option
TCL_PROTECTION	ISO CLTS protection option
TCL_PRIORITY	ISO CLTS priority option
TCO_LTPDU	ISO mgmt max TPDU length option
TCO_ACKTIME	ISO mgmt acknowledge time option
TCO_REASTIME	ISO mgmt reassignment time option
TCO_EXTFORM	ISO mgmt extended format option
TCO_FLOWCTRL	ISO mgmt flow control option
TCO_CHECKSUM	ISO mgmt checksum option
TCO_NETEXP	ISO mgmt network expedited data option
TCO_NETRECPCTF	ISO mgmt network receipt confirmation option
TCO_PREFCLASS	ISO mgmt preferred class option
TCO_ALTCLASS1	ISO mgmt 1st alternative class option
TCO_ALTCLASS2	ISO mgmt 2nd alternative class option
TCO_ALTCLASS3	ISO mgmt 3rd alternative class option
TCO_ALTCLASS4	ISO mgmt 4th alternative class option
TCL_CHECKSUM	ISO CLTS checksum option
INET_TCP	TCP options level
TCP_NODELAY	TCP don't delay packets to coalesce option
TCP_MAXSEG	TCP get maximum segment size option
TCP_KEEPAIVE	TCP check if connections are alive option
T_GARBAGE	Garbage value for <code>t_kpalive</code> <code>kp_timeout</code> value
INET_UDP	UDP options level



Table 15 (Page 4 of 4). Symbolic constants defined in *xti.h*

Symbolic Constant	Description
UDP_CHECKSUM	UDP checksum option
INET_IP	IP options level
IP_OPTIONS	IP per-packet options
IP_TOS	IP per-packet type of service
IP_TTL	IP per-packet time to live
IP_REUSEADDR	IP allow local address reuse option
IP_DONTROUTE	IP just use interface addresses
IP_BROADCAST	IP permit sending of broadcast msgs
T_ROUTINE	IP routine precedence option
T_PRIORITY	IP priority precedence option
T_IMMEDIATE	IP immediate precedence option
T_FLASH	IP flash precedence option
T_OVERRIDEFLASH	IP override-flash precedence option
T_CRITIC_ECP	IP critical-ECP precedence option
T_INETCONTROL	IP internet control precedence option
T_NETCONTROL	IP network control precedence option
T_NOTOS	IP TOS no distinguished type option
T_LDELAY	IP TOS low delay option
T_HITHRPT	IP TOS high throughput option
T_HIREL	IP TOS high reliability option

## Functions:

t_accept()	t_alloc()	t_bind()	t_close()
t_connect()	t_error()	t_free()	t_getinfo()
t_getprotaddr()	t_getstate()	t_listen()	t_look()
t_open()	t_optmgmt()	t_rcv()	t_rcvconnect()
t_rcvdis()	t_rcvrel()	t_rcvudata()	t_rcvuderr()
t_snd()	t_snddis()	t_sndrel()	t_sndudata()
t_strerror()	t_sync()	t_unbind()	



---

## Volume 1 - Part 3. Library Functions

This part describes the OS/390 C/C++ Run-Time Library functions, including the built-in library functions used by the OS/390 C/C++ compilers.

---

### Names

Identifiers (function names, macros, types) defined by the various standards in the headers are reserved. Also reserved are:

- Identifiers that begin with an underscore and either an uppercase letter or another underscore.
- Identifiers that end with “\_t”.

Do not use these reserved identifiers for any purpose other than those defined in the documentation.

All identifiers other than the ISO C identifiers comprise the *user's name space*. You are free to use any of these names. However, a number of names in the OS/390 C/C++ Run-Time Library encroach on the user's name space. This is a result of our desire to provide names that are meaningful and easy to remember, or to support industry-defined names, for example: `fetchep()` or `pthread_cancel()`. The header files cause these names to be renamed into reserved names and these in turn are mapped onto the external entry point names that usually are operating-system specific.

If you want to use names in the OS/390 C/C++ Run-Time Library which are in the user's name space as defined, just include the appropriate header. If you cannot include the appropriate header because it would bring in other names that collide with your own private names, but you still want to use some of the functions defined there, you can refer to these functions by their reserved internal names. These reserved names are unique, not longer than 8 characters, and usually start with a double underscore.

The IBM OS/390 C/C++ compiler automatically maps all underscores and lower-case letters in external identifiers in source code to ‘@’ characters and uppercase characters in the object deck. Thus, to refer to the `fetchep()` function without including the `stdlib.h` header, you can use its reserved internal name `__ftchep()`, which is then automatically mapped to the external entry point `@@FTCHEP`. For C++ functions, you must ensure C by declaring the functions as extern “C”.

Functions that are mapped this way have the external entry point listed in the function description in this part under the heading, “External Entry Point”.

See also the following sections in the *OS/390 C/C++ Language Reference* for more information on external names:

- “#pragma csect”
- “#pragma map”
- “External Name Mapping”

See also the following sections in the *OS/390 C/C++ User's Guide*:

- “Prelinking a C Application”

- The LONGNAME compiler option

See also “Naming Conventions” in “Using Environment Variables”, in the *OS/390 C/C++ Programming Guide* for details about external names.

---

## Standards

Each function description begins with a table to indicate the standards/extensions, language support, and dependencies. A table like this one:

Standards / Extensions	C or C++	Dependencies
ISO C	C only	POSIX(ON)
ISO C Amendment	C++ only	MVS 4.3
POSIX.1	both	MVS 5.1
POSIX.1a		MVS 5.2.2
POSIX.2		OS/390 R2
POSIX.4a		OS/390 R3
XPG4		OS/390 R6
XPG4.2		
SAA		
Language Environment		
OS/390 UNIX		
C Library		

By indicating a standard, we refer to the origin of the function, not necessarily the compliance. For example, functions that are enriched by features from XPG4 have XPG4 listed.

These are the standards referred to:

- Standards/extensions
  1. *ISO C* refers to ISO/IEC 9899 : 1990(E).
  2. *Am* underneath ISO C refers to a subset of the ISO/IEC 9899:1990/Amendment 1:1993(E).
  3. POSIX
    - *POSIX.1* refers to ISO/IEC 9945-1:1990/IEEE POSIX 1003.1-1990.
    - *POSIX.1a* refers to a subset of IEEE POSIX 1003.1a, Draft 7, May 1992.
    - *POSIX.2* refers to IEEE Portable Operating System Interface (POSIX) Part 2, P1003.2 draft 12.
    - *POSIX.4a* refers to a subset of IEEE POSIX 1003.4a, Draft 6, Feb. 26,1992.
  4. *XPG4* refers to X/Open Common Applications Environment Specification, System Interfaces and Headers, Issue 4.
  5. *XPG4.2* refers to X/Open Common Applications Environment Specification, System Interfaces and Headers, Issue 4, Version 2.

6. *Extension* refers to one of the following:

- a. *SAA* refers to the IBM Systems Application Architecture Common Programming Interface (SAA CPI) Level 2 definition of the C language.
- b. *Language Environment* refers to functions that are extensions to the conventional standards.
- c. *OS/390 UNIX* refers to functions that provide OS/390 UNIX support beyond the defined standards.
- d. *C Library* refers to the functions that are extensions to the run-time library.

- Language support

*C or C++* refers to whether the function is supported for the OS/390 C compiler, the OS/390 C++ compiler, or both.

- Dependencies

Some functions have the following dependencies identified. If the dependencies are not met, then the function fails, and returns an errno of EMVSNORTL. Functions defined by the standards that cannot fail, will cause abnormal termination and return Language Environment condition CEE5001.

- *POSIX(ON) required* refers to whether the enclave can run with the POSIX semantics.

POSIX is an application characteristic that is maintained at the enclave level. After you have established the characteristic during enclave initialization, you cannot change it.

When you set POSIX to ON, you can use functions that are unique to POSIX, such as `pthread_create()`.

One of the effects of POSIX(ON) is the enablement of POSIX signal handling semantics, which interact closely with the OS/390 Language Environment condition handling semantics. Where ambiguities exist between ANSI and POSIX semantics, the POSIX run-time setting indicates the POSIX semantics to follow. For more information about running POSIX programs, please see “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9.

- A kernel from MVS 4.3 or a higher release must be active.
- A kernel from MVS 5.1 or a higher release must be active.
- A kernel from MVS 5.2.2 or a higher release must be active.

These standards do have some overlap, as illustrated in Figure 2.

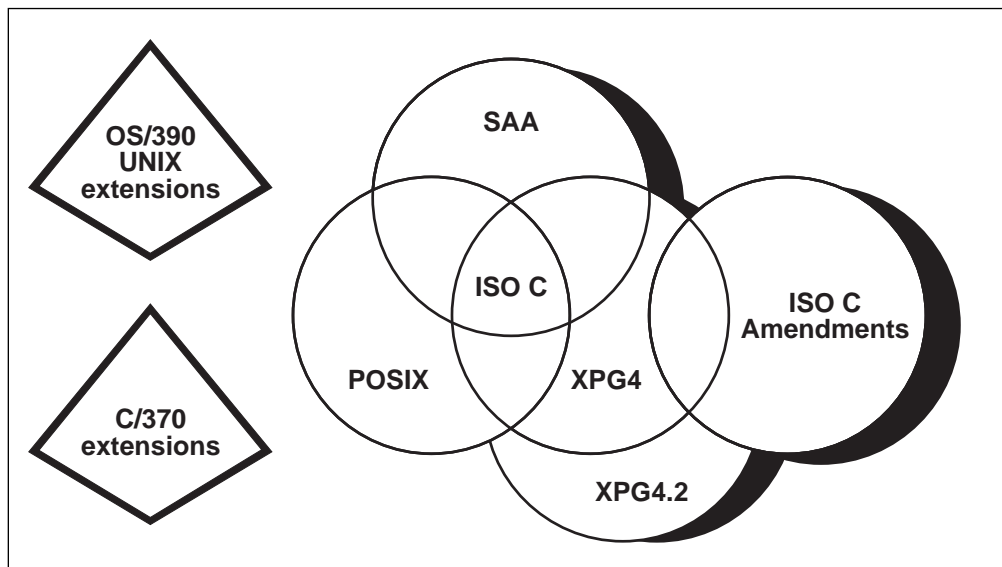


Figure 2. Overlap of C Standards and Extensions

The C library contains several functions that are extensions to the SAA CPI Level 2 definition. These library functions are available only if the `LANGLVL(EXTENDED)` compile-time option is in effect. As indicated, some of the *stub* routines for the extensions are available if you specify `LANGLVL(ANSI)`. They are made available for compatibility with Version 1; they may not be available in the future. (Within runtime libraries, a *stub routine* is a routine that contains the minimum lines of code required to locate a given routine at run time.)

Many of the symbols that are defined in headers are “protected” by a feature test macro. For information on the relationships between feature test macros and the standards, see “Feature Test Macros” on page 17.

---

## Using C Include Files from C++

If you need to use an old C header file in a C++ program, use `extern`, like this:

```
extern "C" {  
    #include "myhdr.include"  
}
```

---

## Built-in Functions

Built-in functions are ones for which the compiler generates inline code at compile time. Every call to a built-in function eliminates a runtime call to the function having the same name in the dynamic library.

Built-in functions are used by application code, while it is running, without reference to the dynamic library. Although built-in functions increase the size of a generated application slightly, this should be offset by the improved performance resulting from reducing the overhead of the dynamic calls. Built-in functions can be used with the System Programming Facilities to generate free-standing C applications.

Table 16 on page 65 shows all of the built-in functions. In the listing of library functions, each built-in function is labelled as such.

Table 16. Built-in Library Functions

abs()	alloca()	cds()	cs()	decabs()
decchk()	decfix()	fabs()	fortrc()	memchr()
memcmp()	memcpy()	memset()	strcat()	strchr()
strcmp()	strcpy()	strlen()	strncat()	strncmp()
strncpy()	strrchr()	tsched()		

The built-in versions of these functions are accessed by preprocessor macros defined in the standard header files. They are not used unless the appropriate header file (such as `decimal.h`, `math.h`, `stdlib.h`, or `string.h`) is included in the source file.

Your program will use the built-in version of a standard function only if you include the associated standard header file. However, `decfix()`, `decabs()`, and `decchk()` are implemented only as built-in functions. They are not available without including the header file.

If you are using the standard header file, but want to use the function in the dynamic library instead of the built-in function, you can force a call to the dynamic library by putting parentheses around the function name in your source code: `(memcpy)(buf1, buf2, len)`

If you will never use the built-in version, you can also use `#undef` with the function name. For example, `#undef memcpy` causes all calls to `memcpy` in the compilation unit to make a dynamic call to the function rather than using the built-in version.

## IEEE Floating Point

Starting with Version 2 Release 6, OS/390 (including the Language Environment and C/C++ components) has added support for IEEE binary floating-point (IEEE floating-point) as defined by the ANSI/IEEE Standard 754-1985, IEEE Standard for Binary Floating-Point Arithmetic.

### Notes:

1. You must have OS/390 Release 6 or higher to use the IEEE floating-point instructions. In Release 6, the base control program (BCP) is enhanced to support the new IEEE floating-point hardware in the IBM S/390 Generation 5 Server. This enables programs running on OS/390 Release 6 to use the IEEE floating-point instructions and 16 floating-point registers. In addition, the BCP provides simulation support for all the new floating-point hardware instructions. This enables applications that make light use of IEEE floating-point, and can tolerate the overhead of software simulation, to execute on OS/390 Release 6 without requiring an IBM S/390 Generation 5 Server.
2. The terms binary floating-point and IEEE floating-point are used interchangeably. The abbreviations BFP and HFP, which are used in some function names, refer to binary floating-point and S/390 hexadecimal floating-point (hexadecimal floating-point), respectively.

The C/C++ compiler provides a `FL0AT` option to select the format of floating-point numbers produced in a compile unit. The `FL0AT` option allows you to select either IEEE floating-point or hexadecimal floating-point format. For details on the OS/390 C/C++ support, see the description of the `FL0AT` option in the *OS/390 C/C++ User's Guide*. In addition, two related sub-options have been introduced, `ARCH(3)` and

TUNE(3). The two sub-options support the new G5 processor architecture, and IEEE binary floating-point data. Refer to the ARCHITECTURE and TUNE compiler options in the *OS/390 C/C++ User's Guide* for details.

The C/C++ run-time library interfaces, which formerly supported only hexadecimal floating-point format, have been changed in OS/390 Version 2 Release 6 to support both IEEE floating-point and hexadecimal floating-point formats. These interfaces are documented in the OS/390 Run-Time Library Reference.

The primary documentation for the IEEE floating-point support is contained in the Enterprise Systems Architecture/390 Principles of Operation and the OS/390 C/C++ User's Guide.

IEEE floating-point is provided on S/390 primarily to enhance interoperability and portability between S/390 and other platforms. It is anticipated that IEEE floating-point will be most commonly used for new and ported applications, and in emerging environments, such as Java. Customers should not migrate existing applications that use hexadecimal floating-point to IEEE floating-point, unless there is a specific reason (such as a need to interoperate with a non-S/390 platform).

IBM does not recommend mixing floating-point formats in an application. However, for applications which must handle both formats, the C/C++ run-time library does offer some support. Reference information for IEEE floating-point can also be found in OS/390 C/C++ Language Reference.

---

## External Variables

The POSIX 1003.1 and X/Open CAE Specification 4.2 (XPG4.2) require that the C system header files define certain external variables. Additional variables are defined for use with POSIX or XPG4.2 functions. If you define one of the POSIX or XPG4 feature test macros and include one of these headers, the external variables will be defined in your program. These external variables are treated differently than other global variables in a multithreaded environment (values are thread-specific) and across a call to a fetched module (values are propagated). To access the global variable values (not thread specific), either C with the RENT compiler option or C++ must be used, and the SCEEOBJ autocall library must be specified during the OS/390 bind. Functions to access the thread-specific values of these variables are provided for use when running in a multithreaded environment.

### errno

When a run-time library function fails, the function may do any of the following to identify the error:

- Set errno to a documented value.
- Set errno to a value that is not documented. You can use `strerror()` or `perror()` to get the message associated with the errno.
- Not set errno.
- Clear errno.

See also **errno.h**.



## daylight

Daylight savings time flag set by `tzset()`. Note that other time zone sensitive functions such as `ctime()`, `localtime()`, `mktime()`, and `strftime()` implicitly call `tzset()`. Use the `__dlight()` function to access the thread-specific value of daylight. See also **time.h**.

## getdate\_err

The variable is set to the value below when an error occurs in the `getdate()` function.

- 1 The DATEMASK environment variable is null or undefined.
- 2 The template file cannot be opened for reading.
- 3 Failed to get file status information.
- 4 The template file is not a regular file.
- 5 An error is encountered while reading the template file.
- 6 Memory allocation failed.
- 7 There is no line in the template that matches the input.
- 8 Invalid input specification.

Any changes to `errno` are unspecified. Use the `__gderr()` function to access the thread-specific value of `getdate_err`. See also **time.h**.

## h\_errno

An integer which holds the specific error code when the network nameserver encounters an error. The network nameserver is used by the `gethostbyname()` and `gethostbyaddr()` functions. Use the `__h_errno()` function to access the thread-specific value of `h_errno`. See also **netdb.h**.

## \_\_loc1

A global character pointer which is set by the `regex()` function to point to the first matched character in the input string. Use the `____loc1()` function to access the thread-specific value of `__loc1`. See also **libgen.h**.

## loc1

A pointer to characters matched by regular expressions used by `step()`. The value is not propagated across a call to a fetched module. See also **regexp.h**.

## loc2

A pointer to characters matched by regular expressions used by `step()`. The value is not propagated across a call to a fetched module. See also **regexp.h**.

## locs

Used by `advance()` to stop regular expression matching in a string. The value is not propagated across a call to a fetched module. See also **regexp.h**.

### optarg

Character pointer used by `getopt()` for options parsing variables. Use the `__opargf()` function to access the thread-specific value of `optarg`. See also **stdio.h** and **unistd.h**.

### opterr

Error value used by `getopt()`. Use the `__operrf()` function to access the thread-specific value of `opterr`. See also **stdio.h** and **unistd.h**.

### optind

Integer pointer used by `getopt()` for options parsing variables. Use the `__opindf()` function to access the thread-specific value of `optind`. See also **stdio.h** and **unistd.h**.

### optopt

Integer pointer used by `getopt()` for options parsing variables. Use the `__opoptf()` function to access the thread-specific value of `optopt`. See also **stdio.h** and **unistd.h**.

### signgam

Storage for sign of `lgamma()`. This function defaults to thread specific. See also **math.h**.

### stdin

Standard Input stream. The external variable will be initialized to point to the enclave-level stream pointer for the standard input file. There is no multithreaded function. See also **stdio.h**.

### stderr

Standard Error stream. The external variable will be initialized to point to the enclave-level stream pointer for the standard error file. There is no multithreaded function. See also **stdio.h**.

### stdout

Standard Output stream. The external variable will be initialized to point to the enclave-level stream pointer for the standard output file. There is no multithreaded function. See also **stdio.h**.

### t\_errno

An integer which holds the specific error code when a failure occurs in one of the X/Open Transport Interface (XTI) functions. Use the `__t_errno()` function to access the thread-specific value of `t_errno`. See also **xti.h**.

## timezone

Long integer difference from UTC and standard time as set by `tzset()`. Note that other time zone sensitive functions such as `ctime()`, `localtime()`, `mktime()`, and `strftime()` implicitly call `tzset()`. Use the `__tzzone()` function to access the thread-specific value of `timezone`. See also **time.h**.

## tzname

Character pointer to unsized array of timezone strings used by `tzset()` and `ctime()`. The `*tzname` variable contains the Standard and Daylight Savings time zone names. If the TZ environment variable is present and correct, `tzname` will be set from TZ. Otherwise `tzname` will be set from the LC\_TOD locale category. See the `tzset()` function for a description. There is no multithreaded function. See also **time.h**.

---

### Functions A-O in the OS/390 C/C++ Library

This section lists all the OS/390 C/C++ Run-Time Library functions, in alphabetical order from A-O.

## abort() — Stop a Program

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
void abort(void);
```

### General Description

Causes an abnormal program termination and returns control to the host environment. The `abort()` function flushes all buffers and closes all open files. Be aware that abnormal termination will *not* run the `atexit()` list functions.

If the `abort()` function is called and the user has a handler for `SIGABRT`, then `SIGABRT` is raised; however, `SIGABRT` is raised again when the handler associated with the default action is returned. The code path only passes through the user's handler once, even if the handler is reset. The same thing occurs if `SIGABRT` is ignored; abnormal termination occurs.

The `abort()` function will not result in program termination if `SIGABRT` is caught by a signal handler, and the signal handler does not return. You can avoid returning by “jumping” out of the handler with `setjmp()` and `longjmp()`. In OS/390 C programs, you can jump out of the handler with `sigsetjmp()` and `siglongjmp()`.

For more information see the process termination sections in the chapter “Using Runtime User Exits” in the *OS/390 C/C++ Programming Guide*.

### Special Behavior for POSIX C Programs

To obtain access to the special POSIX behavior for `abort()`, the POSIX runtime option must be set ON, and the version of MVS must be 4.3 or higher.

Calls to `abort()` raise the `SIGABRT` signal, via `pthread_kill()`, so that the signal is directed to the same thread. A `SIGABRT` signal generated by `abort()` cannot be blocked.

Under POSIX, the handler can use `siglongjmp()` to restore the environment at a place in the code where a `sigsetjmp()` was previously issued. In this way, an application can avoid the process for termination. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information about using POSIX support.

### Special Behavior for C++

If `abort()` is called from a OS/390 C++ program, the program will be terminated immediately, without leaving the current block. Functions passed to `atexit()`, and destructors for static and local (automatic) objects, will not be called.

By default, the OS/390 C++ function `terminate()` calls `abort()`.

### Returned Value

The abnormal termination return code for MVS is 2000.

### Example

#### CBC3BA01

```
/* CBC3BA01
   This example tests for successful opening of the file myfile.
   If an error occurs, an error message is printed and the program ends
   with a call to the abort() function.
*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *stream;

    if ((stream = fopen("myfile.dat", "r")) == NULL)
    {
        printf("Could not open data file\n");
        abort();

        printf("Should not see this message\n");
    }
}
```

### Related Information

- “`stdlib.h`” on page 45
- “`assert()` — Verify Condition” on page 111
- “`atexit()` — Register Program Termination Function” on page 116
- “`exit()` — End Program” on page 330
- “`pthread_kill()` — Send a Signal to a Thread” on page 984
- “`raise()` — Raise Signal” on page 1074
- “`signal()` — Handle Interrupts” on page 1330

## abs() — Calculate Integer Absolute Value

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
int abs(int n);
```

### General Description

The built-in function `abs()` returns the absolute value of an integer argument *n*.

The minimum allowable integer is `INT_MIN+1`. (`INT_MIN` is a macro that is defined in the `limits.h` header file.) For example, with the OS/390 C/C++ compiler, `INT_MIN+1` is `-2147483648`. If the value entered cannot be represented as an integer, the `abs()` function returns the same value.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

The returned value is the absolute value, if the absolute value is possible to represent. Otherwise the input value is returned. There are no return values defined to indicate an error.

### Example

#### CBC3BA02

```
/* CBC3BA02
   This example calculates the absolute value of an integer x and assigns
   it to y.
*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int x = -4, y;

    y = abs(x);
    printf("The absolute value of %d is %d.\n", x, y);
}
```

### Output

The absolute value of    -4 is    4.

**Related Information**

- “stdlib.h” on page 45
- “fabs() — Calculate Floating-Point Absolute Value” on page 338
- “labs() — Calculate Long Absolute Value” on page 733



# accept() — Accept a New Connection on a Socket

## Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

## Format

### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int accept(int socket, struct sockaddr *address, size_t *address_len);
```

### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>

int accept(int socket, struct sockaddr *address, int *address_len);
```

## General Description

The `accept()` call is used by a server to accept a connection request from a client. When a connection is available, the socket created is ready for use to read data from the process that requested the connection. The call accepts the first connection on its queue of pending connections for the given socket *socket*. The `accept()` call creates a new socket descriptor with the same properties as *socket* and returns it to the caller. If the queue has no pending connection requests, `accept()` blocks the caller unless *socket* is in nonblocking mode. If no connection requests are queued and *socket* is in nonblocking mode, `accept()` returns `-1` and sets the error code to `EWOULDBLOCK`. The new socket descriptor cannot be used to accept new connections. The original socket, *socket*, remains available to accept more connection requests.

Parameter	Description
<i>socket</i>	The socket descriptor.
<i>address</i>	The socket address of the connecting client that is filled in by <code>accept()</code> before it returns. The format of <i>address</i> is determined by the domain that the client resides in. This parameter can be <code>NULL</code> if the caller is not interested in the client address.
<i>address_len</i>	Must initially point to an integer that contains the size in bytes of the storage pointed to by <i>address</i> . On return, that integer contains the size of the data returned in the storage pointed to by <i>address</i> . If <i>address</i> is <code>NULL</code> , <i>address_len</i> is ignored.

The *socket* parameter is a stream socket descriptor created with the `socket()` call. It is usually bound to an address with the `bind()` call. The `listen()` call marks the socket as one that accepts connections and allocates a queue to hold pending connection requests. The `listen()` call places an upper boundary on the size of the queue.

The *address* parameter is a pointer to a buffer into which the connection requester's address is placed. The *address* parameter is optional and can be set to be the NULL pointer. If set to NULL, the requester's address is not copied into the buffer. The exact format of *address* depends on the addressing domain from which the communication request originated. For example, if the connection request originated in the AF\_INET domain, *address* points to a **sockaddr\_in** structure as defined in the include file **netinet.in.h**. The *address\_len* parameter is used only if *name* is not NULL. Before calling `accept()`, you must set the integer pointed to by *address\_len* to the size of the buffer, in bytes, pointed to by *address*. On successful return, the integer pointed to by *address\_len* contains the actual number of bytes copied into the buffer. If the buffer is not large enough to hold the address, up to *address\_len* bytes of the requester's address are copied. If the actual length of the address is greater than the length of the supplied **sockaddr**, the stored address is truncated. The *sa\_len* member of the store structure contains the length of the untruncated address.

**Note:** This call is used only with SOCK\_STREAM sockets. There is no way to screen requesters without calling `accept()`. The application cannot tell the system the requesters from which it will accept connections. However, the caller can choose to close a connection immediately after discovering the identity of the requester.

A socket can be checked for incoming connection requests using the `select()` call.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

A nonnegative socket descriptor indicates success; the value `-1` indicates an error. The value of the error code indicates the specific error.

Error Code	Description
EAGAIN	If during an <code>accept</code> call that changes identity, the UID of the new identity is already at MAXPROCUID, the <code>accept</code> call fails.
EBADF	The <i>socket</i> parameter is not within the acceptable range for a socket descriptor.
EFAULT	Using <i>address</i> and <i>address_len</i> would result in an attempt to copy the address into a portion of the caller's address space into which information cannot be written.
EINTR	A signal interrupted the <code>accept()</code> call before any connections were available.
EINVAL	<code>listen()</code> was not called for socket descriptor <i>socket</i> .
EIO	There has been a network or transport failure.
EMFILE	An attempt was made to open more than the maximum number of file descriptors allowed for this process.
EMVSERR	Two consecutive <code>accept</code> calls that cause an identity change are not allowed. The original identity must be restored ( <code>close()</code> the socket that caused the identity change) before any further accepts are allowed to change the identity

ENFILE	The maximum number of file descriptors in the system are already open.
ENOBUFS	Insufficient buffer space is available to create the new socket.
ENOTSOCK	The <i>socket</i> parameter does not refer to a valid socket descriptor.
EOPNOTSUPP	The socket type of the specified socket does not support accepting connections.
EWOULDBLOCK	The socket descriptor <i>socket</i> is in nonblocking mode, and no connections are in the queue.

### Example

The following are two examples of the `accept()` call. In the first, the caller wishes to have the requester's address returned. In the second, the caller does not wish to have the requester's address returned.

```
int clientsocket;
int s;
struct sockaddr clientaddress;
int address_len;
int accept(int s, struct sockaddr *addr, int *address_len);
/* socket(), bind(), and listen() have been called */
/* EXAMPLE 1: I want the address now */
address_len = sizeof(clientaddress);
clientsocket = accept(s, &clientaddress, &address_len);
/* EXAMPLE 2: I can get the address later using getpeername() */
clientsocket = accept(s, (struct sockaddr *) 0, (int *) 0);
```

### Related Information

- “bind() — Bind a Name to a Socket” on page 128
- “connect() — Connect a Socket” on page 214
- “getpeername() — Get the Name of the Peer Connected to a Socket” on page 568
- “listen() — Prepare the Server for Incoming Client Requests” on page 752
- “socket() — Create a Socket” on page 1371

## accept\_and\_recv() — Accept Connection and Receive First Message

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R6

### Format

#### X/Open

```
#define _OPEN_SYS_SOCKET_EXT2
#include <sys/socket.h>

int accept_and_recv(int socket, int *accept_socket,
                   struct sockaddr *remote_address,
                   socklen_t *remote_address_len,
                   struct sockaddr *local_address,
                   socklen_t *local_address_len,
                   void *buffer, size_t length);
```

### General Description

The `accept_and_recv()` function extracts the first connection on the queue of pending connections. It either reuses the specified socket (if *accept\_socket* is not -1) or creates a new socket with the same socket type, protocol, and address family as the listening socket (if *accept\_socket* is -1). It then returns the first block of data sent by the peer and returns the local and remote socket addresses associated with the connection.

The function takes the following arguments:

#### Parameter Description

*socket* Specifies a socket that was created with `socket()`, has been bound to an address with `bind()`, and has issued a successful call to `listen()`.

#### *accept\_socket*

Pointer to an `int` which specifies the socket on which to accept the incoming connection. The socket must not be bound or connected. Use of this parameter lets the application reuse the accepting socket. It is possible that the system may choose to reuse a different socket than the one the application specified by this argument. In this case, the system will set *\*accept\_socket* to the socket actually reused.

A value of -1 for *\*accept\_socket* indicates that the accepting socket should be assigned by the system and returned to the application via this parameter. It is recommended that a value of -1 be used on the first call to `accept_and_recv()`. For more details, see “Application Usage” on page 79.

#### *remote\_address*

Either a NULL pointer or a pointer to a `sockaddr` structure where the address of the connecting socket will be returned.

#### *remote\_address\_len*

Points to a `socklen_t` item. On input, this item specifies the length of the supplied `sockaddr` structure. On output, this item contains the length of the stored address.

*local\_address*

Either a NULL pointer or a pointer to a `sockaddr` structure where the address of the local socket will be returned.

*local\_address\_len*

Points to a `socklen_t` item. On input, this item specifies the length of the supplied `sockaddr` structure. On output, this item contains the length of the stored address.

*buffer*

Either a NULL pointer, or a pointer to a buffer where the message should be stored. If this is a NULL pointer, no receive is performed, and `accept_and_recv()` completes when the incoming connection is received.

*length*

Specifies the length in bytes of the buffer pointed to by the *buffer* argument.

If *\*accept\_socket* is not -1, the incoming connection will be accepted on the socket specified by *\*accept\_socket*. The system may choose to reuse a different socket. If it does, the system will change *\*accept\_socket* to reflect the socket actually used.

If *remote\_address* is not a NULL pointer, the address of the peer for the accepted connection is stored in the `sockaddr` structure pointed to by *remote\_address*, and the length of this address is stored in the object pointed to by *remote\_address\_len*. If *local\_address* is not a NULL pointer, the address of the local socket associated with this connection is stored in the `sockaddr` structure pointed to by *local\_address*, and the length of this address is stored in the object pointed to by *local\_address\_len*.

If the actual length of the address is greater than the length of the supplied `sockaddr` structure, the stored address will be truncated.

Non-blocking mode is not supported for this function. If `O_NONBLOCK` is set on the socket file descriptor, the function will return with -1 and `errno` will be set to `EOPNOTSUPP`. If the listen queue is empty of connection requests and `O_NONBLOCK` is not set on the socket file descriptor, `accept_and_recv()` will block and will not return until an incoming connection is received. In addition, if *buffer* is not NULL, `accept_and_recv` will not return until the first block of data on the connection has been received.

## Application Usage

On the first call to `accept_and_recv()`, it is recommended that the application set the socket pointed to by *accept\_socket* to -1. This will cause the system to assign the accepting socket. The application then passes the assigned value into the next call to `accept_and_recv()` (by setting *accept\_socket* = *socket\_ptr*).

To take full advantage of the performance improvements offered by the `accept_and_recv()` function, a process/thread model different from the one where a parent accepts in a loop and spins off child process threads is needed. The parent/process thread is eliminated. Multiple worker processes/threads are created, and each worker process/thread then executes the `accept_and_recv()` function in a loop. The performance benefits of `accept_and_recv()` include fewer buffer copies, recycled sockets, and optimal scheduling.

## Returned Value

Upon successful completion, `accept_and_recv()` returns the number of bytes actually stored in the buffer pointed to by the *buffer* argument.

If unsuccessful, `accept_and_recv()` returns -1 and sets `errno` to one of the following:

Error Code	Description
EBADF	One of two errors occurred: <ol style="list-style-type: none"> <li>1. The <i>socket</i> argument is not a valid descriptor.</li> <li>2. <i>accept_socket</i> does not point to a valid descriptor.</li> </ol>
ECONNABORTED	A connection has been aborted.
ECONNRESET	A connection was forcibly closed by a peer.
EFAULT	The data buffer pointed to by <i>accept_socket</i> , <i>remote_address</i> , <i>remote_address_len</i> , <i>local_address</i> , <i>local_address_len</i> , or <i>buffer</i> was not valid.
EISCONN	The <i>accept_socket</i> is either bound or connected already.
EINTR	The <code>accept_and_recv()</code> function was interrupted by a signal that was caught before a valid connection arrived.
EINTRNODATA	The <code>accept_and_recv()</code> function was interrupted by a signal that was caught after a valid connection arrived, but before the first block of data arrived.
EINVAL	The <i>socket</i> is not accepting connections.
EIO	An I/O error occurred.
EMFILE	OPEN_MAX descriptors are already open in the calling process.
ENFILE	The maximum number of descriptors in the system are already open.
ENOBUFS	No buffer space is available.
ENOMEM	There was insufficient memory available to complete the operation.
ENOREUSE	Socket reuse is not supported.
ENOSR	There were insufficient STREAMS resources available for the operation to complete.
ENOTSOCK	The <i>socket</i> argument does not refer to a socket, or <i>accept_socket</i> does not point to a socket.
EOPNOTSUPP	One of three errors occurred: <ol style="list-style-type: none"> <li>1. The socket type of the specified socket does not support accepting connections.</li> <li>2. 0_NONBLOCK is set for this socket and non-blocking is not supported for this function.</li> <li>3. The <code>accept_and_recv()</code> function is not supported by this platform.</li> </ol>
EPROTO	A protocol error has occurred.

**Related Information**

- “sys/socket.h” on page 48
- “accept() — Accept a New Connection on a Socket” on page 75
- “read() — Read From a File or Socket” on page 1080
- “getpeername() — Get the Name of the Peer Connected to a Socket” on page 568
- “getsockname() — Get the Name of a Socket” on page 604

## access() — Determine Whether a File Can be Accessed

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
int access(const char *pathname, int how);
```

### General Description

Determines how an HFS file can be accessed. When checking to see if a process has appropriate permissions, access() looks at the *real* user ID (UID) and group ID (GID), not the effective IDs.

*pathname* is the name of the file whose accessibility you want to test. The *how* argument indicates the access modes you want to test. The following symbols are defined in the unistd.h header file for use in the *how* argument:

F_OK	Tests whether the file exists.
R_OK	Tests whether the file can be accessed for reading.
W_OK	Tests whether the file can be accessed for writing.
X_OK	Tests whether the file can be accessed for execution.

You can take the bitwise inclusive OR of any or all of the last three symbols to test several access modes at once. If you are using F\_OK to test for the file's existence, you cannot use OR with any of the other symbols.

To provide an ASCII input/output format for applications using this function, define feature test macro \_\_LIBASCII as described on page 22.

### Returned Value

The returned value of access() is zero if the specified access is permitted. If the given file cannot be accessed in the specified way, access() returns -1.

When unsuccessful, access() sets errno to one of the following:

EACCES	The process does not have appropriate permissions to access the file in the specified way, or does not have search permission on some component of the <i>pathname</i> prefix.
EINVAL	The value of <i>how</i> is incorrect.
ELOOP	A loop exists in the symbolic links.
ENAMETOOLONG	<i>pathname</i> is longer than PATH_MAX characters. The PATH_MAX value is determined using pathconf().



ENOENT	There is no file named <i>pathname</i> , or the <i>pathname</i> argument is an empty string.
ENOTDIR	Some component of the <i>pathname</i> prefix is not a directory.
EROFS	The argument <i>how</i> has specified write access for a file on a read-only file system.

## Returned Value for POSIX C Programs

The following `errno` values behave differently when a program is running with `POSIX(ON)`:

ELOOP	A loop exists in the symbolic links. This error is issued if the number of symbolic links detected in the resolution is greater than <code>POSIX_SYMLINK_MAX</code> (a value defined in the <code>limits.h</code> header file).
ENAMETOOLONG	<i>pathname</i> is longer than <code>PATH_MAX</code> characters, or some component of <i>pathname</i> is longer than <code>NAME_MAX</code> , when <code>_POSIX_NO_TRUNC</code> (defined in the <code>unistd.h</code> header file) is in effect. The <code>PATH_MAX</code> and <code>NAME_MAX</code> values are determined using <code>pathconf()</code> .

See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information about using POSIX support.

## Example CBC3BA03

```
/* CBC3BA03
   The following example determines how a file is accessed.
*/
#define _POSIX_SOURCE
#include <unistd.h>
#include <stdio.h>

main()
{
    char path[] = "/";

    if (access(path, F_OK) != 0)
        printf("%s' does not exist!\n", path);
    else {
        if (access(path, R_OK) == 0)
            printf("You have read access to '%s'\n", path);
        if (access(path, W_OK) == 0)
            printf("You have write access to '%s'\n", path);
        if (access(path, X_OK) == 0)
            printf("You have search access to '%s'\n", path);
    }
}
```

## Output

From a non-superuser:

```
You have read access to '/'
You have search access to '/'
```

### **Related Information**

- “limits.h” on page 32
- “unistd.h” on page 53
- “chmod() — Change the Mode of a File or Directory” on page 174
- “stat() — Get File Information” on page 1404

## acos() — Calculate Arccosine

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double acos(double x);
```

### General Description

Calculates the arccosine of  $x$ , expressed in radians, in the range 0 to  $\pi$ .

The value of  $x$  must be between  $-1$  and  $1$  inclusive.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

#### Special Behavior for C/370

If  $x$  is less than  $-1$  or greater than  $1$ , `acos()` sets `errno` to `EDOM` and returns  $0$ . If the correct value would cause underflow, zero is returned and the value of the macro `ERANGE` is stored in `errno`.

#### Special Behavior for XPG4.2

Upon successful completion, the `acos()` function returns the arc cosine of  $x$ , in the range  $[0, \pi]$  radians.

The `acos()` function will fail if:

**EDOM** The value  $x$  is not in the range  $[-1, 1]$ .

No other errors will occur.

### Example

#### CBC3BA04

```
/* CBC3BA04
   This example prompts for a value for x. It prints an error message if
   x is greater than 1 or less than -1; otherwise, it assigns the
   arccosine of x to y.
*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAX 1.0
#define MIN -1.0
```

```

int main(void)
{
    double x, y;

    printf( "Enter x\n" );
    scanf( "%lf", &x );

    /* Output error if not in range */
    if ( x > MAX )
        printf( "Error: %f too large for acos\n", x );
    else if ( x < MIN )
        printf( "Error: %f too small for acos\n", x );
    else {
        y = acos( x );
        printf( "acos( %f ) = %f\n", x, y );
    }
}

```

**Output**

Expected result if 0.4 is entered:

```

Enter x
acos( 0.400000 ) = 1.159279

```

**Related Information**

- “math.h” on page 35
- “acosh() — Hyperbolic Arccosine” on page 87
- “asin() — Calculate Arcsine” on page 108
- “asinh() — Hyperbolic Arcsine” on page 110
- “atan() - atan2() — Calculate Arctangent” on page 113
- “atanh() — Hyperbolic Arctangent” on page 115
- “cos() — Calculate Cosine” on page 229
- “cosh() — Calculate Hyperbolic Cosine” on page 231
- “sin() — Calculate Sine” on page 1360
- “sinh() — Calculate Hyperbolic Sine” on page 1362
- “tan() — Calculate Tangent” on page 1500
- “tanh() — Calculate Hyperbolic Tangent” on page 1502

## acosh() — Hyperbolic Arccosine

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>
```

```
double acosh(double x);
```

### General Description

The `acosh()` function returns the hyperbolic arccosine of its argument `x`.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

If successful, `acosh()` returns the hyperbolic arccosine of its argument `x`.

If the value of `x` is less than 1.0, then the function returns 0.0 and sets `errno` to `EDOM`.

### Related Information

- “`math.h`” on page 35
- “`acos()` — Calculate Arccosine” on page 85
- “`asin()` — Calculate Arcsine” on page 108
- “`asinh()` — Hyperbolic Arcsine” on page 110
- “`atan()` - `atan2()` — Calculate Arctangent” on page 113
- “`atanh()` — Hyperbolic Arctangent” on page 115
- “`cos()` — Calculate Cosine” on page 229
- “`cosh()` — Calculate Hyperbolic Cosine” on page 231
- “`sin()` — Calculate Sine” on page 1360
- “`sinh()` — Calculate Hyperbolic Sine” on page 1362
- “`tan()` — Calculate Tangent” on page 1500
- “`tanh()` — Calculate Hyperbolic Tangent” on page 1502

## advance() — Pattern Match Given a Compiled Regular Expression

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <regex.h>
```

```
int advance(const char *string, const char *expbuf);
```

```
extern char *loc2, *locs;
```

### General Description

The `advance()` function attempts to match an input string of characters with the compiled regular expression which was obtained by an earlier call to `compile()`.

The first parameter *string* is a pointer to a string of characters to be checked for a match.

*expbuf* is the pointer to the regular expression which was previously obtained by a call to `compile()`.

The external variable *loc2* will point to the next character in *string* after the last character that matched the regular expression.

The external variable *locs* can be optionally set to point to some point in the input regular expression string to cause the `advance()` function to exit its back up loop.

**Note:** The external variables *cirf*, *sed*, and *nbra* are reserved.

During the pattern matching operation, when `advance()` encounters a `*` or `\{ }` sequence in the regular expression, it will advance its pointer to the string to be matched as far as possible and will recursively call itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, `advance()` will back up along the string until it finds a match or reaches the point in the string that initially matched the `*` or `\{ }`. It is sometime desirable to stop this backing up before the initial point in the string is reached. If the external character pointer *locs* is equal to the point in the string at some time during the back up process, `advance()` will break out of the loop that backs up and will return 0 (a failure indication).

### Notes:

1. The application must provide the proper serialization for the `compile()`, `step()`, and `advance()` functions if they are run under a multi-threaded environment.
2. The `compile()`, `step()`, and `advance()` functions are provided for historical reasons. New applications should use the new functions `fnmatch()`, `glob()`, `regcomp()` and `regexec()`, which provide full internationalized regular expression functionality compatible with ISO POSIX.2 standard.

## Returned Value

The `advance()` function returns nonzero if the initial substring of *string* matches the regular expression in *expbuf*. It returns 0 if there is no match.

If there is a match, the `advance()` function sets an external character pointer, *loc2*, as a side effect. The variable *loc2* points to the next character in *string* after the last character that matched the regular expression.

## Related Information

- “`regexp.h`” on page 40
- “`compile()` — Compile Regular Expression” on page 208
- “`fnmatch()` — Match Filename or Pathname” on page 415
- “`glob()` — Generate Pathnames Matching a Pattern” on page 636
- “`regcomp()` — Compile Regular Expression” on page 1120
- “`regexexec()` — Execute Compiled Regular Expression” on page 1126
- “`step()` — Pattern Match with Regular Expression” on page 1411

aio\_cancel() — Cancel an Asynchronous I/O Request

Standards

Standards/Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 UNIX

Format

```

| #define _XOPEN_SOURCE 500
| #include <aio.h>
|
| int aio_cancel(int fildes, struct aiocb *aiocbp);

```

General Description

The `aio_cancel()` function attempts to cancel one or more asynchronous I/O requests currently outstanding against file descriptor *fildes*. The *aiocbp* argument points to an *aiocb* structure for a particular request to be canceled or is NULL to cancel all outstanding cancelable requests against *fildes*.

Normal asynchronous notification occurs for asynchronous I/O operations that are successfully canceled. The associated error status is set to `ECANCELED` and the return status is set to `-1` for the canceled requests.

For requests that cannot be canceled, the normal asynchronous completion process takes place when their I/O completes. In this case the *aiocb* is not modified by `aio_cancel()`.

An asynchronous operation is cancelable if it is currently blocked or becomes blocked. Once an outstanding request can be completed it is allowed to complete. For example, an `aio_read()` will be cancelable if there is no data available at the time that `aio_cancel()` is called.

*Fildes* must be a valid file descriptor, but when *aiocbp* is not NULL *fildes* does not have to match the file descriptor with which the asynchronous operation was initiated. For maximum portability, though, it should match.

The `aio_cancel()` function always waits for the request being canceled to either complete or be canceled. When control returns from `aio_cancel()`, the program may safely free the original request's *aiocb* and buffer. If a signal was specified on the original request, the signal handler for that request's I/O complete notification may run before, during, or after control returns from `aio_cancel()`, so coordination may be necessary between the signal handler and the caller of `aio_cancel()`. This is particularly unpredictable when `aio_cancel()` is called from a different thread than the original request, unless the original thread no longer exists.

Canceling all requests on a given descriptor does not stop new requests from being made or otherwise effect the descriptor. The program may start again or close the descriptor depending on why it issued the cancel.

An individual request can only be canceled once. Subsequent attempts to explicitly cancel the same request will fail with `EALREADY`.



## Returned Value

The `aio_cancel()` function returns one of the following values:

- `AIO_CANCELED` if the requested operations were canceled.
- `AIO_NOTCANCELED` if at least one of the requested operations cannot be canceled because it is in progress.

In this case, the state of the other operations, if any, referenced in the call to `aio_cancel()` is not indicated by the return value of `aio_cancel()`. The application can determine the status of these operations by using `aio_error()`.

- `AIO_ALLDONE` if all of the operations have already completed. This is returned when there are no outstanding requests found that match the criteria specified. This is also the result returned when a file associated with *filides* does not support the asynchronous I/O function because there are no outstanding requests to be found that match the criteria specified.
- `-1` if there was an error. `Errno` is set to indicate the error.

The `aio_cancel()` function will fail if:

Errno Code	Description
<code>EBADF</code>	The <i>filides</i> argument is not a valid file descriptor.
<code>EALREADY</code>	The operation to be canceled is already being canceled.

## Related Information

- “aio.h” on page 23
- “aio\_read() — Asynchronous Read from a Socket” on page 93
- “aio\_write() — Asynchronous Write to a Socket” on page 99
- “aio\_return() — Retrieve Status for an Asynchronous I/O Operation” on page 96
- “aio\_error() — Retrieve Error Status for an Asynchronous I/O Operation” on page 92

## aio\_error() — Retrieve Error Status for an Asynchronous I/O Operation

### Standards

Standards/Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 UNIX

### Format

```
#define _XOPEN_SOURCE 500
#include <aio.h>

int aio_error(const struct aiocb *aiocbp);
```

### General Description

The `aio_error()` function returns the error status associated with the `aiocb` structure referenced by the `aiocbp` argument. The error status for an asynchronous I/O operation is the `errno` value that would be set by the corresponding `read()`, or `write()` operation. If the operation has not yet completed, then the error status will be equal to `EINPROGRESS`.

### Returned Value

If the asynchronous I/O operation has completed successfully, then 0 is returned. If the asynchronous I/O operation has completed unsuccessfully, then the error status as described for `read()`, or `write()` is returned. If the asynchronous I/O operation has not yet completed, then `EINPROGRESS` is returned.

The `aio_error()` function does not set `errno`.

When the `errno` is returned is not `EINPROGRESS` and not zero, the `errno2` set by either `read()` or `write()` can be retrieved by using the `__errno2()` function.

### Related Information

- “`aio.h`” on page 23
- “`aio_return()` — Retrieve Status for an Asynchronous I/O Operation” on page 96
- “`aio_suspend()` — Wait for an Asynchronous I/O Request” on page 97
- “`aio_read()` — Asynchronous Read from a Socket” on page 93

## aio\_read() — Asynchronous Read from a Socket

### Standards

Standards/Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 UNIX

### Format

```
#define _XOPEN_SOURCE 500
#include <aio.h>

int aio_read(struct aiocb *aiocbp);
```

### General Description

The `aio_read()` function initiates an asynchronous read operation as described by the `aiocb` structure (the asynchronous I/O control block).

The `aiocbp` argument points to the `aiocb` structure. This structure contains the following members:

<code>aio_fildes</code>	file descriptor
<code>aio_offset</code>	file offset
<code>aio_buf</code>	location of buffer
<code>aio_nbytes</code>	length of transfer
<code>aio_reqprio</code>	request priority offset
<code>aio_sigevent</code>	signal number and value
<code>aio_lio_opcode</code>	operation to be performed

The operation reads up to `aio_nbytes` from the socket or file associated with `aio_fildes` into the buffer pointed to by `aio_buf`. The call to `aio_read()` returns when the request has been initiated or queued to the file or device (even when the data cannot be delivered immediately).

Asynchronous I/O is currently only supported for sockets. The `aio_offset` field may be set but it will be ignored.

With a stream socket an asynchronous read may be completed when the first packet of data arrives and the application may have to issue additional reads, either asynchronously or synchronously, to get all the data it wants. A datagram socket has message boundaries and the operation will not complete until an entire message has arrived.

The `aiocbp` value may be used as an argument to `aio_error()` and `aio_return()` functions in order to determine the error status and return status, respectively, of the asynchronous operation. While the operation is proceeding, the error status retrieved by `aio_error()` is `EINPROGRESS`; the return status retrieved by `aio_return()` however is unpredictable.

If an error condition is encountered during the queuing, the function call returns without having initiated or queued the request.

When the operation completes asynchronously the program can be notified by a signal as specified in the *aio\_sigevent* structure. At this time the return and error status will have been updated to reflect the outcome of the operation. The sigevent structure's value and notification function fields are not supported. If a signal is not desired the program can occasionally poll the aiocb with *aio\_error()* until the result is no longer *EINPROGRESS*.

Be aware that the operation may complete, and the signal handler may run, before control returns from the call to *aio\_read()*. Even when the operation does complete this quickly the return value from the call to *aio\_read()* will be zero, reflecting the queueing of the I/O request not the results of the I/O itself.

An asynchronous operation may be canceled with *aio\_cancel()* prior to its completion. Canceled operations complete with an error status of *ECANCELED* and any specified signal will be delivered. Due to timing, the operation may still complete naturally, either successfully or unsuccessfully, before it can be canceled by *aio\_cancel()*.

If the file descriptor of this operation is closed, the operation will be deleted if it has not completed or is not just about to complete. Signals specified for deleted operations will not be delivered. *Close()* will wait for asynchronous operations in progress for the descriptor to be deleted or completed.

You may use *aio\_suspend()* to wait for the completion of asynchronous operations.

Sockets must be in blocking state or the operation may fail with *EWouldBlock*.

If the control block pointed by *aiocbp* or the buffer pointed to by *aio\_buf* becomes an illegal address prior to the asynchronous I/O completion, then the behavior of *aio\_read()* is unpredictable.

If the thread that makes the *aio\_read()* request terminates before the I/O completes the *aiocb* structure will still be updated with the return and error status, and any specified signal will be delivered to the process in which the thread was running. If thread related storage was used on the request the results are quite unpredictable.

Simultaneous asynchronous operations using the same *aiocbp*, asynchronous operations using an invalid *aiocbp*, or any system action, that changes the process memory space while asynchronous I/O is outstanding to that address range, will produce unpredictable results

The *aio\_lio\_opcode* field is set to *LIO\_READ* by the function *aio\_read()*.

*\_POSIX-PRIORITIZED\_IO* is not supported. The *aio\_reqprio* field may be set but it will be ignored.

*\_POSIX-SYNCHRONIZED\_IO* is not supported.

## Returned Value

The *aio\_read()* function returns the value of zero to the calling process if the I/O operation is successfully queued; otherwise, the function returns the value *-1* and sets *errno* to indicate the error. The *aio\_read()* function will fail if:

Errno Code	Description
ENOSYS	The file associated with <i>aio_fildes</i> does not support the <i>aio_read()</i> function.
EAGAIN	The requested asynchronous I/O operation was not queued due to system resource limitations.

Each of the following conditions may be detected synchronously at the time of the call to *aio\_read()*, or asynchronously. If any of the conditions below are detected synchronously, the *aio\_read()* function returns `-1` and sets the *errno* to the corresponding value. If any of the conditions below are detected asynchronously, the return status of the asynchronous operation is set to `-1`, and the error status of the asynchronous operation will be set to the corresponding value.

Errno Code	Description
EBADF	The <i>aio_fildes</i> argument is not a valid file descriptor open for reading.
EWOULDBLOCK	The file associated with <i>aio_fildes</i> is in non-blocking state and there is no data available.
EINVAL	<i>aio_sigevent</i> contains an invalid value.

In the case where the *aio\_read()* function successfully queues the I/O operation but the operation is subsequently canceled or encounters an error, the return status of the asynchronous operations is set to `-1`, and the error status of the asynchronous operation will be set to the error status normally set by the *read()* function call, or to the following value:

Errno Code	Description
ECANCELED	The requested I/O was canceled before the I/O completed due to an explicit call to <i>aio_cancel()</i> .

### Related Information

- “*aio.h*” on page 23
- “*aio\_write()* — Asynchronous Write to a Socket” on page 99
- “*aio\_cancel()* — Cancel an Asynchronous I/O Request” on page 90
- “*aio\_error()* — Retrieve Error Status for an Asynchronous I/O Operation” on page 92
- “*aio\_return()* — Retrieve Status for an Asynchronous I/O Operation” on page 96
- “*aio\_suspend()* — Wait for an Asynchronous I/O Request” on page 97

## aio\_return() — Retrieve Status for an Asynchronous I/O Operation

### Standards

Standards/Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 UNIX

### Format

```
#define _XOPEN_SOURCE 500
#include <aio.h>

int aio_return(const struct aiocb *aiocbp);
```

### General Description

The `aio_return()` function returns the return status associated with the `aiocb` structure referenced by the `aiocbp` argument. The return status for an asynchronous I/O operation is the value that would be set by the corresponding `read()` or `write()` operation. While the operation is proceeding, the error status retrieved by `aio_error()` is `EINPROGRESS`; the return status retrieved by `aio_return()` however is unpredictable. The `aio_return()` function may be called to retrieve the return status of a given asynchronous operation; once `aio_error()` has returned with 0.

### Returned Value

If the asynchronous I/O operation has completed successfully, then the return status as described for `read()` or `write()` is returned. If the asynchronous I/O operation has not yet completed, then the return status is unpredictable.

The `aio_return()` does not set `errno`.

### Related Information

- “`aio.h`” on page 23
- “`aio_read()` — Asynchronous Read from a Socket” on page 93
- “`aio_suspend()` — Wait for an Asynchronous I/O Request” on page 97
- “`aio_error()` — Retrieve Error Status for an Asynchronous I/O Operation” on page 92

## aio\_suspend() — Wait for an Asynchronous I/O Request

### Standards

Standards/Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 UNIX

### Format

```
#define _XOPEN_SOURCE 500
#include <aio.h>

int aio_suspend(const struct aiocb * const list[],
               int nent, const struct timespec * timeout);
```

### General Description

The `aio_suspend()` function suspends the calling thread when the *timeout* is a null pointer until at least one of the asynchronous I/O operations referenced by the *list* argument has completed, or until a signal interrupts the function. Or, if *timeout* is not null, it is suspended until the time interval specified by *timeout* has passed. If the time interval indicated in the *timespec* structure pointed to by *timeout* passes before any of the I/O operations referenced by *list*, then `aio_suspend()` returns with an error. If any of the *aiocb* structures in the list correspond to completed asynchronous I/O operations (that is, the error status for the operation is not equal to `EINPROGRESS`) at the time of the call, the function returns without suspending the calling thread.

The *list* argument is an array of pointers to asynchronous I/O control blocks (AIOCBs). The *nent* argument indicates the number of elements in the array. Each *aiocb* structure pointed to will have been used in initiating an asynchronous I/O request. This array may contain null pointers, which are ignored. If this array contains pointers that refer to *aiocb* structures that have not been used in submitting asynchronous I/O or *aiocb* structures that are not valid, the results are unpredictable.

### Returned Value

If the `aio_suspend()` function returns after one or more asynchronous I/O operation have completed, the function returns zero. Otherwise, the function returns a value of `-1` and sets `errno` to indicate the error. The application may determine which asynchronous I/O completed by scanning the associated error and return status using `aio_error()` or `aio_return()`, respectively. The value of `errno` indicates the specific error.

Errno Code	Description
EAGAIN	No asynchronous I/O indicated in the list referenced by <i>list</i> completed in the time interval indicated by <i>timeout</i> .
ENOSYS	OS/390 UNIX System Services does not support the <code>aio_suspend</code> function.
EINTR	A signal interrupted the <code>aio_suspend()</code> function. Note that, since each asynchronous I/O operation may possibly provoke a signal when it completes, this error return may be caused by the completion of one (or more) of the very I/O operations being awaited.

### Usage Notes:

1. The AIOCBs represented by the list of AIOCB pointers must reside in the same storage key as the key of the invoker of aio\_suspend. If the AIOCB Pointer List or any of the AIOCBs represented in the list are not accessible by the invoker an EFAULT may occur.
2. AIOCB pointers in the list with a value of zero will be ignored.
3. A timeout value of zero (seconds+nanoseconds) means that the aio\_suspend() call will not wait at all. It will check for any completed asynchronous I/O requests. If none are found it will return with a EAGAIN. If at least one is found aio\_suspend() will return with success.
4. A timeout value of a *timespec* with the tv\_sec field set with INT\_MAX, as defined in <limits.h>, will cause the aio\_suspend service to wait until a asynchronous I/O request completes or a signal is received.

If the Macro \_AIO\_OS390 is defined then the following may also apply.

5. The number of pointers to AIOCBs that use application supplied event control block (ECB) pointers for invocations of asynchronous I/O is limited to 253. There is no limit when not using the \_AIO\_OS390 Feature Test Macro. See *OS/390 UNIX System Services Programming: Assembler Callable Services Reference* under the BPX1AIO for information on supplying user defined ECBs in the AIOCB data area.
6. The AIOCBs passed to aio\_suspend() must not be freed or reused by other threads in the process while this service is still in progress. This service may use the AIOCBs even after the asynchronous I/O completes. This restriction excludes multiple threads from doing aio\_suspend() on the same AIOCB at the same time. Modifying the AIOCB during an aio\_suspend() will produce unpredictable results.
7. The use of these extensions will require macros from SYS1.CSSLIB. Make sure that it is included in the SYSLIB concatenation during the compile.

### Related Information

- “aio.h” on page 23
- “aio\_return() — Retrieve Status for an Asynchronous I/O Operation” on page 96
- “aio\_read() — Asynchronous Read from a Socket” on page 93
- “aio\_error() — Retrieve Error Status for an Asynchronous I/O Operation” on page 92



## aio\_write() — Asynchronous Write to a Socket

### Standards

Standards/Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 UNIX

### Format

```
#define _XOPEN_SOURCE 500
#include <aio.h>

int aio_write(struct aiocb *aiocbp);
```

### General Description

The `aio_write()` function initiates an asynchronous write as described by the *aiocb* structure (the asynchronous I/O control block).

The *aiocbp* argument points to the *aiocb* structure. This structure contains the following members:

<code>aio_fildes</code>	file descriptor
<code>aio_offset</code>	file offset
<code>aio_buf</code>	location of buffer
<code>aio_nbytes</code>	length of transfer
<code>aio_reqprio</code>	request priority offset
<code>aio_sigevent</code>	signal number and value
<code>aio_lio_opcode</code>	operation to be performed

The operation will write *aio\_nbytes* from the buffer pointed to by *aio\_buf* to the socket or file associated with *aio\_fildes*. The call to `aio_write()` returns when the request has been initiated or queued to the file or device (even when the data cannot be delivered immediately).

Asynchronous I/O is currently only supported for sockets. The *aio\_offset* field may be set but it will be ignored.

The *aiocbp* value may be used as an argument to `aio_error()` and `aio_return()` functions in order to determine the error status and return status, respectively, of the asynchronous operation. While the operation is proceeding, the error status retrieved by `aio_error()` is `EINPROGRESS`; the return status retrieved by `aio_return()` however is unpredictable.

If an error condition is encountered during the queueing, the function call returns without having initiated or queued the request.

When the operation completes asynchronously the program can be notified by a signal as specified in the *aio\_sigevent* structure. At this time the return and error status will have been updated to reflect the outcome of the operation. The *sigevent* structure's value and notification function fields are not supported. If a signal is not desired the program can occasionally poll the *aiocb* with `aio_error()` until the result is no longer `EINPROGRESS`.

Be aware that the operation may complete, and the signal handler may run, before control returns from the call to `aio_read()`. Even when the operation does complete this quickly the return value from the call to `aio_read()` will be zero, reflecting the queueing of the I/O request not the results of the I/O itself.

An asynchronous operation may be canceled with `aio_cancel()` prior to its completion. Canceled operations complete with an error status of `ECANCELED` and any specified signal will be delivered. Due to timing, the operation may still complete naturally, either successfully or unsuccessfully, before it can be canceled by `aio_cancel()`.

If the file descriptor of this operation is closed, the operation will be deleted if it has not completed or is not just about to complete. Signals specified for deleted operations will not be delivered. `Close()` will wait for asynchronous operations in progress for the descriptor to be deleted or completed.

You may use `aio_suspend()` to wait for the completion of asynchronous operations.

Sockets must be in blocking state or the operation may fail with `EWouldBlock`.

If the control block pointed by *aioctx* or the buffer pointed to by *aio\_buf* becomes an illegal address prior to the asynchronous I/O completion, then the behavior of `aio_read()` is unpredictable

If the thread that makes the `aio_read()` request terminates before the I/O completes, the *aioctx* structure will still be updated with the return and error status, and any specified signal will be delivered to the process in which the thread was running. If thread related storage was used on the request the results are quite unpredictable.

Simultaneous asynchronous operations using the same *aioctx*, attempting asynchronous operations using an invalid *aioctx*, or any system action, that changes the process memory space while asynchronous I/O is outstanding to that address range, will produce unpredictable results.

The *aio\_lio\_opcode* field is set to `LIO_WRITE` by the function `aio_write()`.

`_POSIX_PRIORITIZED_IO` is not supported. The *aio\_reqprio* field may be set but it will be ignored.

`_POSIX_SYNCHRONIZED_IO` is not supported.

## Returned Value

The `aio_write()` function returns the value of zero to the calling process if the I/O operation is successfully queued; otherwise, the function returns the value `-1` and sets `errno` to indicate the error. The `aio_write()` function will fail if:

Errno Code	Description
<code>ENOSYS</code>	The file associated with <i>aio_fildes</i> does not support the <code>aio_write()</code> function.
<code>EAGAIN</code>	The requested asynchronous I/O operation was not queued due to system resource limitations.

Each of the following conditions may be detected synchronously at the time of the call to `aio_write()`, or asynchronously. If any of the conditions below are detected synchronously, the `aio_write()` function returns `-1` and sets the `errno` to the corresponding value. If any of the conditions below are detected asynchronously, the return status of the asynchronous operation is set to `-1`, and the error status of the asynchronous operation will be set to the corresponding value.

Errno Code	Description
EBADF	The <i>aio_fildes</i> argument is not a valid file descriptor open for writing.
EWOULDBLOCK	The file associated with <i>aio_fildes</i> is in non-blocking state and there is no data available.
EINVAL	The <i>aio_nbytes</i> is not a valid value or <i>aio_sigevent</i> contains a value that is not valid.

In the case where `aio_write()` successfully queues the I/O operation but the operation is subsequently canceled or encounters an error, the return status of the asynchronous operation is set to `-1`, and the error status of the asynchronous operation is set to the error status normally set by the `write()` function call, or to the following value:

Errno Code	Description
ECANCELED	The requested I/O was canceled before the I/O completed due to an explicit call to <i>aio_cancel()</i> .

### Related Information

- “aio.h” on page 23
- “aio\_read() — Asynchronous Read from a Socket” on page 93
- “aio\_cancel() — Cancel an Asynchronous I/O Request” on page 90
- “aio\_error() — Retrieve Error Status for an Asynchronous I/O Operation” on page 92
- “aio\_return() — Retrieve Status for an Asynchronous I/O Operation” on page 96
- “aio\_suspend() — Wait for an Asynchronous I/O Request” on page 97

## alarm() — Set an Alarm

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	POSIX(ON) MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
unsigned int alarm(unsigned int seconds);
```

### General Description

Generates a SIGALRM signal after the number of seconds specified by the *seconds* parameter has elapsed. The SIGALRM signal delivery is directed at the calling thread.

*seconds* is the number of real seconds to wait before the SIGALRM signal is generated. Because of processor delays, the SIGALRM signal may be generated slightly later than this specified time. If *seconds* is zero, any previously set alarm request is canceled.

Only one such alarm can be active at a time. If you set a new alarm time, any previous alarm is canceled.

This function is supported only in a POSIX program. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information.

### Special Behavior for XPG4

The fork() function clears pending alarms in the child thread. However, a new thread image created by one of the exec functions inherits the time left to an alarm in the old thread's image.

### Special Behavior for XPG4.2

alarm() will interact with the setitimer() function when the setitimer() function is used to set the ‘real’ interval timer (ITIMER\_REAL).

alarm() does not interact with the usleep() function.

### Returned Value

If a prior alarm request has not yet completed, alarm() returns the number of seconds remaining until that request would have generated a SIGALRM signal.

If there are no prior alarm requests with time remaining, alarm() returns zero. Because alarm() is always successful, so there is no failure return. If any failures are encountered that prevent alarm() from completing successfully, an abend is generated.

## Example

### CBC3BA05

```

/* CBC3BA05
   The following example generates a SIGALRM signal.
   */
#define _POSIX_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
#include <time.h>

volatile int footprint=0;

void catcher(int signum) {
    puts("inside signal catcher!");
    footprint = 1;
}

main()
{
    struct sigaction sact;
    volatile double count;
    time_t t;

    sigemptyset(&sact.sa_mask);
    sact.sa_flags = 0;
    sact.sa_handler = catcher;
    sigaction(SIGALRM, &sact, NULL);

    alarm(5); /* timer will pop in five seconds */

    time(&t);
    printf("before loop, time is %s", ctime(&t));
    for (count=0; (count<1e10) && (footprint == 0); count++);
    time(&t);
    printf("after loop, time is %s", ctime(&t));

    printf("the sum so far is %.0f\n", count);

    if (footprint == 0)
        puts("the signal catcher never gained control");
    else
        puts("the signal catcher gained control");
}

```

## Output

```

before loop, time is Fri Jun 16 08:37:03 1995
inside signal catcher!
after loop, time is Fri Jun 16 08:37:08 1995
the sum so far is 17417558

```

## Related Information

- “signal.h” on page 41
- “unistd.h” on page 53
- “exec Functions” on page 322
- “fork() — Create a New Process” on page 422
- “pause() — Suspend a Process Pending a Signal” on page 899
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “sleep() — Suspend Execution of a Thread” on page 1364

## alarm

- “signal() — Handle Interrupts” on page 1330
- “setitimer() — Set Value of an Interval Timer” on page 1232
- “usleep() — Suspend Execution for an Interval” on page 1663

## alloca() — Allocate Storage from the Stack

### Standards

Standards / Extensions	C or C++	Dependencies
	both	

### Format

```
#include <stdlib.h>
```

```
void *alloca(unsigned int size);
```

### General Description

The built-in `alloca()` function obtains memory from the stack. This eliminates the need for an explicit `free()` as the memory is freed when the stack is collapsed.

In the event of an ***out of memory*** condition, `alloca()` will trap the condition and give an out of memory message.

To avoid infringing on the user's name-space, this non-standard function is exposed only when you use the compiler option, `LANGVLV(EXTENDED)`. When you use `LANGVLV(EXTENDED)` any relevant information in the header is also exposed.

### Returned Value

The returned value is the address of the requested storage.

### Related Information

- “`stdlib.h`” on page 45
- “Built-in Functions” on page 64

asctime() — Convert Time to Character String

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

Format

```
#include <time.h>
```

```
char *asctime(const struct tm *timeptr);
```

General Description

Converts time stored as a structure, pointed to by *timeptr*, to a character string. The *timeptr* value can be obtained from a call to *gmtime()* or *localtime()*. Both functions return a pointer to a *tm* structure defined in *time.h*.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

The string result that *asctime()* produces contains exactly 26 characters and has the format:

```
"%.3s %.3s%3d %.2d:%.2d:%.2d %d\n"
```

The following is an example of the string returned:

```
Fri Jun 16 02:03:55 1995\n\0
```

Notes:

- This function is sensitive to time zone information which is provided by:
  - The TZ environmental variable when POSIX(ON) and TZ is correctly defined, or by the \_TZ environmental variable when POSIX(OFF) and \_TZ is correctly defined.
  - The LC\_TOD category of the current locale if POSIX(OFF) or TZ is not defined.

The time zone external variables *tzname*, *timezone*, and *daylight* declarations remain feature test protected in *time.h*.

- The calendar time returned by a call to the *time()* function begins at epoch, which was at 00:00:00 Coordinated Universal Time (UTC), January 1, 1970.
- The *asctime()* function uses a 24-hour clock format.
- The days are abbreviated to: Sun, Mon, Tue, Wed, Thu, Fri, and Sat.
- The months are abbreviated to: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, and Dec.
- All fields have a constant width.
- Dates with only one digit are preceded either with a zero or a blank space.
- The new-line character (`\n`) and the null character (`\0`) occupy the last two positions of the string.



- The `asctime()`, `ctime()`, and other time functions can use a common, statically allocated buffer for holding the return string. Each call to one of these functions may possibly destroy the result of the previous call.

## Returned Value

Returns a pointer to the resulting character string. No specific value is returned if an error occurs.

## Example CBC3BA06

```
/* CBC3BA06
   This example polls the system clock and prints a message giving the
   current time.
*/
#include <time.h>
#include <stdio.h>

int main(void)
{
    struct tm *newtime;
    time_t ltime;

    /* Get the time in seconds */
    time(&ltime);
    /* Break it down and store it in the structure tm */
    newtime = localtime(&ltime);

    /* Print the local time as a string */
    printf("The current date and time are %s",
           asctime(newtime));
}
```

## Output

The current date and time are Fri Jun 16 13:29:51 1995

## Related Information

- “time.h” on page 51
- “ctime() — Convert Time to a Character String” on page 250
- “gmtime() — Convert Time to Broken-Down UTC Time” on page 640
- “localdtconv() — Date/Time Formatting Convention Inquiry” on page 754
- “localtime() — Convert Time and Correct for Local Time” on page 758
- “mktime() — Convert Local Time” on page 828
- “strftime() — Convert to Formatted Time” on page 1432
- “time() — Determine Current Time” on page 1570
- “tzset() — Set the Time Zone” on page 1637

## asin() — Calculate Arcsine

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double asin(double x);
```

### General Description

Calculates the arcsine of  $x$ , in the range  $-\pi/2$  to  $\pi/2$ .

The value of  $x$  must be between  $-1$  and  $1$ .

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

If  $x$  is less than  $-1$  or greater than  $1$ , `asin()` sets `errno` to `EDOM`, and returns a value of  $0$ . Otherwise, `asin()` returns a nonzero value. If the correct value would cause an underflow,  $0$  is returned and the value of the macro `ERANGE` is stored in `errno`.

### Example CBC3BA07

```
/* CBC3BA07
   This example prompts for a value for x. It prints an error message if
   x is greater than 1 or less than -1; otherwise, it assigns the arcsine
   of x to y.
*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAX 1.0
#define MIN -1.0

int main(void)
{
    double x, y;

    printf( "Enter x\n" );
    scanf( "%lf", &x );

    /* Output error if not in range */
    if ( x > MAX )
        printf( "Error: %f too large for asin\n", x );
    else if ( x < MIN )
        printf( "Error: %f too small for asin\n", x );
    else {
        y = asin( x );
        printf( "asin( %f ) = %f\n", x, y );
    }
}
```

```
    }
}
```

### Output

Enter x

0.2 is entered

asin( 0.200000 ) = 0.201358

### Related Information

- “math.h” on page 35
- “acos() — Calculate Arccosine” on page 85
- “acosh() — Hyperbolic Arccosine” on page 87
- “asinh() — Hyperbolic Arcsine” on page 110
- “atan() - atan2() — Calculate Arctangent” on page 113
- “atanh() — Hyperbolic Arctangent” on page 115
- “cos() — Calculate Cosine” on page 229
- “cosh() — Calculate Hyperbolic Cosine” on page 231
- “sin() — Calculate Sine” on page 1360
- “sinh() — Calculate Hyperbolic Sine” on page 1362
- “tan() — Calculate Tangent” on page 1500
- “tanh() — Calculate Hyperbolic Tangent” on page 1502

## asinh() — Hyperbolic Arcsine

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>
```

```
double asinh(double x);
```

### General Description

The `asinh()` function returns the hyperbolic arcsine of its argument  $x$ .

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

`asinh()` returns the hyperbolic arcsine of its argument  $x$ . The function is always successful.

### Related Information

- “`math.h`” on page 35
- “`acos()` — Calculate Arccosine” on page 85
- “`acosh()` — Hyperbolic Arccosine” on page 87
- “`asin()` — Calculate Arcsine” on page 108
- “`atan()` - `atan2()` — Calculate Arctangent” on page 113
- “`atanh()` — Hyperbolic Arctangent” on page 115
- “`cos()` — Calculate Cosine” on page 229
- “`cosh()` — Calculate Hyperbolic Cosine” on page 231
- “`sin()` — Calculate Sine” on page 1360
- “`sinh()` — Calculate Hyperbolic Sine” on page 1362
- “`tan()` — Calculate Tangent” on page 1500
- “`tanh()` — Calculate Hyperbolic Tangent” on page 1502

## assert() — Verify Condition

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <assert.h>
```

```
void assert(int expression);
```

### General Description

The `assert()` macro inserts diagnostics into a program. If the expression is false (zero), a diagnostic message of the form shown below is printed to `stderr`, and `abort()` is called to abnormally end the program. The `assert()` macro takes no action if the expression is true (nonzero).

The diagnostic message has the format:

Assertion failed: *expression*, file *filename*, line *line-number*.

If you define `NDEBUG` to any value with a `#define` directive or with the `DEFINE` compiler option, the C/C++ preprocessor expands all `assert()` invocations to void expressions.

**Note:** The `assert()` function is a macro. Using the `#undef` directive with the `assert()` macro results in undefined behavior. The `assert()` macro uses `__FILE__` and `__LINE__`.

### Returned Value

There is no returned value.

### Example

#### CBC3BA08

```
/* CBC3BA08
   In this example, the assert() macro tests the string argument for a
   null string and an empty string, and verifies that the length argument
   is positive before proceeding.
*/
#include <stdio.h>
#include <assert.h>

void analyze(char *, int);
int main(void)
{
    char *string1 = "ABC";
    char *string2 = "";
    int length = 3;

    analyze(string1, length);
    printf("string1 %s is not null or empty, "
           "and has length %d \n", string1, length);
    analyze(string2, length);
```

## assert

```
printf("string2 %s is not null or empty,"  
      "and has length %d\n", string2, length);  
}  
  
void analyze(char *string, int length)  
{  
    assert(string != NULL);           /* cannot be NULL */  
    assert(*string != '\0');         /* cannot be empty */  
    assert(length > 0);              /* must be positive */  
}
```

### Output

String1 ABC is not null or empty, and has length 3  
Assertion failed: \*string != '\0', file: CBC3BA08 C

A1, line: 26

### Related Information

- “assert.h” on page 23
- “abort() — Stop a Program” on page 71

## atan() - atan2() — Calculate Arctangent

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double atan(double x);
double atan2(double y, double x);
```

### General Description

The `atan()` and `atan2()` functions calculate the arctangent of  $x$  and  $y/x$ , respectively.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

Returns a value in the range  $-\pi/2$  to  $\pi/2$  radians. The `atan2()` function returns a value in the range  $-\pi$  to  $\pi$  radians. If both arguments of `atan2()` are zero, the function sets `errno` to `EDOM`, and returns a value of zero. If the correct value would cause underflow, zero is returned and the value of the macro `ERANGE` is stored in `errno`.

### Example

#### CBC3BA09

```
/* CBC3BA09 */
#include <math.h>
#include <stdio.h>

int main(void)
{
    double a,b,c,d;

    c = 0.45;
    d = 0.23;

    a = atan(c);
    b = atan2(c,d);

    printf("atan( %f ) = %f\n", c, a);
    printf("atan2( %f, %f ) = %f\n", c, d, b);
}
```

### Output

```
atan( 0.450000 ) = 0.422854
atan2( 0.450000, 0.230000 ) = 1.098299
```

## Related Information

- “math.h” on page 35
- “acos() — Calculate Arccosine” on page 85
- “acosh() — Hyperbolic Arccosine” on page 87
- “asin() — Calculate Arcsine” on page 108
- “asinh() — Hyperbolic Arcsine” on page 110
- “atanh() — Hyperbolic Arctangent” on page 115
- “cos() — Calculate Cosine” on page 229
- “cosh() — Calculate Hyperbolic Cosine” on page 231
- “sin() — Calculate Sine” on page 1360
- “sinh() — Calculate Hyperbolic Sine” on page 1362
- “tan() — Calculate Tangent” on page 1500
- “tanh() — Calculate Hyperbolic Tangent” on page 1502



## atanh() — Hyperbolic Arctangent

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>
```

```
double atanh(double x);
```

### General Description

The `atanh()` function returns the hyperbolic arctangent of its argument  $x$ .

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

If successful, `atanh()` returns the hyperbolic arctangent of its argument  $x$ .

`atanh()` fails, returns 0.0 and sets `errno` to the following `errno`s under the conditions described:

EDOM        The  $x$  argument has a value greater than 1.0.  
ERANGE      The  $x$  argument has a value equal to 1.0.

### Related Information

- “`math.h`” on page 35
- “`acos()` — Calculate Arccosine” on page 85
- “`acosh()` — Hyperbolic Arccosine” on page 87
- “`asin()` — Calculate Arcsine” on page 108
- “`asinh()` — Hyperbolic Arcsine” on page 110
- “`atan()` - `atan2()` — Calculate Arctangent” on page 113
- “`cos()` — Calculate Cosine” on page 229
- “`cosh()` — Calculate Hyperbolic Cosine” on page 231
- “`sin()` — Calculate Sine” on page 1360
- “`sinh()` — Calculate Hyperbolic Sine” on page 1362
- “`tan()` — Calculate Tangent” on page 1500
- “`tanh()` — Calculate Hyperbolic Tangent” on page 1502

## atexit() — Register Program Termination Function

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
int atexit(void (*func)(void));
```

### General Description

Records a function, pointed to by *func*, that the system calls at normal program termination. Termination is a result of `exit()` or returning from `main()`, regardless of the language of the `main()` routine. Process termination started by `_exit()` or by a terminating signal under Language Environment is not included.

The functions are executed in the reverse order that they were registered. The registered function must return to ensure that all registered functions are called. The functions registered with `atexit()` are started before streams and files are closed. You can use `atexit()` to register up to 32 functions.

Under OS/390 UNIX services only, when a process ends, the address space is ended; otherwise, the address space persists.

### Special Behavior for OS/390 C

The C/MVS `atexit()` function has the following restrictions:

- Any function registered by a fetched module that has been released is removed from the list at the time of `release()`. See `fetch()`, `fetchep()`, and `release()` for details about fetching and releasing modules.
- Any function registered in an explicitly loaded DLL (using `dllload()`) that has been freed (using `dllfree()`) is removed from the list. But, if the DLL in question has also been implicitly loaded, then the function is NOT removed from the `atexit` list.
- All C/MVS library functions can be used in a registered routine except `atexit()` and `exit()`.
- When a program is running under CICS control, if an `EXEC CICS RETURN` command or an `EXEC CICS XCTL` command is issued, the `atexit()` list that has been previously registered is not run.
- Use of the `system()` library function within `atexit()` may result in undefined behavior.
- Use of non-C subroutines or functions in the `atexit()` list will result in undefined behavior.
- The `atexit()` list will not be run when `abort()` is called.

### Special Behavior for C++

- All of the behaviors listed under “Special Behavior for OS/390 C”.
- Because C and C++ linkage conventions are incompatible, atexit() cannot receive C++ function pointers. If you attempt to pass a C++ function pointer to atexit(), the compiler will flag it as an error. To use the C++ atexit() function, you must ensure that all functions registered for atexit() have C linkage by declaring them as extern “C”.
- You can use try, throw, and catch in a function registered for atexit(). However, by the time an atexit() function is driven, all stack frames will have collapsed. As a result, the only catch clauses available for throw will be the ones coded in the atexit() function. If those catch clauses cannot handle the thrown object, terminate() will be called.

### Special Behavior for XPG4.2

The maximum number of functions that can be registered is specified by the symbol ATEXIT\_MAX which is defined in the limits.h header.

### Returned Value

Returns zero if it is successful, and nonzero if it fails.

### Example

#### CBC3BA10

```
/* CBC3BA10
   This example uses the atexit() function to call the function goodbye()
   at program termination.
*/
#include <stdlib.h>
#include <stdio.h>

#ifdef __cplusplus          /* the __cplusplus macro is      */
extern "C" void goodbye(void); /* automatically defined by the */
#else                      /* C++/MVS compiler      */
void goodbye(void);
#endif

int main(void)
{
    int rc;

    rc = atexit(goodbye);
    if (rc != 0)
        printf("Error in atexit");
    exit(0);
}

void goodbye(void)
{
    /* This function is called at normal program termination */
    printf("The function goodbye was called \
at program termination\n");
}
```

### Output

The function goodbye was called at program termination

## **Related Information**

- “Condition Handling” in the *IBM Language Environment Programming Guide*
- “stdlib.h” on page 45
- “abort() — Stop a Program” on page 71
- “exit() — End Program” on page 330
- “fetch() — Get a Load Module” on page 369
- “fetchep() — Share Writable Static” on page 382
- “release() — Delete a Load Module” on page 1130
- “signal() — Handle Interrupts” on page 1330

## \_\_atoe() — ISO8859-1 to EBCDIC String Conversion

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#include <unistd.h>
```

```
int __atoe(char *string);
```

### General Description

The `__atoe()` function converts an ISO8859-1 character string *string* to its ebcdic equivalent. The conversion is performed using the codeset page associated with the current locale. The input character string up to, but not including, the null is changed from an ISO8859-1 representation to that of the current locale.

The argument, *string*, points to the ISO8859-1 character string to be converted to its ebcdic equivalent.

### Returned Value

If successful, `__atoe()` converts the input ISO8859-1 character string to its equivalent ebcdic value, and returns the length of the converted string.

If unsuccessful, `__atoe()` returns `-1`, and stores the error value in `errno`. The following are the possible values of `errno`:

`EINVAL`      The current locale does not describe a single-byte character set.  
`ENOMEM`     There is insufficient storage to complete the conversion process.

**Note:** This function may internally call `iconv_open()` and `iconv()`. The `errno`s returned by these functions are propagated without modification.

### Related Information

- “unistd.h” on page 53

## \_\_atoe\_l() — ISO8859-1 to EBCDIC Conversion Operation

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#include <unistd.h>
```

```
int __atoe_l(char * bufferptr, int leng);
```

### General Description

The `__atoe_l()` function converts *leng* ISO8859-1 bytes in the buffer pointed to by *bufferptr* to their ebcdic equivalent. The conversion is performed using the codeset page associated with the current locale.

The argument, *bufferptr*, points to a buffer containing the ISO8859-1 bytes to be converted to their ebcdic equivalent. The input buffer is treated as sequence of bytes, and all bytes in the input buffer are converted, including any imbedded nulls.

### Returned Value

If successful, `__atoe_l()` converts the input IOS8859-1 bytes to their equivalent ebcdic value, and returns the number of bytes that were converted.

If unsuccessful, `__atoe_l()` returns `-1`, and returns the error value in `errno`. The following are the possible values of `errno`:

`EINVAL`      The current locale does not describe a single-byte character set.  
`ENOMEM`     There is insufficient storage to complete the conversion process.

**Note:** This function may internally call `iconv_open()` and `iconv()`. The `errno`s returned by these functions are propagated without modification.

### Related Information

- “unistd.h” on page 53

## atof() — Convert Character String to Double

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
double atof(const char *nptr);
```

### General Description

The `atof()` function converts the initial portion of the string pointed to by `nptr` to a 'double'. This is equivalent to

```
strtod(nptr, NULL)
```

The double value is IBM S/390 hexadecimal floating-point (hexadecimal floating-point) or IEEE binary floating-point (IEEE floating-point) format depending on the floating-point mode of the thread invoking the `atof()` function. This function uses `__isBFP()` to determine the floating-point mode of the invoking thread.

See the “*fscanf* Family of Formatted Input Functions” on page 467 for a description of special infinity and NaN sequences recognized by OS/390 formatted input functions, including `atof()` and `strtod()` in IEEE floating-point mode.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

There are no documented `errno`s for this function.

### Related Information

- “`stdlib.h`” on page 45
- “`atoi()` — Convert Character String to Integer” on page 122
- “`atol()` — Convert Character String to Long” on page 123
- “`fscanf()` – `scanf()` – `sscanf()` — Read and Format Data” on page 464
- “`__isBFP()` — Determine Application Floating-Point Format” on page 705
- “`strtod()` — Convert Character String to Double” on page 1456
- “`strtol()` — Convert Character String to Long” on page 1460
- “`strtoul()` — Convert String to Unsigned Integer” on page 1462

## atoi() — Convert Character String to Integer

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
int atoi(const char *nptr);
```

### General Description

The `atoi()` function converts the initial portion of the string pointed to by *nptr* to a 'int'. This is equivalent to

```
(int)strtol(nptr, NULL, 10)
```

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

There are no documented errors for this function.

### Related Information

- “`stdlib.h`” on page 45
- “`atof()` — Convert Character String to Double” on page 121
- “`atol()` — Convert Character String to Long” on page 123
- “`fscanf()` – `scanf()` – `sscanf()` — Read and Format Data” on page 464
- “`strtod()` — Convert Character String to Double” on page 1456
- “`strtol()` — Convert Character String to Long” on page 1460
- “`strtoul()` — Convert String to Unsigned Integer” on page 1462



## atol() — Convert Character String to Long

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
long int atol(const char *nptr);
```

### General Description

The `atol()` function converts the initial portion of the string pointed to by *nptr* to a 'long int'. This is equivalent to

```
strtol(nptr, NULL, 10)
```

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

There are no documented errors for this function.

### Related Information

- “`stdlib.h`” on page 45
- “`atof()` — Convert Character String to Double” on page 121
- “`atoi()` — Convert Character String to Integer” on page 122
- “`fscanf()` – `scanf()` – `sscanf()` — Read and Format Data” on page 464
- “`strtod()` — Convert Character String to Double” on page 1456
- “`strtol()` — Convert Character String to Long” on page 1460
- “`strtoul()` — Convert String to Unsigned Integer” on page 1462

## a64l() — Convert Base-64 String Representation to Long Integ

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>
```

```
long a64l (const char *string);
```

### General Description

The `a64l()` function converts a string representation of a base-64 number into its corresponding long value. It scans the string from left to right with the least significant character on the left, decoding each character as a 6-bit base-64 number. If the string pointed to by *string* contains more than six characters, `a64l()` uses only the first six. If the first six characters of the string contain a null character, `a64l()` uses only the characters preceding the first null. The following characters are used to represent digits:

#### *Character*    *Digit represented*

.	0
/	1
0-9	2-11
A-Z	12-37
a-z	38-63

### Returned Value

On successful completion, the `a64l()` function returns the long value resulting from conversion of the input string. If the string pointed to by *string* is null, `a64l()` returns zero.

There are no `errno` values defined for `a64l()`.

### Related Information

- “`stdlib.h`” on page 45
- “`strtoul()` — Convert String to Unsigned Integer” on page 1462
- “`l64a()` — Convert Long to Base-64 String Representation” on page 782

## basename() — Return the Last Component of a Pathname

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <libgen.h>
```

```
char *basename(char *path);
```

### General Description

The `basename()` function takes the pathname pointed to by *path* and returns a pointer to the final component of the pathname, deleting any trailing '/' characters.

If the string consists entirely of the '/' character, `basename()` returns a pointer to the string "/".

If *path* is a null pointer or points to an empty string, `basename()` returns a pointer to the string ".". The `basename()` function may modify the string pointed to by *path*.

Examples:

Input String	Output String
"/usr/lib"	"lib"
"/usr/"	"usr"
"/"	"/"

### Returned Value

The `basename()` function returns a pointer to the final component of *path*.

There are no `errno` values defined for `basename()`.

### Related Information

- "libgen.h" on page 31
- "dirname() — Report the Parent Directory of a Pathname" on page 275

## bcmp() — Compare Bytes in Memory

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <strings.h>
```

```
int bcmp(const void *s1, const void *s2, size_t n);
```

### General Description

The `bcmp()` function compares the first  $n$  bytes of the area pointed to by `s1` with the area pointed to by `s2`.

### Returned Value

The `bcmp()` function returns 0 if `s1` and `s2` are identical and nonzero otherwise. Both areas are assumed to be at least  $n$  bytes long.

If the value of  $n$  is zero, `bcmp()` returns 0.

There are no `errno` values defined for `bcmp()`.

### Related Information

- “strings.h” on page 46
- “bcopy() — Copy Bytes in Memory” on page 127
- “bzero() — Zero Out Bytes in Memory” on page 140
- “memcmp() — Compare Bytes” on page 811

## bcopy() — Copy Bytes in Memory

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <strings.h>
```

```
void bcopy(const void *s1, void *s2, size_t n);
```

### General Description

The `bcopy()` function copies *n* bytes from the area pointed to by *s1* to the area pointed to by *s2* using the `memcpy()` function.

### Returned Value

The `bcopy()` function returns no value.

There are no `errno` values defined for `bcopy()`.

### Related Information

- “strings.h” on page 46
- “`bcmp()` — Compare Bytes in Memory” on page 126
- “`bzero()` — Zero Out Bytes in Memory” on page 140
- “`memccpy()` — Copy Bytes in Memory” on page 808
- “`memcpy()` — Copy Buffer” on page 813
- “`memmove()` — Move Buffer” on page 814

## bind() — Bind a Name to a Socket

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int bind(int socket, const struct sockaddr *address, size_t address_len);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>

int bind(int socket, struct sockaddr *address, int *address_len);
```

### General Description

The `bind()` call binds a unique local name to the socket with descriptor *socket*. After calling `socket()`, a descriptor does not have a name associated with it. However, it does belong to a particular address family as specified when `socket()` is called. The exact format of a name depends on the address family.

Parameter	Description
<i>socket</i>	The socket descriptor returned by a previous <code>socket()</code> call.
<i>name</i>	The pointer to a <b>sockaddr</b> structure containing the name that is to be bound to <i>socket</i> .
<i>namelen</i>	The size of <i>name</i> in bytes.

The *socket* parameter is a socket descriptor of any type created by calling `socket()`.

The *name* parameter is a pointer to a buffer containing the name to be bound to *socket*. The *namelen* parameter is the size, in bytes, of the buffer pointed to by *name*. For `AF_UNIX`, this function creates a file that you later need to unlink besides closing the socket.

#### Socket Descriptor Created in the AF\_INET Domain

If the socket descriptor *socket* was created in the `AF_INET` domain, the format of the name buffer is expected to be **sockaddr\_in**, as defined in the include file **netinet/in.h**:

```

struct in_addr
{
    ip_addr_t s_addr;
};

struct sockaddr_in {
    unsigned char  sin_len;
    unsigned char  sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    unsigned char  sin_zero[8];
};

```

The *sin\_family* field must be set to AF\_INET.

The *sin\_port* field is set to the port to which the application must bind. It must be specified in network byte order. If *sin\_port* is set to 0, the caller leaves it to the system to assign an available port. The application can call `getsockname()` to discover the port number assigned.

The *sin\_addr.s\_addr* field is set to the Internet address and must be specified in network byte order. On hosts with more than one network interface (called multi-homed hosts), a caller can select the interface to which it is to bind. Subsequently, only UDP packets and TCP connection requests from this interface (which match the bound name) are routed to the application. If this field is set to the constant INADDR\_ANY, as defined in **netinet/in.h**, the caller is requesting that the socket be bound to all network interfaces on the host. Subsequently, UDP packets and TCP connections from all interfaces (which match the bound name) are routed to the application. This becomes important when a server offers a service to multiple networks. By leaving the address unspecified, the server can accept all UDP packets and TCP connection requests made for its port, regardless of the network interface on which the requests arrived.

The *sin\_zero* field is not used and must be set to all zeros.

### Socket Descriptor Created in the AF\_UNIX Domain

If the socket descriptor *socket* is created in the AF\_UNIX domain, the format of the name buffer is expected to be **sockaddr\_un**, as defined in the include file **un.h**.

```

struct sockaddr_un {
    unsigned char  sun_len;
    unsigned char  sun_family;
    char  sun_path[108];      /* pathname */
};

```

The *sun\_family* field is set to AF\_UNIX.

The *sun\_path* field contains the null-terminated pathname, and *sun\_len* contains the length of the pathname.

**Notes:**

1. For AF\_UNIX, when a bind is issued, a file is created with a mode of 660. In order to allow other users to access this file, a chmod() should be issued to modify this mode if desired.
2. For AF\_UNIX, when closing sockets that were bound, you should also use unlink() to delete the file created at bind() time.
3. The pathname the client uses on the bind() must be unique.
4. The sendto() call must specify the pathname associated with the server.
5. For AF\_INET, the user must have appropriate privileges to bind to a port in the range from 1 to 1023.

**Special Behavior for C++**

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

**Returned Value**

The value 0 indicates success; the value -1 indicates an error. The value of the error code indicates the specific error.

Error Code	Description
EACCES	Permission denied.
EADDRINUSE	The address is already in use. See the SO_REUSEADDR option described under “getsockopt() — Get the Options Associated with a Socket” on page 606 and the SO_REUSEADDR described under the “setsockopt() — Set Options Associated with a Socket” on page 1270 for more information.
EADDRNOTAVAIL	The address specified is not valid on this host. For example, the Internet address does not specify a valid network interface.
EAFNOSUPPORT	The address family is not supported (it is not AF_UNIX or AF_INET).
EBADF	The <i>socket</i> parameter is not a valid socket descriptor.
EINVAL	The socket is already bound to an address—for example, trying to bind a name to a socket that is already connected. Or the socket was shut down.
EIO	There has been a network or transport failure.
ENOBUFS	bind() is unable to obtain a buffer due to insufficient storage.
ENOTSOCK	The descriptor is for a file, not for a socket.
EOPNOTSUPP	The socket type of the specified socket does not support binding to an address.
EPERM	The user is not authorized to bind to the port specified.

The following are for AF\_UNIX only:

Error Code	Description
EDESTADDRREQ	The <i>address</i> argument is a null pointer.



EACCES	A component of the path prefix denies search permission, or the requested name requires writing in a directory with a mode that denies write permission.
ENOTDIR	A component of the path prefix of the pathname in <i>address</i> is not a directory.
ENAMETOOLONG	A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX characters.
ENOENT	A component of the pathname does not name an existing file or the pathname is an empty string.
ELOOP	Too many symbolic links were encountered in translating the pathname in <i>address</i> .
EIO	An I/O error occurred.
EROFS	The name would reside on a read-only file system.

### Example

The following are examples of the `bind()` call. It is a good idea to zero the structure before using it to ensure that the name requested does not set any reserved fields.

#### AF\_INET Domain Example

The following example illustrates the `bind()` call binding to interfaces in the AF\_INET domain. The Internet address and port must be in network byte order. To put the port into network byte order, the `htons()` utility routine is called to convert a short integer from host byte order to network byte order. The *address* field is set using another utility routine, `inet_addr()`, which takes a character string representing the dotted-decimal address of an interface and returns the binary Internet address representation in network byte order.

```
int rc;
int s;
struct sockaddr_in myname;

/* Bind to a specific interface in the Internet domain */
/* make sure the sin_zero field is cleared */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr.s_addr = inet_addr("129.5.24.1"); /* specific interface */
myname.sin_port = htons(1024);
:
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
/* Bind to all network interfaces in the Internet domain */
/* make sure the sin_zero field is cleared */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr.s_addr = INADDR_ANY; /* specific interface */
myname.sin_port = htons(1024);
:
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
/* Bind to a specific interface in the Internet domain.
   Let the system choose a port */
/* make sure the sin_zero field is cleared */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
```

```

myname.sin_addr.s_addr = inet_addr("129.5.24.1"); /* specific interface */
myname.sin_port = 0;
:
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));

```

### **AF\_UNIX Domain Example**

The following example illustrates the `bind()` call binding to interfaces in the `AF_UNIX` domain.

```

/* Bind to a name in the UNIX domain */
struct sockaddr_un myname;
char socket_name[]="/tmp/socket.for._";
:
memset(&myname, 0, sizeof(myname));
myname.sun_family = AF_UNIX;
strcpy(myname.sun_path,socket_name);
myname.sun_len = sizeof(myname.sun_path);
:
rc = bind(s, (struct sockaddr *) &myname, SUN_LEN(&myname));

```

### **Related Information**

- “`connect()` — Connect a Socket” on page 214
- “`getnetbyname()` — Get a Network Entry by Name” on page 560
- “`getsockname()` — Get the Name of a Socket” on page 604
- “`htons()` — Translate an Unsigned Short Integer into Network Byte Order” on page 649
- “`inet_addr()` — Translate an Internet Address into Network Byte Order” on page 661
- “`listen()` — Prepare the Server for Incoming Client Requests” on page 752
- “`socket()` — Create a Socket” on page 1371

## brk() — Change Space Allocation

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
```

```
int brk(void * addr);
```

### General Description

The `brk()` function is used to change the space allocated for the calling process. The change is made by setting the process' break value to `addr` and allocating the appropriate amount of space. The amount of allocated space increases as the break value increases. The newly-allocated space is set to 0. However, if the application first decrements and then increments the break value, the contents of the reallocated space are not zeroed.

The storage space from which the `brk()` and `sbrk()` functions allocate storage is separate from the storage space that is used by the other memory allocation functions (`malloc()`, `calloc()`, etc.). Because this storage space must be a contiguous segment of storage, it is allocated from the initial heap segment only and thus is limited to the initial heap size specified for the calling program or the largest contiguous segment of storage available in the initial heap at the time of the first `brk()` or `sbrk()` call. Since this is a separate segment of storage, the `brk()` and `sbrk()` functions can be used by an application that is using the other memory allocation functions. However, it is possible that the user's region may not be large enough to support extensive usage of both types of memory allocation.

Prior usage of the `brk()` function has been limited to specialized cases where no other memory allocation function performed the same function. Because the `brk()` function may be unable to sufficiently increase the space allocation of the process when the calling application is using other memory functions, the use of other memory allocation functions, such as `mmap()`, is now preferred because it can be used portably with all other memory allocation functions and with any function that uses other allocation functions. Applications that require the use of `brk()` and/or `sbrk()` should refrain from using the other memory allocation functions and should be run with an initial heap size that will satisfy the maximum storage requirements of the program. The `brk()` function is not supported from a multi-threaded environment, it will return in error if it is invoked in this environment.

### Returned Value

If successful, `brk()` returns 0. Otherwise, it returns -1 and sets `errno` to indicate the error. The following are the possible values of `errno`:

**ENOMEM** The requested change would allocate more space than allowed for the calling process, or the caller is running in a multi-threaded environment, which is not a valid environment for this function.

**Related Information**

- “sbrk() — Change Space Allocation” on page 1152

## bsd\_signal() — BSD Version of signal()

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON) MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>
```

```
void (*bsd_signal (int sig, void (*func)(int)))(int);
```

### General Description

The bsd\_signal() function provides a partially compatible interface for programs written to use the BSD form of the signal() function.

BSD signal() differs from ANSI signal() in that the **SA\_RESTART** flag is set and the **SA\_RESETHAND** is cleared when bsd\_signal() is used. Whereas for signal() both of these flags are cleared and **\_SA\_OLD\_STYLE** is set.

There are three functions available for establishing a signal's action, signal(), bsd\_signal(), and sigaction(). The sigaction() function is the strategic way to establish a signal's action. The bsd\_signal() and signal() functions are provided for compatibility with BSD and ANSI, respectively.

The argument *sig* is the signal type. See Table 32 on page 1295 for a list of the supported signal types or refer to the <signal.h> header. The argument *func* is the signal action. It may be set to **SIG\_DFL**, **SIG\_IGN**, or the address of a signal catching function that takes one input argument.

### Special Behavior for C++

Because C and C++ linkage conventions are incompatible, bsd\_signal() cannot receive a C++ function pointer as the start routine function pointer. If you attempt to pass a C++ function pointer to bsd\_signal(), the compiler will flag it as an error. You can pass a C or C++ function to bsd\_signal() by declaring it as extern "C".

### Returned Value

If successful, bsd\_signal() returns the previous action established for this signal type.

If unsuccessful, bsd\_signal() returns **SIG\_ERR** and returns the error value in errno. The following are the possible values of errno:

**EINVAL** The value of the argument *sig* was not a valid signal type, or an attempt was made to catch a signal that cannot be caught, or ignore a signal that cannot be ignored.

### **Related Information**

- “signal.h” on page 41
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “signal() — Handle Interrupts” on page 1330
- “sigprocmask() — Examine or Change a Thread” on page 1339

## bsearch() — Search Arrays

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
void *bsearch(const void *key, const void *base, size_t num, size_t size,
              int (*compare)(const void *element1, const void *element2));
```

### General Description

Performs a binary search of an array of *num* elements, each of *size* bytes.

The pointer *base* points to the initial element of the array to be searched. *key* points to the object containing the value being sought. The array must be sorted in ascending key sequence, according to the comparison function. Otherwise, undefined behavior occurs.

The *compare* parameter is a pointer to a function you must supply. It compares two array elements and returns a value specifying their relationship. The `bsearch()` function calls this function one or more times during the search, passing the key and the pointer to one array element on each call. The function compares the elements and then returns one of the following values:

Value	Meaning
Less than 0	Object pointed to by key is less than the array element.
0	Object pointed to by key is equal to the array element.
Greater than 0	Object pointed to by key is greater than the array element.

### Special Behavior for C++

Because C++ and C linkage conventions are incompatible, `bsearch()` cannot receive C++ function pointers. If you attempt to pass a C++ function pointer to `bsearch()`, the compiler will flag it as an error. To use the C++ `bsearch()` function, you must ensure that the *compare* function has C linkage by declaring it as extern “C”.

### Returned Value

Returns a pointer to a matching element of the array. If two or more elements are equal, the element pointed to is not specified.

If `bsearch()` cannot find the *key*, it returns NULL.

## Example

### CBC3BB01

```

/* CBC3BB01
   This example performs a binary search on the argv array of pointers
   to the program arguments and finds the position of the argument PATH.
   It first removes the program name from argv, and then sorts the
   array alphabetically before calling bsearch().

   The functions compare1 and compare2 compare the values pointed to
   by arg1 and arg2, and they return the result to bsearch().
*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#ifdef __cplusplus /* the __cplusplus macro is */
extern "C" {      /* automatically defined by the */
#endif           /* C++/MVS compiler */
    int compare1(const void *arg1, const void *arg2);
    int compare2(const void *arg1, const void *arg2);
#ifdef __cplusplus
}
#endif

int main(int argc, char *argv[])
{
    char **result;
    char *key = "PATH";
    int i;
    argv++;
    argc--;

    /* sort to ensure that the input is ordered */
    qsort((char *)argv, argc, sizeof(char *), compare1);

    result = (char**)bsearch(key, (void *)argv, argc, sizeof(char *),
                           compare2);
    if (result != NULL) {
        printf("The key <%s> was found.\n",*result);
    }
    else printf("Match not found\n");
}

int compare1(const void *arg1, const void *arg2)
{
    return (strcmp(*(char **)arg1, *(char **)arg2));
}

int compare2(const void *arg1, const void *arg2) {
    return (strcmp((char *)arg1, *(char **)arg2));
}

```

### Input

programe Is there PATH in this sentence?

### Output

The key <PATH> was found.



**Related Information**

- “stdlib.h” on page 45
- “qsort() — Sort Array” on page 1068

## bzero() — Zero Out Bytes in Memory

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <strings.h>
```

```
void bzero(void *s, size_t n);
```

### General Description

The `bzero()` function places *n* zero-valued bytes in the area pointed to by *s*.

### Returned Value

The `bzero()` function returns no value.

There are no `errno` values defined for `bzero()`.

### Related Information

- “strings.h” on page 46
- “bcmp() — Compare Bytes in Memory” on page 126
- “bcopy() — Copy Bytes in Memory” on page 127
- “memset() — Set Buffer to Value” on page 816

## calloc() — Reserve and Initialize Storage

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
void *calloc(size_t num, size_t size);
```

### General Description

Reserves storage space for an array of *num* elements, each of length *size* bytes. The `calloc()` function then gives all the bits of each element an initial value of 0.

The `calloc()` function returns a pointer to the reserved space. The storage space to which the returned value points is aligned for storage of any type of object.

This function is also available to C applications in free-standing Systems Programming Facility applications.

### Special Behavior for C++

The C++ keywords `new` and `delete` are not interoperable with `calloc()`, `free()`, `malloc()`, or `realloc()`.

### Returned Value

Returns the pointer to the area of memory reserved. If there is not enough space to satisfy the request or if *num* or *size* is 0, `calloc()` returns NULL. If `calloc()` returns a NULL because there is not enough storage, it will also return an error value in `errno`. The following are the possible values of `errno`:

ENOMEM    Insufficient memory is available

### Example

#### CBC3BC01

```
/* CBC3BC01
   This example prompts for the number of array entries required and then
   reserves enough space in storage for the entries.
   If calloc() is successful, the example prints out each entry;
   otherwise, it prints out an error message.
*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    long * array;    /* start of the array */
    long * index;    /* index variable */
    int    i;        /* index variable */
    int    num;      /* number of entries in the array */
```

```

printf( "Enter the number of elements in the array\n" );
scanf( "%i", &num );

/* allocate num entries */
if ( (index = array = (long *)calloc( num, sizeof( long ))) != NULL )
{
    for ( i = 0; i < num; ++i )          /* put values in array */
        *index++ = i;                  /* using pointer notation */

    for ( i = 0; i < num; ++i )          /* print the array out */
        printf( "array[ %i ] = %i\n", i, array[i] );
}
else
{ /* out of storage */
    printf( "Out of storage\n" );
    abort();
}
}

```

**Output**

```

Enter the size of the array
array[ 0 ] = 0
array[ 1 ] = 1
array[ 2 ] = 2

```

**Related Information**

- “Using the System Programming C Facilities” in the *OS/390 C/C++ Programming Guide*.
- “stdlib.h” on page 45
- “free() — Free a Block of Storage” on page 458
- “malloc() — Reserve Storage Block” on page 786
- “realloc() — Change Reserved Storage Block Size” on page 1096

## catclose() — Close a Message Catalog Descriptor

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE
#include <nl_types.h>
```

```
int catclose(nl_catd catd);
```

### General Description

The `catclose()` function closes the message catalog identified by `catd`. If a catalog is opened more than once in the same process, a use count is incremented.

`catclose()` decrements this use count. When the use count reaches zero then the file descriptor for that catalog is closed.

### Returned Value

Upon successful completion, the `catclose()` function returns a value of zero. Otherwise a value of -1 is returned, and `errno` is set to indicate the error.

The `catclose()` function may fail if:

EBADF	The catalog descriptor is not valid.
EINTR	The <code>catclose()</code> function was interrupted by a signal.

### Related Information

- “`nl_types.h`” on page 38
- “`catopen()` — Open a Message Catalog” on page 146
- “`catgets()` — Read a Program Message” on page 144

## catgets() — Read a Program Message

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE
#include <nl_types.h>
```

```
char *catgets(nl_catd catd, int set_id, int msg_id, const char *s);
```

### General Description

The `catgets()` function attempts to read message `msg_id`, in set `set_id`, from the message catalog identified by `catd`. The `catd` argument is a message catalog descriptor returned from an earlier call to `catopen()`. The `s` argument points to a default message string which will be returned by `catgets()` if it cannot retrieve the identified message.

When message source files are processed by the `gencat` command, the CODESET used to create them is saved in the resulting message catalog. The `catgets()` function interrogates this CODESET value to see if it differs from the CODESET value of the current locale. If it does differ then `catgets()` uses the `iconv()` function to convert the message text coming from the message catalog into the codeset of the current locale. The default message string (`s`) is not affected by this conversion. If `iconv()` does not support the conversion specified by the two CODESETs then the default message string is returned.

### Returned Value

If the identified message is retrieved successfully, `catgets()` returns a pointer to an internal buffer area containing the null-terminated message string. If the call is unsuccessful for any reason, `s` is returned.

The `catgets()` function may fail if:

EBADF	The <code>catd</code> argument is not a valid message catalog descriptor open for reading.
EINTR	The read operation was terminated due to the receipt of a signal, and no data was transferred.

### Special Behavior for OS/390 UNIX Services

- EINVAL May be returned for several reasons:
- The message catalog identified by `catd` is not a valid message catalog, or has been corrupted. Ensure that the message catalog was created using the OS/390 UNIX `gencat` command.
  - The `iconv` function does not support the conversion between the codeset of the message catalog and that of the current locale. To check the codeset that the message catalog was created in, look for the codeset name at offset 28 into the message catalog.

ENOMSG    The message identified by `set_id` and `msg_id` is not in the message catalog.

**Related Information**

- “`nl_types.h`” on page 38
- “`catclose()` — Close a Message Catalog Descriptor” on page 143
- “`catopen()` — Open a Message Catalog” on page 146

## catopen() — Open a Message Catalog

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE
#include <nl_types.h>
```

```
nl_catd catopen(const char *name, int oflag);
```

### General Description

The `catopen()` function opens a message catalog and returns a message catalog descriptor. The name argument specifies the name of the message catalog to be opened. If name contains a "/", then name specifies a complete name for the message catalog. Otherwise, the environment variable NLSPATH is used with name substituted for %N (see the XBD specification, Chapter 6, Environment Variables). If NLSPATH does not exist in the environment, or if a message catalog cannot be found in any of the components specified by NLSPATH, then the default path of "/usr/lib/nls/msg/%L/%N" is used. The "%L" component of this default path is replaced by the setting of LC\_MESSAGES if the value of oflag is NL\_CAT\_LOCALE, or the LANG environment variable if oflag is 0. A change in the setting of the LANG or LC\_MESSAGES will have no effect on existing open catalogs.

A message catalog descriptor remains valid in a process until that process closes it, or a successful call to one of the exec functions. When a message catalog is opened the FD\_CLOEXEC flag will be set. See "fcntl() — Control Open File Descriptors" on page 350. Portable applications must assume that message catalog descriptors are not valid after a call to one of the exec functions.

If a catalog is opened more than once in the same process, a use count is incremented. When the use count reaches zero, by using `catclose()` to close the catalog, then the file descriptor for that catalog is closed.

### Returned Value

Upon successful completion, `catopen()` returns a message catalog descriptor for use on subsequent calls to `catgets()` and `catclose()`. Otherwise `catopen()` returns `(nl_catd) -1` and sets `errno` to indicate the error.

The `catopen()` function may fail if:

- EACCES** Search permission is denied for the component of the path prefix of the message catalog or read permission is denied for the message catalog.
- EMFILE** {OPEN\_MAX} file descriptors are currently open in the calling process.



**ENAMETOOLONG**

The length of the pathname of the message catalog exceeds {PATH\_MAX}, or a pathname component is longer than {NAME\_MAX}.

**ENFILE** Too many files are currently open in the system.

**ENOENT** The message catalog does not exist or the name argument points to an empty string.

**ENOMEM** Insufficient storage space is available.

**ENOTDIR** A component of the path prefix of the message catalog is not a directory.

**Related Information**

- “nl\_types.h” on page 38
- “catclose() — Close a Message Catalog Descriptor” on page 143
- “catgets() — Read a Program Message” on page 144

## cbrt() — Cube Root

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>
```

```
double cbrt(double x);
```

### General Description

The `cbrt()` function calculates the cube root of its argument `x`.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

`cbrt()` does not fail.

### Related Information

- “math.h” on page 35

## cclass() — Return Characters in a Character Class

### Standards

Standards / Extensions	C or C++	Dependencies
C library	both	

### Format

```
#include <collate.h>
```

```
int cclass(char *class, coll_t **list);
```

### General Description

Finds all the collating elements of the class, `class`. The list is updated to point to the array of collating elements found. The list is valid until the next call to `setlocale()`.

The function supports user-defined character classes. In C/MVS programs, the function also supports POSIX.2 character classes.

### Returned Value

The number returned is the number of elements in the list pointed to by `list` or `-1`. The function, `cclass()`, returns `-1` if its first argument specifies a class that does not exist in the `LC_CTYPE` category of the current locale.

### Example

#### CBC3BC02

```
/* CBC3BC02 */
#include <stdio.h>
#include <collate.h>

int main(void)
{
    coll_t *list;      /* ptr to the digit class collation weights */
    int weights;      /* no. of class collation class weights found */
    int i;

    weights = cclass("digit", &list);

    printf("weights=%d\n", weights);
    for (i=0; i<weights; i++)
        printf("*(list + %d) = %d\n", i, *(list + i) );
}
```

### Related Information

- “locale.h” on page 33
- “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*
- “collequiv() — Return a List of Equivalent Collating Elements” on page 200
- “collorder() — Return List of Collating Elements” on page 202
- “collrange() — Calculate the Range List of Collating Elements” on page 204
- “colltostr() — Return a String for a Collating Element” on page 206
- “getmccoll() — Get Next Collating Element from String” on page 554
- “getwmccoll() — Get Next Collating Element from Wide String” on page 632
- “ismccollet() — Identify a Multi-Character Collating Element” on page 710

- “maxcoll() — Return Maximum Collating Element” on page 788
- “setlocale() — Set Locale” on page 1241
- “strtolcoll() — Return Collating Element for String” on page 1454
- “wctype() — Obtain Handle for Character Property Classification” on page 1753

## cds() — Compare Double and Swap

### Standards

Standards / Extensions	C or C++	Dependencies
C library	both	

### Format

```
#include <stdlib.h>
```

```
int cds(cds_t *oldptr, cds_t *curptr, cds_t newwords);
```

### General Description

The `cds()` built-in function compares the 8-byte value pointed to by *oldptr* to the 8-byte value pointed to by *curptr*. If they are equal, the 8-byte value *newwords* is copied into the location pointed to by *curptr*. If they are unequal, the value pointed to by *curptr* is copied into the location pointed to by *oldptr*.

To avoid infringing on the user's name-space, this non-standard function is exposed only when you use the compiler option, `LANGLVL(EXTENDED)`. When you use `LANGLVL(EXTENDED)` any relevant information in the header is also exposed.

The function uses the System/370 CDS instruction. For a detailed description, see the appendixes in the *ESA/390 Principles of Operations*.

### Returned Value

Returns 0 if the 8-byte value pointed to by *oldptr* is equal to the 8-byte value pointed to by *curptr*. Otherwise it returns 1.

### Related Information

- *ESA/390 Principles of Operations*
- “`stdlib.h`” on page 45
- “`cs()` — Compare and Swap” on page 239

## cdump() — Request a Main Storage Dump

### Standards

Standards / Extensions	C or C++	Dependencies
C library	both	

### Format

```
#include <ctest.h>
```

```
int cdump(char *dumptitle);
```

### General Description

1. Creates a display of the activation stack, by calling trace in the same way as the ctrace() function does.
2. Displays the LE-formatted dump.
3. If the source file was compiled with TEST(SYM), cdump() displays the contents of the user's variables. The output is identified with *dumptitle*. See the CEE3DMP Language Environment callable service in the *OS/390 Language Environment Programming Guide*, SC28-1939 to determine where the output is written to.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANTLRVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with LANTLRVL(EXTENDED). When you use LANTLRVL(EXTENDED) any relevant information in the header is also exposed.

### Returned Value

Returns 0 if successful, and nonzero otherwise.

### Related Information

- “ctest.h” on page 24
- “csnap() — Request a Condensed Dump” on page 242
- “ctrace() — Request a Traceback” on page 252

# ceil() — Round Up to Integral Value

## Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

## Format

#include <math.h>

double ceil(double x);

## General Description

Computes the smallest integer that is greater than or equal to x.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

## Returned Value

Returns the calculated value as a double value. If there is an overflow, ceil() sets errno to ERANGE and returns HUGE\_VAL.

## Example CBC3BC04

```
/* CBC3BC04
   This example sets y to the smallest integer greater than 1.05, and then
   to the smallest integer greater than -1.05.

   The results are 2.0 and -1.0, respectively.
*/
#include <math.h>
#include <stdio.h>
int main(void)
{
    double y, z;

    y = ceil(1.05);      /* y = 2.0 */
    z = ceil(-1.05);     /* z = -1.0 */
    printf("y = %f\n z = %f\n", y, z);
}
```

## Related Information

- “math.h” on page 35
- “floor() — Round Down to Integral Value” on page 408

## \_\_certificate() — Register/Deregister a Digital Certificate

### Standards

Standards/Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 UNIX

### Format

```
#define _OPEN_SYS
#include <unistd.h>

int __certificate(int function_code,
                  int certificate_length);
char *certificate);
```

### General Description

The \_\_certificate() function allows the user to register or deregister a digital certificate to authenticate the user to gain access to protected data, environments and applications.

The function takes the following arguments:

*function\_code*

Specifies one of the following functions:

\_\_CERTIFICATE\_REGISTER

Register the passed certificate to the user.

\_\_CERTIFICATE\_DEREGISTER

Deregister the passed certificate from the user. Certificate must have been previously registered to the user.

*certificate\_length*

The length of the digital certificate. A zero length will cause -1 return value with errno set to EINVAL.

*certificate*

The certificate must be a single BER encoded X.509 certificate. PKCS7, PEM or Base64 encoded certificates are allowed.

### Usage Notes

1. The \_\_certificate() function calls the kernel, but it is intended for servers that support the automatic registration of certificates for clients they are supporting (on the World Wide Web for example).
2. The BPX1SEC service will associate the passed certificate with whatever user identity is present. If the task level identity is present the certificate is associated with the task. Task level security can be created by the pthread\_security(), \_\_login() or by any other means of creating a task level ACEE. If no task level identity (ACEE) is present, the certificate will be associated with the address space identity.



## Returned Value

If the request is successful, \_\_certificate() returns 0.

If the request is unsuccessful, \_\_certificate() returns -1 and sets *errno* to indicate the error.

## Restrictions

A security manager supporting digital certificate registration and deregistration must be installed and operational.

## Errors

If unsuccessful, \_\_certificate() returns -1 and sets *errno* to one of the following:

Errno Code	Description
EINVAL	A parameter is invalid.
EPERM	The operation was not permitted. Calling process may not be authorized in BPX.DAEMON facility class.
EMVSSAF2ERR	An error occurred in the security product. Certificate is already defined for another process or certificate is invalid or certificate does not meet required format. Also, realized when an internal error has occurred.
EMVSERR	An MVS environmental error or internal error occurred.
ENOSYS	The function is not implemented or installed.
EACCES	Permission is denied.

## Related Information

- “\_\_login() — Create a New Security Environment for Process” on page 764

## cfgetispeed() — Determine the Input Baud Rate

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	

### Format

```
#define _POSIX_SOURCE
#include <termios.h>
```

```
speed_t cfgetispeed(const struct termios *term_ptr);
```

### General Description

Extracts the input baud rate from the termios structure indicated by *\*term\_ptr*. The termios structure contains information about a terminal. A program should first use `tcgetattr()` to get the termios structure, and then use `cfgetispeed()` to extract the speed from the structure. The program can then use `cfsetispeed()` to set a new baud rate in the structure and `tcsetattr()` to pass the changed value to the system.

Although in an OS/390 UNIX application valid speeds can be set with `cfsetispeed()` and passed to the system with `tcsetattr()`, the speed has no effect on the operation of a pseudoterminal. However, the operation will have an effect if issued for an OCS remote terminal.

### Returned Value

Returns a code indicating the baud rate; see Table 17.

There are no documented errnos for this function.

*Table 17 (Page 1 of 2). Baud Rate Codes*

B0	Hang up
B50	50 baud
B75	75 baud
B110	110 baud
B134	134.5 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19,200 baud

Table 17 (Page 2 of 2). Baud Rate Codes

B38400	38,400 baud
--------	-------------

### Example CBC3BC05

```

/* CBC3BC05
   This example determines the speed of stdin.
*/
#define _POSIX_SOURCE
#include <termios.h>
#include <stdio.h>

char *see_speed(speed_t speed) {
    static char  SPEED[20];
    switch (speed) {
        case B0:      strcpy(SPEED, "B0");
                        break;
        case B50:     strcpy(SPEED, "B50");
                        break;
        case B75:     strcpy(SPEED, "B75");
                        break;
        case B110:    strcpy(SPEED, "B110");
                        break;
        case B134:    strcpy(SPEED, "B134");
                        break;
        case B150:    strcpy(SPEED, "B150");
                        break;
        case B200:    strcpy(SPEED, "B200");
                        break;
        case B300:    strcpy(SPEED, "B300");
                        break;
        case B600:    strcpy(SPEED, "B600");
                        break;
        case B1200:   strcpy(SPEED, "B1200");
                        break;
        case B1800:   strcpy(SPEED, "B1800");
                        break;
        case B2400:   strcpy(SPEED, "B2400");
                        break;
        case B4800:   strcpy(SPEED, "B4800");
                        break;
        case B9600:   strcpy(SPEED, "B9600");
                        break;
        case B19200:  strcpy(SPEED, "B19200");
                        break;
        case B38400:  strcpy(SPEED, "B38400");
                        break;
        default:      sprintf(SPEED, "unknown (%d)", (int) speed);
    }
    return SPEED;
}

main() {
    struct termios term;
    speed_t speed;

    if (tcgetattr(0, &term) != 0)
        perror("tcgetattr() error");
    else {
        speed = cfgetispeed(&term);
        printf("cfgetispeed() says the speed of stdin is %s\n",

```

```
        see_speed(speed));  
    }  
}
```

**Output**

cfgetispeed() says the speed of stdin is B0

**Related Information**

- “termios.h” on page 51
- “cfgetospeed() — Determine the Output Baud Rate” on page 159
- “cfsetispeed() — Set the Input Baud Rate in the Termios” on page 161
- “cfsetospeed() — Set the Output Baud Rate in the Termios” on page 163
- “tcgetattr() — Get the Attributes for a Terminal” on page 1515
- “tcsetattr() — Set the Attributes for a Terminal” on page 1531

## cfgetospeed() — Determine the Output Baud Rate

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	

### Format

```
#define _POSIX_SOURCE
#include <termios.h>
```

```
speed_t cfgetospeed(const struct termios *term_ptr);
```

### General Description

Extracts the output baud rate from the termios structure indicated by *\*term\_ptr*. The termios structure contains information about a terminal. A program should first use `tcgetattr()` to get the termios structure, and then use `cfgetospeed()` to extract the speed from the structure. The program can then use `cfsetospeed()` to set a new baud rate in the structure and `tcsetattr()` to pass the changed value to the system.

Although in an OS/390 UNIX application valid speeds can be set with `cfsetospeed()` and passed to the system with `tcsetattr()`, the speed has no effect on the operation a pseudoterminal. However, the operation will have an effect if issued for an OCS remote terminal.

### Returned Value

Returns a code indicating the baud rate. The codes are defined in the `termios.h` header file and have an unsigned integer type. Table 17 on page 156 shows the codes to set the baud rate.

There are no documented errors for this function.

### Example

#### CBC3BC06

```
/* CBC3BC06
   This example determines the speed of stdout.
*/
#define _POSIX_SOURCE
#include <termios.h>
#include <stdio.h>

char *see_speed(speed_t speed) {
    static char  SPEED[20];
    switch (speed) {
        case B0:      strcpy(SPEED, "B0");
                       break;
        case B50:     strcpy(SPEED, "B50");
                       break;
        case B75:     strcpy(SPEED, "B75");
                       break;
        case B110:    strcpy(SPEED, "B110");
                       break;
        case B134:    strcpy(SPEED, "B134");
                       break;
```

```

        case B150:    strcpy(SPEED, "B150");
                      break;
        case B200:    strcpy(SPEED, "B200");
                      break;
        case B300:    strcpy(SPEED, "B300");
                      break;
        case B600:    strcpy(SPEED, "B600");
                      break;
        case B1200:   strcpy(SPEED, "B1200");
                      break;
        case B1800:   strcpy(SPEED, "B1800");
                      break;
        case B2400:   strcpy(SPEED, "B2400");
                      break;
        case B4800:   strcpy(SPEED, "B4800");
                      break;
        case B9600:   strcpy(SPEED, "B9600");
                      break;
        case B19200:  strcpy(SPEED, "B19200");
                      break;
        case B38400:  strcpy(SPEED, "B38400");
                      break;
        default:      sprintf(SPEED, "unknown (%d)", (int) speed);
    }
    return SPEED;
}

main() {
    struct termios term;
    speed_t speed;

    if (tcgetattr(1, &term) != 0)
        perror("tcgetattr() error");
    else {
        speed = cfgetospeed(&term);
        printf("cfgetospeed() says the speed of stdout is %s\n",
               see_speed(speed));
    }
}

```

**Output**

cfgetospeed() says the speed of stdout is B0

**Related Information**

- “termios.h” on page 51
- “cfgetispeed() — Determine the Input Baud Rate” on page 156
- “cfsetispeed() — Set the Input Baud Rate in the Termios” on page 161
- “cfsetospeed() — Set the Output Baud Rate in the Termios” on page 163
- “tcgetattr() — Get the Attributes for a Terminal” on page 1515
- “tcsetattr() — Set the Attributes for a Terminal” on page 1531

## cfsetispeed() — Set the Input Baud Rate in the Termios

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	

### Format

```
#define _POSIX_SOURCE
#include <termios.h>
```

```
int cfsetispeed(struct termios *term_ptr, speed_t speed);
```

### General Description

Specifies a new input baud rate for the termios control structure, *\*term\_ptr*. `cfsetispeed()` records this new baud rate in the control structure but does not actually change the terminal device file. The program must call `tcsetattr()` to modify the terminal device file to reflect the settings in the termios structure.

A program should first use `tcgetattr()` to get the termios structure. Then it should use `cfsetispeed()` to set the speed in `termios` and `tcsetattr()` to pass the modified termios structure to the system.

Although in an OS/390 UNIX application valid speeds can be set with `cfsetispeed()` and passed to the system with `tcsetattr()`, the speed has no effect on the operation of a pseudoterminal. However, the operation will have an effect if issued for an OCS remote terminal.

The *speed* argument indicates the new baud rate with one of the following codes, defined in the `termios.h` header file. The codes have an unsigned integer type. Table 17 on page 156 shows the codes to set the baud rate.

### Returned Value

If `cfsetispeed()` successfully sets the baud rate in the control structure, it returns 0. If unsuccessful, it returns -1 and sets `errno` to `EINVAL`, which indicates an unsupported value for *speed*.

### Example CBC3BC07

```
/* CBC3BC07
   This example specifies a new input baud rate.
*/
#define _POSIX_SOURCE
#include <termios.h>
#include <stdio.h>

main() {
    struct termios term;

    if (tcgetattr(0, &term) != 0)
        perror("tcgetattr() error");
    else if (cfsetispeed(&term, B0) != 0)
        perror("cfsetispeed() error");
```

```
else if (tcsetattr(0, TCSANOW, &term) != 0)
    perror("tcsetattr() error");
}
```

**Related Information**

- “termios.h” on page 51
- “cfgetispeed() — Determine the Input Baud Rate” on page 156
- “cfgetospeed() — Determine the Output Baud Rate” on page 159
- “cfsetospeed() — Set the Output Baud Rate in the Termios” on page 163
- “tcgetattr() — Get the Attributes for a Terminal” on page 1515
- “tcsetattr() — Set the Attributes for a Terminal” on page 1531



## cfsetospeed() — Set the Output Baud Rate in the Termios

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	

### Format

```
#define _POSIX_SOURCE
#include <termios.h>
```

```
int cfsetospeed(struct termios *term_ptr, speed_t speed);
```

### General Description

Specifies a new output baud rate for the termios control structure, *\*term\_ptr*. `cfsetospeed()` records this new baud rate in the control structure, but does not actually change the terminal device file. The program must call `tcsetattr()` to modify the terminal device file to reflect the settings in the termios structure.

A program should first use `tcgetattr()` to get the termios structure. It should then use `cfsetospeed()` to set the speed in termios and `tcsetattr()` to pass the modified termios structure to the system.

Although in an OS/390 UNIX application valid speeds can be set with `cfsetospeed()` and passed to the system with `tcsetattr()`, the speed has no effect on the operation of a pseudoterminal. However, the operation will have an effect if issued for an OCS remote terminal.

The *speed* argument should be a code indicating the new baud rate. These codes are defined in the `termios.h` header file and have an unsigned integer type. Table 17 on page 156 shows the codes to set the baud rate.

### Returned Value

If `cfsetospeed()` successfully sets the baud rate for the structure, it returns 0. If unsuccessful, it returns -1 and sets `errno` to `EINVAL`, which indicates that the *speed* is not supported by the hardware or software.

### Example CBC3BC08

```
/* CBC3BC08
   This example specifies a new output baud rate.
*/
#define _POSIX_SOURCE
#include <termios.h>
#include <stdio.h>

main() {
    struct termios term;

    if (tcgetattr(1, &term) != 0)
        perror("tcgetattr() error");
    else if (cfsetospeed(&term, B38400) != 0)
        perror("cfsetospeed() error");
```

```
else if (tcsetattr(1, TCSANOW, &term) != 0)
    perror("tcsetattr() error");
}
```

**Related Information**

- “termios.h” on page 51
- “cfgetispeed() — Determine the Input Baud Rate” on page 156
- “cfgetospeed() — Determine the Output Baud Rate” on page 159
- “cfsetispeed() — Set the Input Baud Rate in the Termios” on page 161
- “tcgetattr() — Get the Attributes for a Terminal” on page 1515
- “tcsetattr() — Set the Attributes for a Terminal” on page 1531

## chaudit() — Change Audit Flags for a File by Path

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 4.3

### Format

```
#_OPEN_SYS 1
#include <sys/stat.h>
```

```
int chaudit(const char *pathname, unsigned int flags,
            unsigned int option);
```

### General Description

Changes the audit flags for a file to indicate the type of requests the security product should audit. `chaudit()` can change user audit flags or security auditor audit flags, depending on the *option* specified.

*pathname* is the name of the file for which the audit flags are to be changed.

*flags* is the setting for the audit flags:

AUDTREADFAIL	Audit the failing read requests.
AUDTREADSUCC	Audit the successful read requests.
AUDTWRITEFAIL	Audit the failing write requests.
AUDTWritesucc	Audit the successful write requests.
AUDTEXECFAIL	Audit the failing execute or search requests.
AUDTEXECsucc	Audit the successful execute or search requests. The bitwise inclusive OR of any or all of these can be used to set more than one type of auditing.

*option* indicates whether the user audit flags or the security-auditor audit flags are to be changed:

AUDT_USER (0)	Change user flags. The user must be the file owner or have appropriate authority to change the user audit flags for a file.
AUDT_AUDITOR (1)	Change security auditor audit flags. The user must have security-auditor authority to modify the security auditor audit flags for a file.

### Returned Value

If successful, `chaudit()` returns 0. If unsuccessful, `chaudit()` returns -1 and sets `errno` to one of the following:

EACCES	The calling process does not have permission to search some component of <i>pathname</i> .
--------	--

ELOOP	A loop exists in symbolic links. This error is issued if the number of symbolic links detected in the resolution of <i>pathname</i> is greater than POSIX_SYMLINK_MAX (a value defined in the limits.h header file).
ENAMETOOLONG	<i>pathname</i> is longer than PATH_MAX characters or a component of <i>pathname</i> is longer than NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the <i>pathname</i> string substituted for a symbolic link exceeds PATH_MAX. The PATH_MAX and NAME_MAX values are determined using pathconf().
ENOENT	There is no file named <i>pathname</i> , or <i>pathname</i> is an empty string.
ENOTDIR	A component of the path prefix is not a directory.
EPERM	The effective user ID (UID) of the calling process does not match the owner of the file, and the calling process does not have appropriate privileges.
EROFS	<i>pathname</i> specifies a file that is on a read-only file system.
EINVAL	<i>option</i> is not AUDT_USER or AUDT_AUDITOR.

### Example

#### CBC3BC09

```

/* CBC3BC09
   This example changes the audit flags.
*/
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

main() {
    int fd;
    char fn[]="chaudit.file";

    if ((fd = creat(fn, S_IRUSR|S_IWUSR)) < 0)
        perror("creat() error");
    else {
        close(fd);
        if (chaudit(fn, AUDTREADFAIL, AUDT_USER) != 0)
            perror("chaudit() error");
        unlink(fn);
    }
}

```

### Related Information

- “sys/stat.h” on page 48
- “access() — Determine Whether a File Can be Accessed” on page 82
- “chmod() — Change the Mode of a File or Directory” on page 174
- “chown() — Change the Owner or Group of a File or Directory” on page 177
- “fchaudit() — Change Audit Flags for a File by Descriptor” on page 341
- “stat() — Get File Information” on page 1404

## chdir() — Change the Working Directory

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
int chdir(const char *pathname);
```

### General Description

Makes *pathname* your new working directory.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

If successful, `chdir()` changes the working directory and returns 0. If unsuccessful, `chdir()` does not change the working directory, returns `-1`, and sets `errno` to one of the following:

**EACCES** The process does not have search permission on one of the components of *pathname*.

**ELOOP** A loop exists in symbolic links. This error is issued if the number of symbolic links detected in the resolution of *pathname* is greater than `POSIX_SYMLINK` (a value defined in the `limits.h` header file).

**ENAMETOOLONG**

*pathname* is longer than `PATH_MAX` characters, or some component of *pathname* is longer than `NAME_MAX` characters while `_POSIX_NO_TRUNC` is in effect. For symbolic links, the length of the *pathname* string substituted for a symbolic link exceeds `PATH_MAX`. The `PATH_MAX` and `NAME_MAX` values are determined using `pathconf()`.

**ENOENT** *pathname* is an empty string, or the specified directory does not exist.

**ENOTDIR** Some component of *pathname* is not a directory.

### Example

#### CBC3BC10

```
/* CBC3BC10 */
#define _POSIX_SOURCE
#include <unistd.h>
#include <stdio.h>

main() {
    if (chdir("/tmp") != 0)
        perror("chdir() to /tmp failed");
    if (chdir("/chdir/error") != 0)
```

```
|         perror("chdir() to /chdir/error failed");
|     }
| }
```

## Output

```
chdir() to /chdir/error failed: No such file or directory
```

## Related Information

- “limits.h” on page 32
- “unistd.h” on page 53
- “closedir() — Close a Directory” on page 194
- “getcwd() — Get Path Name of the Working Directory” on page 508
- “mkdir() — Make a Directory” on page 817
- “opendir() — Open a Directory” on page 877
- “readdir() — Read an Entry from a Directory” on page 1086
- “rewinddir() — Reposition a Directory Stream to the Beginning” on page 1141

## \_\_check\_resource\_auth\_np() — Determine Access to MVS Resources

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 5.2.2

### Format

```
#include <unistd.h>
```

```
int __check_resource_auth_np( char *principal_uuid,
                             char *cell_uuid,
                             char *userid,
                             char *security_class,
                             char *entity_name,
                             int access_type);
```

### General Description

The \_\_check\_resource\_auth\_np() function is used to check the access a user has to an MVS resource.

For authorization to use this function, the caller must have read permission to the BPX.SERVER Facility class, or if BPX.SERVER is not defined, the caller must be a superuser (UID=0).

The user identity can be specified in several forms. The identities are scanned in the order below, and the access check will be made with the first identity that is found:

- userid
- principal UUID and if known, a cell UUID
- caller's thread-level (task) security context, if one exists
- caller's process-level (address space) security context

**Note:** When no identity is specified by the caller and the caller's task has an ACEE created with pthread\_security\_np() for a SURROGATE (non-password) client, both the task and address space level ACEEs are used in determining the type of access permitted to a resource.

The parameters supported are:

*principal\_uuid*

Specifies a 36-byte principal UUID. A value of NULL indicates that no principal UUID is specified.

*cell\_uuid*

Specifies a 36-byte cell UUID. A value of NULL indicates that no cell UUID is specified.

*userid*

Specifies a userid. A value of NULL indicates that no userid is specified. The *userid* must be 1-8 characters in length.

*security\_class*

Specifies the name of a class of resources. The access check will be made on a resource in this security class. The *security\_class* must be 1-8 characters in length.

*entity\_name*

Specifies the name of a resource profile name. The access check will be made on the resource specified by the resource profile name. The *entity\_name* must be 1-246 characters in length.

*access* Specifies a numeric value that identifies the type of access to check for. Possible access values are:

**\_\_READ\_RESOURCE**

check if the specified user has read access to the resource.

**\_\_UPDATE\_RESOURCE**

check if the specified user has update access to the resource.

**\_\_CONTROL\_RESOURCE**

check if the specified user has control access to the resource.

**\_\_ALTER\_RESOURCE**

check if the specified user has alter access to the resource.

**Returned Value**

The `__check_resource_auth_np()` function returns 0 if successful. Otherwise, a -1 is returned and `errno` is set to indicate the error.

If any of the following conditions occurs, the `__check_resource_auth_np()` function returns -1 and sets `errno` to one of the following:

**EINVAL** One of the following errors was detected:

- `Aaccess_type` specified is undefined.
- `Userid` was not 1 to 8 characters in length.
- `Security_class` was not 1 to 8 characters in length.
- `Entity_name` was not 1 to 246 characters in length.

**ESRCH** One of the following errors was detected:

- No mapping exists between a UUID and `Userid`.
- The resource specified is not defined to the security product.
- The DCEUUIDS class is not active.
- The `userid` is not defined to the security product.

**EMVSERR** An MVS internal or environmental error occurred.

**EMVSSAF2ERR**

One of the following errors was detected:

- Received an unexpected return code for the security product.
- The security product detected an error in the input parameters.
- An internal error occurred in the security product.

**ENOSYS** One of the following errors was detected:

- No security product is installed on the system.
- The security product does not have support for this function.



- EPERM      One of the following errors was detected:
- The caller is not permitted to use this service.
  - Do not have the access\_type specified to the resource.
  - Not permitted in address spaces where a load from an unauthorized library has been performed.

**Related Information**

- “unistd.h” on page 53

## CheckSchEnv() — Check WLM Scheduling Environment

### Standards

Standards / Extensions	C or C++	Dependencies
	both	

### Format

```
#include <sys/__wlm.h>
int __CheckSchEnv(const char *sched_env,
                  const char *system_name);
```

### General Description

The CheckSchEnv() function provides the ability for an application to connect to check the WLM scheduling environment.

*sched_env	Points to a 16 byte character string that represents the WLM scheduling environment to be queried. If the environment name is less than 16 characters, the name should be right padded with blanks.
*sys_name	Points to a 8 bytes character string that represents the system name to be queried. If the system name is less than 8 characters, the name should be right padded with blanks.

### Returned Value

Upon successful completion CheckSchEnv() returns a zero. If the function is unsuccessful, -1 is returned and errno is set on to one of the following values:

EFAULT	An argument of this function contained an address that was not accessible to the caller.
EINVAL	An argument of this function contained an incorrect value.
EMVSWLMERROR	The WLM check scheduling environment failed. Use __errno2() to obtain the WLM service reason code for the failure.
EPERM	The calling thread's address space is not permitted to the BPX.WLMSEVER Facility class. The caller's address space must be permitted to the BPX.WLMSEVER Facility class if it is defined. If BPX.WLMSEVER is not defined, the calling process is not defined as a superuser (UID=0).
EMVSSAF2ERR	An error occurred in the security product.

### Related Information

- “sys/\_\_wlm.h” on page 51
- “ConnectServer() — Connect to WLM as a Server Manager” on page 219
- “ConnectWorkMgr() — Connect to WLM as a Work Manager” on page 221
- “ContinueWorkUnit() — Continue WLM Work Unit” on page 225
- “CreateWorkUnit() — Create WLM Work Unit” on page 236
- “DeleteWorkUnit() — Delete a WLM Work Unit” on page 272
- “DisconnectServer() — Disconnect from WLM Server” on page 276
- “JoinWorkUnit() — Join a WLM Work Unit” on page 723
- “LeaveWorkUnit() — Leave a WLM Work Unit” on page 742

- “QueryMetrics() — Query WLM System Information” on page 1070
- “QuerySchEnv() — Query WLM Scheduling Environment” on page 1072

# chmod() — Change the Mode of a File or Directory

## Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

## Format

```
#define _POSIX_SOURCE
#include <sys/stat.h>
```

```
int chmod(const char *pathname, mode_t mode);
```

## General Description

Changes the mode of the file or directory specified in *pathname*.

| To provide an ASCII input/output format for applications using this function, define  
| feature test macro `__LIBASCII` as described on page 22.

The *mode* argument is created with one of the following symbols defined in the `sys/stat.h` header file.

Any mode flags that are not defined will be turned off, and the function will be allowed to proceed.

- S\_ISUID** Privilege to set the user ID (UID) for execution. When this file is run through an `exec` function, the effective user ID of the process is set to the owner of the file. The process then has the same authority as the file owner, rather than the authority of the actual invoker.
- S\_ISGID** Privilege to set group ID (GID) for execution. When this file is run through an `exec` function, the effective group ID of the process is set to the group ID of the file. The process then has the same authority as the file owner, rather than the authority of the actual invoker.
- S\_ISVTX** The sticky bit indicating shared text. Keep loaded as an executable file in storage.
- S\_IRUSR** Read permission for the file owner.
- S\_IWUSR** Write permission for the file owner.
- S\_IXUSR** Search permission (for a directory) or execute permission (for a file) for the file owner.
- S\_IRWXU** Read, write, and search, or execute, for the file owner; **S\_IRWXG** is the bitwise inclusive OR of **S\_IRUSR**, **S\_IWUSR**, and **S\_IXUSR**.
- S\_IRGRP** Read permission for the file's group.
- S\_IWGRP** Write permission for the file's group.
- S\_IXGRP** Search permission (for a directory) or execute permission (for a file) for the file's group.

S_IRWXG	Read, write, and search or execute permission for the file's group. S_IRWXG is the bitwise inclusive <b>OR</b> of S_IRGRP, S_IWGRP, and S_IXGRP.
S_IROTH	Read permission for users other than the file owner.
S_IWOTH	Write permission for users other than the file owner.
S_IXOTH	Search permission for a directory, or execute permission for a file, for users other than the file owner.
S_IRWXO	Read, write, and search or execute permission for users other than the file owner. S_IRWXO is the bitwise inclusive OR of S_IROTH, S_IWOTH, and S_IXOTH.

### Special Behavior for XPG4.2

If a directory is writable and the mode bit S\_ISVTX is set on the directory, a process may remove or rename files within that directory only if one or more of the following is true:

- The effective user ID of the process is the same as that of the owner ID of the file.
- The effective user ID of the process is the same as that of the owner ID of the directory.
- The process has appropriate privileges.

A process can set mode bits only if the effective user ID of the process is the same as the file's owner or if the process has appropriate privileges (superuser authority). chmod() automatically clears the S\_ISGID bit in the file's mode bits if all these conditions are true:

- The calling process does not have appropriate privileges, that is, superuser authority (UID=0).
- The group ID of the file does not match the group ID or supplementary group IDs of the calling process.
- One or more of the S\_IXUSR, S\_IXGRP, or S\_IXOTH bits of the file mode are set to 1.

### Returned Value

If successful, chmod() marks for update the st\_ctime field of the file and returns 0. If unsuccessful, it returns -1 and sets errno to one of the following:

EACCES	The process does not have search permission on some component of the <i>pathname</i> prefix.
ELOOP	A loop exists in symbolic links. This error is issued if the number of symbolic links detected in the resolution of <i>pathname</i> is greater than POSIX_SYMLINK_MAX (a value defined in the limits.h header file).
ENAMETOOLONG	<i>pathname</i> is longer than PATH_MAX characters, or some component of <i>pathname</i> is longer than NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the <i>pathname</i> string substituted for a symbolic link exceeds PATH_MAX. The PATH_MAX and NAME_MAX values are determined using pathconf().

ENOENT	There is no file named <i>pathname</i> , or the <i>pathname</i> argument is an empty string.
ENOTDIR	Some component of the <i>pathname</i> prefix is not a directory.
EPERM	The effective user ID (UID) of the calling process does not match the owner of the file, and the calling process does not have appropriate privileges (superuser authority).
EROFS	<i>pathname</i> is on a read-only file system.

### Example CBC3BC11

```

/* CBC3BC11
   This example changes the permission from the file owner to the file's
   group.
*/
#define _POSIX_SOURCE
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

main() {
    char fn[] = "/temp.file";
    FILE *stream;
    struct stat info;

    if ((stream = fopen(fn, "w")) == NULL)
        perror("fopen() error");
    else {
        fclose(stream);
        stat(fn, &info);
        printf("original permissions were: %08x\n", info.st_mode);
        if (chmod(fn, S_IRWXU|S_IRWXG) != 0)
            perror("chmod() error");
        else {
            stat(fn, &info);
            printf("after chmod(), permissions are: %08x\n", info.st_mode);
        }
        unlink(fn);
    }
}

```

### Output

```

original permissions were: 030001b6
after chmod(), permissions are: 030001f8

```

### Related Information

- “sys/stat.h” on page 48
- “sys/types.h” on page 49
- “chown() — Change the Owner or Group of a File or Directory” on page 177
- “fchmod() — Change the Mode of a File or Directory by Descriptor” on page 344
- “mkdir() — Make a Directory” on page 817
- “mkfifo() — Make a FIFO Special File” on page 820
- “open() — Open a File” on page 872
- “stat() — Get File Information” on page 1404

# chown() — Change the Owner or Group of a File or Directory

## Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

## Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
int chown(const char *pathname, uid_t owner, gid_t group);
```

## General Description

Changes the owner or group (or both) of a file. *pathname* is the name of the file whose owner or group you want to change. *owner* is the user ID (UID) of the new owner of the file. *group* is the group ID (GID) of the new group for the file.

| To provide an ASCII input/output format for applications using this function, define  
| feature test macro \_\_LIBASCII as described on page 22.

If \_POSIX\_CHOWN\_RESTRICTED is defined in the unistd.h header file, a process can change the group of a file only if one of these is true:

1. The process has appropriate privileges.
2. Or all of the following are true:
  - a. The effective user ID of the process is equal to the user ID of the file owner.
  - b. The *owner* argument is equal to the user ID of the file owner or (uid\_t)-1,
  - c. The *group* argument is either the effective group ID or a supplementary group ID of the calling process.

If *pathname* is a regular file and one or more of the S\_IXUSR, S\_IXGRP, or S\_IXOTH bits of the file mode are set, chown() clears the set-user-ID (S\_ISUID) and set-group-ID (S\_ISGID) bits of the file mode and returns successfully.

If *pathname* is not a regular file and one or more of the S\_IXUSR, S\_IXGRP, or S\_IXOTH bits of the file mode are set, chown() clears the set-user-ID (S\_ISUID) and set-group-ID (S\_ISGID) bits of the file.

When chown() completes successfully, it marks the st\_ctime field of the file to be updated.

## Special Behavior for XPG4.2

If *owner* or *group* is specified as (uid\_t)-1 or (gid\_t)-1 respectively, the corresponding ID of the file is unchanged.

## Returned Value

If successful, `chown()` updates the owner, group, and change time for the file and returns 0. If unsuccessful, it returns `-1` and sets `errno` to one of the following:

EACCES	The process does not have search permission on some component of the <i>pathname</i> prefix.
EINVAL	<i>owner</i> or <i>group</i> is not a valid user ID (UID) or group ID (GID).
ELOOP	A loop exists in symbolic links. This error is issued if the number of symbolic links detected in the resolution of <i>pathname</i> is greater than <code>POSIX_SYMLLOOP</code> (a value defined in the <code>limits.h</code> header file).
ENAMETOOLONG	<i>pathname</i> is longer than <code>PATH_MAX</code> characters, or some component of <i>pathname</i> is longer than <code>NAME_MAX</code> characters while <code>_POSIX_NO_TRUNC</code> is in effect. For symbolic links, the length of the <i>pathname</i> string substituted for a symbolic link exceeds <code>PATH_MAX</code> . The <code>PATH_MAX</code> and <code>NAME_MAX</code> values can be determined using <code>pathconf()</code> .
ENOENT	There is no file named <i>pathname</i> , or the <i>pathname</i> argument is an empty string.
ENOTDIR	Some component of the <i>pathname</i> prefix is not a directory.
EPERM	The effective user ID of the calling process does not match the owner of the file, or the calling process does not have appropriate privileges, that is, superuser authority (UID=0).
EROFS	<i>pathname</i> is on a read-only file system.

## Special Behavior for XPG4.2

Adds the following values:

EIO	An I/O error occurred while reading or writing to the file system.
EINTR	The <code>chown()</code> function was interrupted by a signal which was caught.

## Example

### CBC3BC12

```

/* CBC3BC12
   This example changes the owner and group of a file.
   */
#define _POSIX_SOURCE
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <stdio.h>

main() {
    char fn[] = "./temp.file";
    FILE *stream;
    struct stat info;

    if ((stream = fopen(fn, "w")) == NULL)
        perror("fopen() error");
    else {
        fclose(stream);
        stat(fn, &info);
        printf("original owner was %d and group was %d\n", info.st_uid,
              info.st_gid);
    }
}

```



```

    if (chown(fn, 25, 0) != 0)
        perror("chown() error");
    else {
        stat(fn, &info);
        printf("after chown(), owner is %d and group is %d\n",
               info.st_uid, info.st_gid);
    }
    unlink(fn);
}
}

```

### Output

original owner was 0 and group was 0  
after chown(), owner is 25 and group is 0

### Related Information

- “limits.h” on page 32
- “unistd.h” on page 53
- “chmod() — Change the Mode of a File or Directory” on page 174
- “fchown() — Change the Owner or Group by File Descriptor” on page 346
- “fstat() — Get Status Information about a File” on page 479
- “lstat() — Get Status of File or Symbolic Link” on page 778
- “stat() — Get File Information” on page 1404

## chpriority() — Change the Scheduling Priority of a Process

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 5.2.2

### Format

```
#define _OPEN_SOURCE 2
#include <sys/resource.h>
```

```
int chpriority (int which, id_t who, int prioritytype, int priority);
```

### General Description

The `chpriority()` function changes the scheduling priority of a process, process group or user.

Processes are specified by the values of the *which* and *who* arguments. The *which* argument may be one of the following values: `PRIO_PROCESS`, `PRIO_PGRP`, or `PRIO_USER`, indicating that the *who* argument is to be interpreted as a process ID, a process group ID, or a user ID, respectively. A 0 (zero) value for the *who* argument specifies the current process, process group or user ID.

If more than one process is specified, the `chpriority()` function changes the priorities of all of the specified processes.

The default priority is 0; negative priorities cause more favorable scheduling. The range of legal priority values is -20 to 19. If the `CPRIO_ABSOLUTE` value is specified for the *prioritytype* argument and the priority value specified to `chpriority()` is less than the system's lowest supported priority value, the system's lowest supported value is used; if it is greater than the system's highest supported value, the system's highest supported value is used. If the `CPRIO_RELATIVE` value is specified on the *prioritytype* argument, request for values above or below the legal limits result in the priority value being set to the corresponding limit.

The changing of a process' scheduling priority value has the equivalent effect of a process' nice value, since they both represent the process' relative CPU priority. For example, changing one's scheduling priority value via the `chpriority()` function to its maximum value (19) has the equivalent effect of increasing one's nice value to its maximum value  $2^{\{NZERO\}}-1$ , and will be reflected on the `nice()`, `getpriority()`, `chpriority()`, and `setpriority()` functions.

Only a process with appropriate privilege can lower its priority. In addition to lowering the priority value, a process with appropriate privilege has the ability to change the priority of any process regardless of the process' saved set-user-ID value.

### Returned Value

If successful, the `chpriority()` function returns a value of 0 (zero).

If unsuccessful, the `chpriority()` function returns a -1, and `errno` is set to indicate the error.

The following are the possible values of `errno`:

ESRCH	No process could be located using the <i>which</i> and <i>who</i> argument values specified.
EINVAL	The value of the <i>which</i> argument was not recognized, or the value of the <i>who</i> argument is not a valid process ID, process group ID or user ID, or the value of the <i>prioritytype</i> argument was not recognized.
EPERM	A process was located, but the save set-user-ID of the executing process does not match the saved set-user-ID of the process whose priority is to be changed.
EACCES	The priority is being changed to a lower value and the current process does not have the appropriate privilege.
ENOSYS	The system does not support this function.

### Related Information

- “getpriority() — Get Process Scheduling Priority” on page 578
- “nice() — Change Priority of a Process” on page 864
- “setpriority() — Set Process Scheduling Priority” on page 1257

## chroot() — Change Root Directory

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE
#include <unistd.h>
```

```
int chroot(const char *path);
```

### General Description

The *path* argument points to a pathname naming a directory. The *chroot()* function causes the named directory to become the root directory, that is the starting point for path searches for pathnames beginning with /. The process' working directory is unaffected by *chroot()*. Only a superuser can request *chroot()*.

The dot-dot entry in the root directory is interpreted to mean the root directory. Thus, dot-dot cannot be used to access files outside the subtree rooted at the root directory.

**Note:** The *chroot()* function, is an obsolete element and is to be deleted from the next revision of the C standard.

### Returned Value

If successful, *chroot()* changes the root directory, and returns 0. If unsuccessful, *chroot()* returns -1 and sets *errno* to one of the following:

- EACCES** Search permission is denied for a component of *path*
- ELOOP** A loop exists in symbolic links. This error is issued if the number of symbolic links detected in the resolution of pathname is greater than `POSIX_SYMLLOOP` (a value defined in the `limits.h` header file).
- ENAMETOOLONG** Pathname is longer than `PATH_MAX` characters, or some component of pathname is longer than `NAME_MAX` characters while `_POSIX_NO_TRUNC` is in effect. For symbolic links, the length of the pathname string substituted for a symbolic link exceeds `PATH_MAX`. The `PATH_MAX` and `NAME_MAX` values are determined using `pathconf()`.
- ENOENT** A component of *path* does not name an existing directory or *path* is an empty string.
- ENOTDIR** A component of the *path* name is not a directory.
- EPERM** The effective user ID does not have appropriate privileges.

**Related Information**

- “unistd.h” on page 53
- “chdir() — Change the Working Directory” on page 167
- “closedir() — Close a Directory” on page 194
- “mkdir() — Make a Directory” on page 817
- “opendir() — Open a Directory” on page 877

## clearenv() — Clear Environment Variables

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a Language Environment	both	

### Format

#### POSIX - C only

```
#define _POSIX1_SOURCE 2
#include <env.h>
int clearenv(void);
```

#### Non-POSIX

```
#include <stdlib.h>

int clearenv(void);
```

### General Description

Clears all environment variables from the environment table and frees the associated storage.

clearenv() also resets all behavior modified by OS/390 C/C++ specific environment variables back to their defaults. For example, if a binary file was opened, then it would support seeking by byte offsets, regardless of record format. If the file is a Variable Record format MVS DASD file, then clearing the environment variable causes seeking by encoded values the next time it is opened.

To avoid infringing on the user's name-space, the non-POSIX version of this function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

For details about environment variables, see “Using Environment Variables” in the *OS/390 C/C++ Programming Guide*.

### Special Behavior for POSIX C

clearenv() can change the value of the pointer `environ`. Therefore, a copy of that pointer made before a call to `clearenv()` may no longer be valid after the call to `clearenv()`. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information about using POSIX support.

## Returned Value

Returns 0 if it is successful. If unsuccessful, it returns a nonzero value and sets `errno` to `ENOMEM`, which indicates that the process requires more space than is available.

## Example CBC3BC13

```
/* CBC3BC13
   This OS/390 C example needs to be run with POSIX(ON).
   It clears the process environment variable list.
*/
#define _POSIX1_SOURCE 2
#include <env.h>
#include <stdio.h>

extern char **environ;

int count_env() {
    int num;

    for (num=0; environ[num] != NULL; num++);
    return num;
}

main() {
    printf("before clearenv(), there are %d environment variables\n",
        count_env());
    if (clearenv() != 0)
        perror("clearenv() error");
    else {
        printf("after clearenv(), there are %d environment variables\n",
            count_env());
        setenv("var1", "value1", 1);
        setenv("var-two", "Value Two", 1);
        printf("after setenv()'s, there are %d environment variables\n",
            count_env());
        if (clearenv() != 0)
            perror("clearenv() error");
        else
            printf("after clearenv(), there are %d environment variables\n",
                count_env());
    }
}
```

## Output

```
before clearenv(), there are 9 environment variables
after clearenv(), there are 0 environment variables
after setenv()'s, there are 2 environment variables
after clearenv(), there are 0 environment variables
```

## CBC3BC14

```
/* CBC3BC14
   This example is for a non-POSIX environment, and thus will work under
   C++/MVS.
*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
```

```

char *x;

/* set 3 environment variables to "Y" */
setenv("_EDC_ANSI_OPEN_DEFAULT","Y",1);
setenv("_EDC_BYTE_SEEK","Y",1);
setenv("_EDC_COMPAT","3",1);

/* query the setting of _EDC_BYTE_SEEK */
x = getenv("_EDC_BYTE_SEEK");

if (x != NULL)
    printf("_EDC_BYTE_SEEK = %s\n",x);
else
    printf("_EDC_BYTE_SEEK is undefined\n");

/* clear the environment variable table */
clearenv();

/* query the setting of _EDC_BYTE_SEEK */
x = getenv("_EDC_BYTE_SEEK");

if (x != NULL)
    printf("_EDC_BYTE_SEEK = %s\n",x);
else
    printf("_EDC_BYTE_SEEK is undefined\n");
}

```

**Output**

```

_EDC_BYTE_SEEK = Y
_EDC_BYTE_SEEK is undefined

```

**Related Information**

- “Using Environmental Variables” in the *OS/390 C/C++ Programming Guide*
- “env.h” on page 25
- “stdlib.h” on page 45
- “getenv() — Get Value of Environment Variables” on page 515
- “\_\_getenv() — Get an Environment Variable” on page 517
- “setenv() — Add, Delete, and Change Environment Variables” on page 1215
- “putenv() — Change or Add an Environment Variable” on page 1054



## clearerr() — Reset Error and End-of-File

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
void clearerr(FILE *stream);
```

### General Description

Resets the error indicator and EOF indicator for the stream that *stream* points to. Generally, the indicators for a stream remain set until your program calls `clearerr()` or `rewind()`.

### Returned Value

Returns no value.

### Example CBC3BC15

```
/* CBC3BC15
   This example reads a data stream and then checks that a read error has
   not occurred.
*/
#include <stdio.h>

int main(void)
{
    char string[100];
    FILE *stream;
    int eofvalue;

    stream = fopen("myfile.dat", "r");

    /* scan an input stream until an end-of-file character is read */
    while (!feof(stream))
        fscanf(stream,"%s",STRING[0]);

    /* print EOF value: will be nonzero */
    eofvalue=feof(stream);
    printf("feof value=%i\n",eofvalue);

    /* print EOF value-after clearerr, will be equal to zero */
    clearerr(stream);
    eofvalue=feof(stream);
    printf("feof value=%i\n",eofvalue);
}
```

### **Related Information**

- “stdio.h” on page 43
- “feof() — Test End-of-File Indicator” on page 365
- “ferror() — Test for Read/Write Errors” on page 367
- “fseek() — Change File Position” on page 474
- “rewind() — Set File Position to Beginning of File” on page 1139

## clock() — Determine Processor Time

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <time.h>
```

```
clock_t clock(void);
```

### General Description

Approximates the processor time used by the program, since the beginning of an implementation-defined time period that is related to the program invocation. To measure the time spent in a program, call the clock() function at the start of the program, and subtract its returned value from the value returned by subsequent calls to clock(). Then, to obtain the time in seconds, divide the value returned by clock() by CLOCKS\_PER\_SEC.

If you use the system() function in your program, do not rely on clock() for program timing, because calls to system() may reset the clock.

In a multithread POSIX C application, if you are creating threads with a function that is based on a POSIX.4a draft standard, the clock() function is thread scoped.

### Returned Value

Returns the calculated time if it is available and if the time can be represented. Otherwise, clock() returns the value (clock\_t)-1. The clock() function may return -1 when running with STIMER\_REAL TQE present on MVS/ESA Version 3 Release 1 Modification 2 (or earlier) system.

### Special Behavior for XPG4

If \_XOPEN\_SOURCE or \_XOPEN\_SOURCE\_EXTENDED are defined when your application is compiled, CLOCKS\_PER\_SEC is defined as 1000000. Also, in this case, the following C/370 pragma in the <time.h> header is used to compile your application:

```
#pragma map ( clock(), "@@0CLCK")
```

Because of this pragma, when your application executes, it will attempt to access an XPG4 version of clock() which returns a clock\_t value in units of 1000000 CLOCKS\_PER\_SEC. The XPG4 version of clock() is only available if POSIX(ON) is specified for execution of your application.

If \_XOPEN\_SOURCE or \_XOPEN\_SOURCE\_EXTENDED are defined when you compile your program AND your application is run with POSIX(OFF), clock() will return (clock\_t)-1.

If neither \_XOPEN\_SOURCE or \_XOPEN\_SOURCE\_EXTENDED are defined when you compile your application, the historical C/370 value of

CLOCKS\_PER\_SEC will be used and clock() calls in your application will be mapped to the historical C/370 version of clock() which returns a clock\_t value in historical C/370 CLOCKS\_PER\_SEC units whether your application executes with POSIX(ON) or POSIX(OFF).

**Example**

```
/* This example prints the time elapsed since the program was invoked. */
#include <time.h>
#include <stdio.h>

double time1, timedif; /* use doubles to show small values */

int main(void)
{
    time1 = (double) clock();          /* get initial time */
    time1 = time1 / CLOCKS_PER_SEC;    /* in seconds */
    :
    /* call clock a second time */
    timedif = ( ((double) clock()) / CLOCKS_PER_SEC) - time1;
    printf("The elapsed time is %f seconds\n", timedif);
}
```

**Related Information**

- “time.h” on page 51
- “time() — Determine Current Time” on page 1570

## close() — Close a File

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
int close(int fildes);
```

### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
```

```
int close(int socket);
```

### Berkeley Sockets

```
#define _OE_SOCKETS
#include <unistd.h>
```

```
int close(int socket);
```

### General Description

Closes a file descriptor, *fildes*. This frees the file descriptor to be returned by future `open()` calls and other calls that create file descriptors. The *fildes* argument must represent a hierarchical file system (HFS) file.

When the last open file descriptor for a file is closed, the file itself is closed. If the file's link count is 0 at that time, its space is freed and the file becomes inaccessible.

When the last open file descriptor for a pipe or FIFO file is closed, any data remaining in the pipe or FIFO file is discarded. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information about using POSIX support.

`close()` unlocks (removes) all outstanding record locks that a process has on the associated file.

### Behavior for Sockets

`close()` call shuts down the socket associated with the socket descriptor *socket*, and frees resources allocated to the socket. If *socket* refers to an open TCP connection, the connection is closed. If a stream socket is closed when there is input data queued, the TCP connection is reset rather than being cleanly closed.

Parameter	Description
<i>socket</i>	The descriptor of the socket to be closed.

**Note:** All sockets should be closed before the end of your process. You should issue a `shutdown()` call before you issue a `close()` call for a socket.

For `AF_INET` stream sockets (`SOCK_STREAM`) using `SO_LINGER` socket option, the socket does not immediately end if data is still present when a `close` is issued. The following structure is used to set or unset this option, and it can be found in `sys/socket.h`.

```
struct linger {
    int l_onoff;      /* zero=off, nonzero=on */
    int l_linger;     /* time is seconds to linger */
};
```

If the `l_onoff` switch is nonzero, the system attempts to deliver any unsent messages. If a linger time is specified, the system waits for *n* seconds before flushing the data and terminating the socket.

For `AF_UNIX`, when closing sockets that were bound, you should also use `unlink()` to delete the file created at `bind()` time.

### Special Behavior for XPG4.2

If a STREAMS-based *fildev* is closed and the calling process was previously registered to receive a `SIGPOL` signal for events associated with that STREAM, the calling process will be unregistered for events associated with the STREAM. The last `close()` for a STREAM causes the STREAM associated with *fildev* to be dismantled. If `O_NONBLOCK` is not set and there have been no signals posted for the STREAM, and if there is data on the module's write queue, `close()` waits for an unspecified time (for each module and driver) for any output to drain before dismantling the STREAM. The time delay can be changed via an `l_setcltime` `ioctl()` request. If the `O_NONBLOCK` flag is set, or if there are any pending signals, `close()` does not wait for output to drain, and dismantles the STREAM immediately.

**Note:** OS/390 UNIX services do not supply any STREAMS devices or pseudodevices. See “`open()` — Open a File” on page 872 for more information.

If *fildev* refers to the master side of a pseudoterminal, a `SIGHUP` signal is sent to the process group, if any, for which the slave side of the pseudoterminal is the controlling terminal.

If *fildev* refers to the slave side of a pseudoterminal, a zero-length message will be sent to the master.

If *fildev* refers to a socket, `close()` causes the socket to be destroyed. If the socket is connection-oriented and the `SO_LINGER` option is set for the socket and the socket has untransmitted data, then `close()` will block for up to the current linger interval until all data is transmitted.

### Returned Value

If successful, `close()` returns 0. If unsuccessful, it returns `-1` and sets `errno` to one of the following:

<code>EAGAIN</code>	The call did not complete because the specified socket descriptor is currently being used by another thread in the same process.
---------------------	--

For example, in a multithreaded environment, `close()` fails and returns

EAGAIN when one thread is blocked in a read() or select() call on a given file or socket descriptor and another thread issues a simultaneous close() call for the same descriptor.

EBADF *fd* is not a valid open file descriptor, or the *socket* parameter is not a valid socket descriptor.

EBUSY The file cannot be closed because it is blocked.

EINTR close() was interrupted by a signal. The file may or may not be closed.

EIO **Added for XPG4.2:** An I/O error occurred while reading from or writing to the file system.

ENXIO *fd* does not exist. The minor number for the file is incorrect.

### Example

```
#define _POSIX_SOURCE
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

main() {
    int fd;
    char out[20]="Test string";
    if ((fd = creat("./myfile", S_IRUSR | S_IWUSR)) < 0)
        perror("creat error");
    else {
        if (write(fd, out, strlen(out)+1) == -1)
            perror("write() error");

        if (fd = 0) perror("write() error");
        close(fd);
    }
}
```

### Related Information

- “unistd.h” on page 53
- “accept() — Accept a New Connection on a Socket” on page 75
- “creat() — Create a New File or Rewrite an Existing One” on page 233
- “dup() — Duplicate an Open File Descriptor” on page 288
- “exec Functions” on page 322
- “fclose() — Close File” on page 348
- “fcntl() — Control Open File Descriptors” on page 350
- “fork() — Create a New Process” on page 422
- “getsockopt() — Get the Options Associated with a Socket” on page 606
- “open() — Open a File” on page 872
- “pipe() — Create an Unnamed Pipe” on page 907
- “setsockopt() — Set Options Associated with a Socket” on page 1270
- “socket() — Create a Socket” on page 1371
- “shutdown() — Shut Down All or Part of a Duplex Connection” on page 1293
- “unlink() — Remove a Directory Entry” on page 1660

## closedir() — Close a Directory

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <dirent.h>
```

```
int closedir(DIR *dir);
```

### General Description

Closes the directory indicated by *dir*. It frees the buffer that *readdir()* uses when reading the directory stream.

### Returned Value

If successful, *closedir()* returns 0. If unsuccessful, it returns *-1* and sets *errno* to one of the following:

**EBADF**        *dir* does not refer to an open directory stream.

**EINTR**        *closedir()* was interrupted by a signal. The directory may or may not be closed.

### Example

#### CBC3BC18

```
/* CBC3BC18
   This example closes a directory.
*/
#define _POSIX_SOURCE
#include <dirent.h>
#include <sys/types.h>
#include <stdio.h>

main() {
    DIR *dir;
    struct dirent *entry;
    int count;

    if ((dir = opendir("/")) == NULL)
        perror("opendir() error");
    else {
        count = 0;
        while ((entry = readdir(dir)) != NULL) {
            printf("directory entry %03d: %s\n", ++count, entry->d_name);
        }
        closedir(dir);
    }
}
```

### Output



```
directory entry 001: .  
directory entry 002: ..  
directory entry 003: bin  
directory entry 004: dev  
directory entry 005: etc  
directory entry 006: lib  
directory entry 007: tmp  
directory entry 008: u  
directory entry 009: usr
```

### Related Information

- “dirent.h” on page 24
- “stdio.h” on page 43
- “sys/types.h” on page 49
- “opendir() — Open a Directory” on page 877
- “readdir() — Read an Entry from a Directory” on page 1086
- “rewinddir() — Reposition a Directory Stream to the Beginning” on page 1141
- “seekdir() — Set Position of Directory Stream” on page 1158
- “telldir() — Current Location of Directory Stream” on page 1557

## closelog() — Close the Control Log

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <syslog.h>
```

```
void closelog(void);
```

### General Description

The `closelog()` function closes the log file.

### Returned Value

Neither accepts an input nor returns a result. The system control log is closed for this process.

No errors are defined.

### Related Information

- “syslog.h” on page 47
- “openlog() — Open the System Control Log” on page 882
- “setlogmask() — Set the Mask for the Control Log” on page 1250
- “syslog() — Send a Message to the Control Log” on page 1486

## clrmemf() — Clear Memory Files

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <stdio.h>
```

```
int clrmemf(int level);
```

### General Description

Removes memory files created by the current program and any program that was called using a non-POSIX `system()` call. `clrmemf()` can remove memory files regardless of whether they are open or not.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use `LANGLVL(EXTENDED)`.

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with `LANGLVL(EXTENDED)`. When you use `LANGLVL(EXTENDED)` any relevant information in the header is also exposed.

The argument *level* indicates which memory files are to be removed. The level can be one of the following:

<code>__LOWER</code>	Removes memory files that were created in other programs and called from this program via <code>system()</code> .
<code>__CURRENT</code>	Removes only the memory files created at the current level.
<code>__CURRENT_LOWER</code>	Removes all the memory files created by the current program and by all the programs called at the current level.

### Special Behavior for Multiple Shared PIC1 C Environments

Only files created by the C environment from which the `clrmemf()` function is called will be cleared.

### Returned Value

`clrmemf()` returns 0 if successful and a nonzero value otherwise.

### Example

```
/*
   In this example, when Program2 calls clrmemf(__CURRENT) only
   A3.FILE and A4.FILE will be removed.
*/
/***** Program1 *****/
:
fp1 = fopen ("A1.FILE", "w,type=memory(hiperspace)");
fp2 = fopen ("A2.FILE", "w,type=memory(hiperspace)");
```

```
    system("Program2");
    :
/***** Program2 *****/
    :
    fp3 = fopen("A3.FILE","w,type=memory");
    fp4 = fopen("A4.FILE","w,type=memory");
    system("Program3");
    :
    clrmemf(__CURRENT);
    :
/***** Program3 *****/
    :
    fp5 = fopen("A5.FILE","w,type=memory");
    fp6 = fopen("A6.FILE","w,type=memory");
    :
```

### **Related Information**

- “stdio.h” on page 43
- “fopen() — Open a File” on page 417
- “system() — Execute a Command” on page 1488

## \_\_cnvblk() — Convert Block

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#define _XOPEN_SOURCE
#include <unistd.h>
```

```
void __cnvblk(char bits[8], char bytes[64], int flag);
```

### General Description

The \_\_cnvblk() function maps an 8 character array, *bits*, of bits to or from a 64 character array, *bytes*, of bytes depending on the value of *flag*.

If the value of *flag* is 0, \_\_cnvblk() sets all bytes in the *bytes* array to 0x00 for which the corresponding bits in the *bits* array have value 0, and it sets all bytes in the *bytes* array to 0x01 for which the corresponding bits in the *bits* array have value 1.

If the value of *flag* is **not** 0, \_\_cnvblk() sets all bits in the *bits* array to 0 for which corresponding bytes in the *bytes* array have value 0x00, and it sets all bits in the *bits* array to 1 for which the corresponding bytes in the *bytes* array have value 0x01.

This function may be used to prepare input to setkey() or encrypt() functions and to map results back to 8 bit characters.

### Returned Value

If the value of *flag* is nonzero, \_\_cnvblk() sets errno = EINVAL if the value of any byte in the array *bytes* is not 0x00 or 0x01. Otherwise, \_\_cnvblk() functions without error checking.

**Note:** Because \_\_cnvblk() does not return a value, applications wishing to check for errors should set errno to 0, call \_\_cnvblk(), then test errno and, if it is nonzero, assume an error has occurred.

### Related Information

- “unistd.h” on page 53

## collequiv() — Return a List of Equivalent Collating Elements

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <collate.h>
```

```
int collequiv(collel_t c, collel_t **list);
```

### General Description

Finds all the collating elements whose primary weight is the same as the primary weight of *c*. It then updates the list to point to the first element of the array in which all the found elements are stored. The list of elements is valid until the next call to `setlocale()`, with categories `LC_ALL`, `LC_COLLATE`, or `LC_CTYPE`.

Another call to `collequiv()` may override the current list.

For information about the effect of `setlocale()` and `locale.h`, see “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*.

### Returned Value

The function returns the number of collating elements found. Otherwise, if the value of *c* is not in the valid range of collating elements in the current locale, the function returns `-1`.

### Notes:

- If the collating element passed is specified with the weight of `IGNORE` in the `LC_COLLATE` category, the list returned will contain all the characters specified as `IGNORE`.
- The list will only contain characters defined in the `charmap` file in the current locale.

### Example CBC3BC22

```
/* CBC3BC22
   This example prints the collating elements that have an equivalent
   weight as the collating element passed in argv[1].
*/
#include "stdio.h"
#include "locale.h"
#include "collate.h"
#include "stdlib.h"
#include "wctype.h"
#include "wchar.h"

main(int argc, char *argv[]) {
    collel_t e, *rp;
    int i;

    setlocale(LC_ALL, "");
    if ((e = strtocoll(argv[1])) == (collel_t)-1) {
        printf("'s' collating element not defined\n", argv[1]);
    }
}
```

```

        exit(1);
    }
    if ((i = collequiv(e, &rp)) == -1) {
        printf("Invalid collating element '%s'\n", argv[1]);
        exit(1);
    }
    for (; i-- > 0; rp++) {
        if (ismccollet(*rp))
            printf("%s ", colltostr(*rp));
        else if (iswprint(*rp))
            printf("%lc ", *rp);
        else
            printf("%x ", *rp);
    }
}

```

## Related Information

- “collate.h” on page 23
- “cclass() — Return Characters in a Character Class” on page 149
- “collorder() — Return List of Collating Elements” on page 202
- “collrange() — Calculate the Range List of Collating Elements” on page 204
- “colltostr() — Return a String for a Collating Element” on page 206
- “getmccoll() — Get Next Collating Element from String” on page 554
- “getwmccoll() — Get Next Collating Element from Wide String” on page 632
- “ismccollet() — Identify a Multi-Character Collating Element” on page 710
- “maxcoll() — Return Maximum Collating Element” on page 788
- “strtcoll() — Return Collating Element for String” on page 1454

## collorder() — Return List of Collating Elements

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <collate.h>
```

```
int      collorder(coll_el_t **list);
```

### General Description

Finds the number of collating elements in the collate order list and sets a pointer to the list. The list returned is valid until another call to `setlocale()`.

### Notes:

- Collating elements specified with the weight of IGNORE in the LC\_COLLATE category are defined as having the lowest weight.
- The list will only contain characters defined in the charmap file in the current locale.

### Example

#### CBC3BC23

```
/* CBC3BC23
   This example creates a list of all the collating elements using the
   collorder() function.
*/
#include <stdio.h>
#include <locale.h>
#include <collate.h>
#include <wchar.h>
#include <wctype.h>

main(int argc, char *argv[]) {
    coll_el_t e, *rp;
    int i;

    setlocale(LC_ALL, "");
    i = collorder(&rp);
    for (; i-- > 0; rp++) {
        if (ismccoll_el(*rp))
            printf("%s ", colltostr(*rp));
        else if (iswprint(*rp))
            printf("%lc ", *rp);
        else
            printf("%x ", *rp);
    }
}
```



## Related Information

- “collate.h” on page 23
- “cclass() — Return Characters in a Character Class” on page 149
- “collequiv() — Return a List of Equivalent Collating Elements” on page 200
- “collrange() — Calculate the Range List of Collating Elements” on page 204
- “colltostr() — Return a String for a Collating Element” on page 206
- “getmccoll() — Get Next Collating Element from String” on page 554
- “getwmccoll() — Get Next Collating Element from Wide String” on page 632
- “ismccollel() — Identify a Multi-Character Collating Element” on page 710
- “maxcoll() — Return Maximum Collating Element” on page 788
- “strtcoll() — Return Collating Element for String” on page 1454

## collrange() — Calculate the Range List of Collating Elements

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <collate.h>
```

```
int collrange(collel_t start, collel_t end, collel_t **list);
```

### General Description

Finds a list of collating elements whose primary weights are between the *start* and *end* points, inclusive. The number returned is the number of elements in the list, whose pointer is returned.

This value will be zero if the end point collates earlier than the *start* point. The list returned is valid until the next call to `setlocale()`.

### Returned Value

The number returned will be `-1` if either *start* or *end* are out of range.

### Notes:

- Collating elements specified with the weight of `IGNORE` in the `LC_COLLATE` category are defined having the lowest weight. Therefore, such elements can only be specified as the starting collating element.
- The list will only contain characters defined in the charmap file in the current locale.

### Example

#### CBC3BC24

```
/* CBC3BC24
   This example prints the collating elements in the range between the
   start and end points passed in argv{1} and argv{2}, using
   the collrange() function.
*/
#include <stdio.h>
#include <locale.h>
#include <collate.h>
#include <stdlib.h>
#include <wctype.h>
#include <wchar.h>

main(int argc, char *argv[]) {
    collel_t s, e, *rp;
    int i;

    setlocale(LC_ALL, "");
    if ((s = strtocoll(argv[1])) == (collel_t)-1) {
        printf("%s' collating element not defined\n", argv[1]);
        exit(1);
    }
    if ((e = strtocoll(argv[2])) == (collel_t)-1) {
        printf("%s' collating element not defined\n", argv[2]);
        exit(1);
    }
}
```

```

    }
    if ((i = collrange(s, e, &rp)) == -1) {
        printf("Invalid range for '%s' to '%s'\n", argv[1], argv[2]);
        exit(1);
    }
    for (; i-- > 0; rp++) {
        if (ismccollet(*rp))
            printf("%s ", colltostr(*rp));
        else if (iswprint(*rp))
            printf("%lc ", *rp);
        else
            printf("%x ", *rp);
    }
}

```

## Related Information

- “cclass() — Return Characters in a Character Class” on page 149
- “collequiv() — Return a List of Equivalent Collating Elements” on page 200
- “collorder() — Return List of Collating Elements” on page 202
- “colltostr() — Return a String for a Collating Element” on page 206
- “getmccoll() — Get Next Collating Element from String” on page 554
- “getwmccoll() — Get Next Collating Element from Wide String” on page 632
- “ismccollet() — Identify a Multi-Character Collating Element” on page 710
- “maxcoll() — Return Maximum Collating Element” on page 788
- “strtcoll() — Return Collating Element for String” on page 1454

## colltostr() — Return a String for a Collating Element

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <collate.h>
```

```
char *colltostr(collel_t c);
```

### General Description

Converts *c* to the string of the collating element. The `colltostr()` function is the inverse of `strtcoll()`.

An application program can use the returned array from `collrange()` or `collequiv()`, calling `ismccollel()` on each element, only calling `colltostr()` if `ismccollel()` is true for the element. The string returned is valid until another call to `setlocale()`.

### Returned Value

If a value representing a single character or a value that is not in range is passed, the returned value is `NULL`.

### Example

#### CBC3BC25

```
/* CBC3BC25
   This example prints all the collating elements in the collating
   sequence, using the colltostr function to get the string for the
   multi-character collating elements.
*/
#include <collate.h>
#include <locale.h>
#include <stdio.h>
#include <wchar.h>
#include <wctype.h>

main(int argc, char *argv[]) {
    collel_t e, *rp;
    int i;

    setlocale(LC_ALL, "");
    i = collorder(&rp);
    for (; i-- > 0; rp++) {
        if (ismccollel(*rp))
            printf('%s' ", colltostr(*rp));
        else if (iswprint(*rp))
            printf('%lc' ", *rp);
        else
            printf('%x' ", *rp);
    }
}
```

## Related Information

- “collate.h” on page 23
- “cclass() — Return Characters in a Character Class” on page 149
- “collequiv() — Return a List of Equivalent Collating Elements” on page 200
- “collorder() — Return List of Collating Elements” on page 202
- “collrange() — Calculate the Range List of Collating Elements” on page 204
- “colltostr() — Return a String for a Collating Element” on page 206
- “getmccoll() — Get Next Collating Element from String” on page 554
- “getwmccoll() — Get Next Collating Element from Wide String” on page 632
- “ismccollel() — Identify a Multi-Character Collating Element” on page 710
- “maxcoll() — Return Maximum Collating Element” on page 788
- “strtcoll() — Return Collating Element for String” on page 1454

## compile() — Compile Regular Expression

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define INIT declarations
#define GETC() getc_code
#define PEEK() peek_code
#define UNGETC() ungetc_code
#define RETURN(ptr) return_code
#define ERROR(val) error_code
```

```
#define _XOPEN_SOURCE
#include <regex.h>
```

```
char *compile(char *instring, char *expbuf,
              const char *endbuf, int eof);
```

### General Description

The compile() function takes as input a simple regular expression and produces a compiled expression that can be used with the step() and advance() functions.

The first parameter *instring* is never used explicitly by compile(). It is a pointer to a character string defining a source regular expression. It is useful for programs that pass down different pointers to input characters. Programs which invoke functions to input characters or have characters in an external array can pass down **(char \*)0** for this parameter.

*expbuf* is a pointer to the place where the compiled regular expression will be placed.

*endbuf* points to one more than the highest address where the compiled regular expression may be placed. If the compiled expression cannot fit in (*endbuf-expbuf*) bytes, a call to ERROR(50) is made. (See "Returned Value" below.)

*eof* is the character which marks the end of the regular expression.

The OS/390 UNIX services implementation of the compile() function does **not** accept internationalized simple expressions as input. Internationalized simple expressions (e.g. `[[=c=]]` (an equivalence class)) may yield unpredictable results.

Programs must have the following five macros declared before the **#include <regex.h>** statement. The macros GETC(), PEEKC() and UNGETC() operate on the regular expression given as input to compile().

GETC()	This macro returns the value of the next character (byte) in the regular expression pattern. Successive calls to GETC() should return successive characters of the regular expression.
--------	--

PEEK()	This macro returns the next character (byte) in the regular expression pattern. Immediate successive calls to PEEK() should return the same byte, which should also be the next character returned by GETC().
UNGETC( <i>c</i> )	This macro causes the argument <i>c</i> to be returned by the next call to GETC(). No more than one character is ever needed and this character is guaranteed to be the last character read by GETC(). The value of the macro UNGETC() is always ignored.
RETURN( <i>ptr</i> )	This macro is used on normal exit of the compile() function. The value of the argument <i>ptr</i> is a pointer to the character after the last character of the compiled regular expression.
ERROR( <i>val</i> )	This macro is the abnormal return from compile(). The argument <i>val</i> is an error number. (See "Returned Value" below for meanings.) This call should never return.

**Notes:**

1. OS/390 UNIX services do not provide any default macros if the above user macros are not provided.
2. Each program that includes the <regexp.h> must have a **#define** statement for INIT. It is used for dependent declarations and initializations. For example, it can be used to set a variable to point to the beginning of the regular expression so that this variable can be used in the declarations for GETC(), PEEK(), and UNGETC().
3. The external variables *cirf*, *sed*, and *nbra* are reserved.
4. The application must provide the proper serialization for the compile(), step(), and advance() functions if they are run under a multi-threaded environment.

**Simple Regular Expressions**

A Simple Regular Expression (SRE) specifies a set of character strings. The simplest form of regular expression is a string of characters with no special meaning. A small set of special characters, known as metacharacters, do have special meaning when encountered in patterns.

Expression	Meaning
<i>c</i>	The character <i>c</i> where <i>c</i> is not a special character.
<i>\c</i>	The character <i>c</i> where <i>c</i> is any special character. For example, <i>a\.e</i> is equivalent to <i>a.e</i> .
<i>^</i>	The beginning of the string being compared
<i>\$</i>	The dollar symbol matches the end of the string.
<i>.</i>	The period symbol matches any one character.
[ <i>string</i> ]	A string within square brackets specifies any of the characters in <i>string</i> . Thus, [ <i>abc</i> ], if compared to other strings, would match any which contained <i>a</i> , <i>b</i> , or <i>c</i> .  The <i>]</i> (right bracket) can be used alone within a pair of brackets, but only if it immediately follows either the opening left bracket or if it immediately follows [ <i>^</i> .

Ranges may be specified as *c–c*. The hyphen symbol, within square brackets, means "through". It fills in the intervening characters according to the collating sequence. For example, `[a–z]` is equivalent to `[abc...xyz]`. If the end character in the range is lower in collating sequence to the start character, then only the range start and range end characters are accepted in the search pattern. For example, `[9–1]` is equivalent to `[91]`. Note that ranges in Simple Regular Expressions are only valid if the `LC_COLLATE` category is set to the C locale.

The `–` (hyphen) can be used by itself, but only if it is the first or last character in the expression. For example, the expression `[]a–f]` matches either the `]` or one of the characters *a* through *f*.

`[^string]`

The caret symbol, when inside square brackets, negates the characters within the square brackets. Thus, `[^abc]`, if compared to other strings, would *fail* to match any which contains even one *a*, *b*, or *c*.

**Note:** Characters `.`, `*`, `[`, and `\` (period, asterisk, left square bracket, and backslash, respectively) have special meaning, except when they appear within square brackets (`[]`), or are preceded by `\`.

`*`

The asterisk symbol indicates 0 or more of any preceding characters. For example, `(a*e)` will match any of the following: *e*, *ae*, *aae*, *aaae*, .... The longest leftmost match is chosen.

`rx`

The occurrence of regular expression *r* followed by the occurrence of regular expression *x*.

`\{m\}` `\{m,u\}` `\{m,u\}`

Integer values enclosed in `\{ \}` indicate the number of times to apply the preceding regular expression. *m* is the minimum number and *u* is the maximum number. *u* must be less than 256. If you specify only *m*, it indicates the exact number of times to apply the regular expression.

`\{m,u\}` is equivalent to `\{m,255\}`. They both match *m* or more occurrences of the expression. The `*` (asterisk) operation is equivalent to `\{0,u\}`.

The maximum number of occurrences is matched.

`\(r)`

The regular expression *r*. The `\(` and `\)` sequences are ignored.

`\n`

When `\n` (where  $1 \leq n \leq 9$ ) appears in a concatenated regular expression, it stands for the regular expression *x*, where *x* is the *n*th regular expression enclosed in `\(` and `\)` sequences that appeared earlier in the concatenated regular expression. For example, in the pattern `\(c\)onc\(ate\)n2`, the `\2` is equivalent to *ate*, giving *concatenate*.

The character `^` at the beginning of an expression permits a successful match only immediately after a newline or at the beginning of each of the string to which a match is to be applied. The character `$` at the end of an expression requires a trailing newline.

**Note:** The `compile()` function is physically embedded in the **regex.h** header. This header will be protected from multiple invocations just like other *c* headers.



**Note:** The `compile()` function is provided for historical reasons. New applications should use the new functions `fnmatch()`, `glob()`, `regcomp()` and `regexexec()`, which provide full internationalized regular expression functionality compatible with ISO POSIX.2 standard.

## Returned Value

If successful, `compile()` exits using the user-provided macro `RETURN(ptr)`. The value of the argument *ptr* is a pointer to the character after the last character of the compiled regular expression.

If unsuccessful, `compile()` exits using the user-provided macro `ERROR(val)`. The argument *val* is an error number identifying the error. The following error numbers are defined:

errcode	Description String
11	Range endpoint too large
16	Bad number
25	<code>\digit</code> out of range
36	Illegal or missing delimiter
41	No remembered search string
42	<code>\( \)</code> imbalance
43	Too many <code>\(</code>
44	More than two numbers given in <code>\{ \}</code>
45	<code>}</code> expected after <code>\</code>
46	First number exceeds second in <code>\{ \}</code>
49	<code>[ ]</code> imbalance
50	Regular expression overflow

## Related Information

- “`regex.h`” on page 40
- “`advance()` — Pattern Match Given a Compiled Regular Expression” on page 88
- “`fnmatch()` — Match Filename or Pathname” on page 415
- “`glob()` — Generate Pathnames Matching a Pattern” on page 636
- “`regcomp()` — Compile Regular Expression” on page 1120
- “`regexexec()` — Execute Compiled Regular Expression” on page 1126
- “`step()` — Pattern Match with Regular Expression” on page 1411

## confstr() — Get Configurable Variables

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <unistd.h>
```

```
size_t confstr(int name, char *buf, size_t len);
```

### General Description

The `confstr()` function provides a method for applications to get configuration-defined string values. Its use and purpose are similar to the `sysconf()` function, but it is used where string values rather than numeric values are returned.

The *name* argument represents the system variable to be queried. It may be any one of the following symbolic constants, defined in `<unistd.h>`:

`_CS_PATH` Request the value of the **PATH** environment variable that will find all standard utilities.

OS/390 UNIX services use the following constant:

`_CS_SHELL` Request the fully qualified name of the default shell.

If the *len* argument is not zero, and if the *name* argument has a configuration-defined value, `confstr()` copies that value into the buffer pointed to by the *buf* argument. If the value to be returned is longer than *len* bytes, including the . terminating null, then `confstr()` truncates the string to *len*-1 bytes and null-terminates the results. The application can detect that the string was truncated by comparing the value returned by `confstr()` with *len*.

If the *len* argument is zero, and the *buf* argument is a null pointer, then `confstr()` still returns the integer value defined below, but does not return a string. If the *len* argument is zero, but the *buf* argument is not a null pointer the results are unspecified.

### Returned Value

If *name* has a configuration-defined value, the `confstr()` function returns the size of the buffer that would be needed to hold the entire configuration-defined string value. If this return value is greater than *len*, the string returned in *buf* is truncated.

If *name* is invalid, `confstr()` returns zero and returns an error value in `errno`.

If *name* does not have a configuration-defined value, `confstr()` returns zero and leaves `errno` unchanged.

The following are the possible values of `errno`:

`EINVAL` The value of the *name* argument is invalid.

**Related Information**

- “unistd.h” on page 53
- “fpathconf() — Determine Configurable Path Name Variables” on page 427
- “pathconf() — Determine Configurable Path Name Variables” on page 896
- “sysconf() — Determine System Configuration Options” on page 1483

## connect() — Connect a Socket

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int connect(int socket, const struct sockaddr *address, size_t address_len);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>

int connect(int socket, struct sockaddr *address, int address_len);
```

### General Description

For stream sockets, the `connect()` call attempts to establish a connection between two sockets. For datagram sockets, the `connect()` call specifies the peer for a socket. The *socket* parameter is the socket used to originate the connection request. The `connect()` call performs two tasks when called for a stream socket. First, it completes the binding necessary for a stream socket (in case it has not been previously bound using the `bind()` call). Second, it attempts to make a connection to another socket.

**Note:** For the X/Open socket function, the *socket* description applies to *socket*, *address* to *address*, and *address\_len* to *address\_len*. **const** is added to **struct sockaddr**.

Parameter	Description
<i>socket</i>	The socket descriptor.
<i>name</i>	The pointer to a socket address structure containing the address of the socket to which a connection will be attempted.
<i>namelen</i>	The size of the socket address pointed to by <i>name</i> in bytes.

The `connect()` call on a stream socket is used by the client application to establish a connection to a server. The server must have a passive open pending. A server that is using sockets must successfully call `bind()` and `listen()` before a connection can be accepted by the server with `accept()`. Otherwise, `connect()` returns `-1` and the error code is set to `ECONNREFUSED`.

If *socket* is in blocking mode, the `connect()` call blocks the caller until the connection is set up, or until an error is received. If the socket is in nonblocking mode, `connect()` returns `-1` with the error code set to `EINPROGRESS` to indicate that the connection has been initiated but is not yet complete (if no errors occurred). The caller can test the completion of the connection setup by calling `select()` and testing for the ability to write to the socket.

When called for a datagram socket, `connect()` specifies the peer with which this socket is associated. This gives the application the ability to use data transfer calls reserved for sockets that are in the connected state. In this case, `read()`, `write()`, `readv()`, `writew()`, `send()`, and `recv()` calls are then available in addition to `sendto()`, `recvfrom()`, `sendmsg()`, and `recvmsg()` calls. Stream sockets can call `connect()` only once, but datagram sockets can call `connect()` multiple times to change their association. Datagram sockets can dissolve their association by connecting to an incorrect address, such as the null address (all fields zeroed).

The *name* parameter is a pointer to a buffer containing the name of the peer to which the application needs to connect. The *namelen* parameter is the size, in bytes, of the buffer pointed to by *name*.

### Servers in the AF\_INET Domain

If the server is in the AF\_INET domain, the format of the name buffer is expected to be **sockaddr\_in**, as defined in the include file **netinet/in.h**.

```
struct in_addr
{
    ip_addr_t s_addr;
};

struct sockaddr_in {
    unsigned char  sin_len;
    unsigned char  sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    unsigned char  sin_zero[8];
};
```

The *sin\_family* field must be set to AF\_INET. The *sin\_port* field is set to the port to which the server is bound. It must be specified in network byte order. The *sin\_zero* field is not used and must be set to all zeros.

### Servers in the AF\_UNIX Domain

If the server is in the AF\_UNIX domain, the format of the name buffer is expected to be **sockaddr\_un**, as defined in the include file **un.h**.

```
struct sockaddr_un {
    unsigned char  sun_len;
    unsigned char  sun_family;
    char  sun_path[108];    /* pathname */
};
```

The *sun\_family* field is set to AF\_UNIX.

The *sun\_path* field contains the null-terminated pathname, and *sun\_len* contains the length of the pathname.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

**Returned Value**

The value 0 indicates success; the value -1 indicates an error. The value of the error code indicates the specific error.

Error Code	Description
EADDRNOTAVAIL	The specified address is not available from the local machine.
EINVAL	The address family is not supported.
EALREADY	The socket descriptor <i>socket</i> is marked nonblocking, and a previous connection attempt has not completed.
EBADF	The <i>socket</i> parameter is not a valid socket descriptor.
ECONNREFUSED	The connection request was rejected by the destination host.
EFAULT	Using <i>name</i> and <i>namelen</i> would result in an attempt to copy the address into a portion of the caller's address space to which data cannot be written.
EINPROGRESS	O_NONBLOCK is set for the file descriptor for the socket, and the connection cannot be immediately established. The connection will be established asynchronously. The EINPROGRESS value does not indicate an error condition.
EINTR	The attempt to establish a connection was interrupted by delivery of a signal that was caught. The connection will be established asynchronously.
EINVAL	The <i>namelen</i> parameter is not a valid length.
EISCONN	The socket descriptor <i>socket</i> is already connected.
EIO	There has been a network or a transport failure.
ENETUNREACH	The network cannot be reached from this host.
ENOTSOCK	The descriptor refers to a file, not a socket.
EOPNOTSUPP	The <i>socket</i> parameter is not of type SOCK_STREAM.
EPERM	connect() caller was attempting to extract a user's identity and the caller's process was not verified to be a server. To be server-verified, the caller's process must have permission to the BPX.SERVER profile (or superuser and BPX.SERVER is undefined) and have called either the __pwd() or pthread_security () services prior to calling connect() to propagate identity.
EPROTOTYPE	The protocol is the wrong type for this socket.
ETIMEDOUT	The connection establishment timed out before a connection was made.

The following are for AF\_UNIX only:

Error Code	Description
ENOTDIR	A component of the path prefix of the pathname in <i>address</i> is not a directory.
ENAMETOOLONG	A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX characters.

EACCES	Search permission is denied for a component of the path prefix, or write access to the named socket is denied.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating the pathname in <i>address</i> .
ENOENT	A component of the pathname does not name an existing file or the pathname is an empty string.

### Example

The following are examples of the `connect()` call. The Internet address and port must be in network byte order. To put the port into network byte order, the `htons()` utility routine is called to convert a short integer from host byte order to network byte order. The *address* field is set using another utility routine, `inet_addr()`, which takes a character string representing the dotted-decimal address of an interface and returns the binary Internet address representation in network byte order. Finally, it is a good idea to zero the structure before using it to ensure that the name requested does not set any reserved fields. These examples could be used to connect to the servers shown in the examples listed with the call, “`bind()` — Bind a Name to a Socket” on page 128.

```
int s;
struct sockaddr_in inet_server;
struct sockaddr_un unix_server;
int rc;
int connect(int s, struct sockaddr *name, int namelen);

/* Connect to server bound to a specific interface in the Internet domain */
/* make sure the sin_zero field is cleared */
memset(&inet_server, 0, sizeof(inet_server));
inet_server.sin_family = AF_INET;
inet_server.sin_addr = inet_addr("129.5.24.1"); /* specific interface */
inet_server.sin_port = htons(1024);
:
rc = connect(s, (struct sockaddr *) &inet_server, sizeof(inet_server));

/* Connect to a server bound to a name in the UNIX domain */
/* make sure the unix_addr, unix_port, unix_nodeid fields are cleared */
memset(&unix_server, 0, sizeof(unix_server));
unix_server.sun_family = AF_UNIX;
strncpy(unix_server.sun_path, "mysocket");
unix_server.sun_len = sizeof(unix_server.sun_len);
strncpy(mvsservername.sunix_name, "APPL", 8);
:
rc = connect(s, (struct sockaddr *) &unix_server, sizeof(unix_server));
```

### Related Information

- “`accept()` — Accept a New Connection on a Socket” on page 75
- “`bind()` — Bind a Name to a Socket” on page 128
- “`htons()` — Translate an Unsigned Short Integer into Network Byte Order” on page 649
- “`inet_addr()` — Translate an Internet Address into Network Byte Order” on page 661
- “`listen()` — Prepare the Server for Incoming Client Requests” on page 752
- “`select()` — Monitor Activity on Files/Sockets and Message Queues” on page 1159

## connect

- “selectex() — Monitor Activity on Files/Sockets and Message Queues” on page 1166
- “socket() — Create a Socket” on page 1371



## ConnectServer() — Connect to WLM as a Server Manager

### Standards

Standards / Extensions	C or C++	Dependencies
	both	

### Format

```
#include <sys/_wlm.h>
unsigned long ConnectServer(const char *subsystype,
                           const char *subsysname,
                           const char *applenv,
                           int *paralleleu);
```

### General Description

The ConnectServer function provides the ability for an application to connect to WLM as a WLM server manager to preform WLM server manager functions.

<i>*subsys<del>type</del></i>	Points to a null terminated character string containing the generic subsystem type (CICS, IMS, WEB, etc.). This is the primary category under which WLM classification rules are grouped. The character string can be up to 4 bytes in length.
<i>*subsys<del>name</del></i>	Points to a null terminated character string containing the subsystem name used for classifying work requests. The character string can be up to 8 bytes in length.
<i>*applenv</i>	Points to a null terminated character string that contains the name of the application environment under which work requests are processed. The character string can be up to 32 bytes in length.
<i>*paralle<del>leu</del></i>	Points to an integer which contains the maximum number of tasks within the address space which will be created to process concurrent work requests.

### Returned Value

Upon successful completion ConnectServer() returns a non-zero value representing a WLM connect token. If the function is unsuccessful, -1 is returned and errno is set to one of the following values:

EFAULT	An argument of this function contained an address that was not accessable to the caller.
EINVAL	An argument of this function contained an incorrect value.
EMVSWLMERROR	The WLM connect failed. Use __errno2() to obtain the WLM service reason code for the failure.
EPERM	The calling thread's address space is not permitted to the BPX.WLMSEVER Facility class. The caller's address space must be permitted to the BPX.WLMSEVER Facility class if it is defined. If BPX.WLMSEVER is not defined, the calling process is not defined as a superuser (UID=0).
EMVSSAF2ERR	An error occured in the security product.

### Related Information

- “sys/\_\_\_wlm.h” on page 51
- “CheckSchEnv() — Check WLM Scheduling Environment” on page 172
- “ConnectWorkMgr() — Connect to WLM as a Work Manager” on page 221
- “ContinueWorkUnit() — Continue WLM Work Unit” on page 225
- “CreateWorkUnit() — Create WLM Work Unit” on page 236
- “DeleteWorkUnit() — Delete a WLM Work Unit” on page 272
- “DisconnectServer() — Disconnect from WLM Server” on page 276
- “JoinWorkUnit() — Join a WLM Work Unit” on page 723
- “LeaveWorkUnit() — Leave a WLM Work Unit” on page 742
- “QueryMetrics() — Query WLM System Information” on page 1070
- “QuerySchEnv() — Query WLM Scheduling Environment” on page 1072

## ConnectWorkMgr() — Connect to WLM as a Work Manager

### Standards

Standards / Extensions	C or C++	Dependencies
	both	

### Format

```
#include <sys/_wlm.h>
unsigned long ConnectWorkMgr(const char *subsystype,
                             const char *subsysname);
```

### General Description

The ConnectServer function provides the ability for an application to connect to WLM as a WLM work manager to preform WLM work manager functions.

<i>*subsystype</i>	Points to a null terminated character string containing the generic subsystem type (CICS, IMS, WEB, etc.). This is the primary category under which WLM classification rules are grouped. The character string can be up to 4 bytes in length.
<i>*subsysname</i>	Points to a null terminated character string containing the subsystem name used for classifying work requests. The character string can be up to 8 bytes in length.

### Returned Value

Upon successful completion ConnectServer() returns a non-zero value representing a WLM connect token. If the function is unsuccessful, -1 is returned and errno is to one of the following values:

EFAULT	An argument of this function contained an address that was not accessible to the caller.
EINVAL	An argument of this function contained an incorrect value.
EMVSWLMERROR	The WLM connect failed. Use __errno2() to obtain the WLM service reason code for the failure.
EPERM	The calling thread's address space is not permitted to the BPX.WLMSEVER Facility class. The caller's address space must be permitted to the BPX.WLMSEVER Facility class if it is defined. If BPX.WLMSEVER is not defined, the calling process is not defined as a superuser (UID=0).
EMVSSAF2ERR	An error occurred in the security product.

### Related Information

- “sys/\_wlm.h” on page 51
- “CheckSchEnv() — Check WLM Scheduling Environment” on page 172
- “ConnectServer() — Connect to WLM as a Server Manager” on page 219
- “ConnectWorkMgr() — Connect to WLM as a Work Manager”
- “CreateWorkUnit() — Create WLM Work Unit” on page 236
- “DeleteWorkUnit() — Delete a WLM Work Unit” on page 272
- “DisconnectServer() — Disconnect from WLM Server” on page 276
- “JoinWorkUnit() — Join a WLM Work Unit” on page 723
- “LeaveWorkUnit() — Leave a WLM Work Unit” on page 742

- “QueryMetrics() — Query WLM System Information” on page 1070
- “QuerySchEnv() — Query WLM Scheduling Environment” on page 1072

# \_\_console() — Console Communication Services

## Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R2

## Format

```
#include <sys/__messag.h>

int __console(struct __cons_msg *cons, char *modstr, int *concmd);
```

## General Description

The \_\_console() function is used to communicate with the operator's console. The \_\_console() function allows users to send messages to the operator's console and to wait on a modify/stop request from the console.

The parameters supported are:

- cons      If the argument is not NULL, it points to a structure specifying the message that is to be sent to the operator's console. If the argument is NULL then no message is sent.
- modstr    Specifies the string where the \_\_console() function returns the data entered at the operator's console. If modstr is not NULL the invoker will wait until an operator MODIFY's the invoking job and specifies 'APPL=' parameter. See *OS/390 MVS System Commands* for more information on the MODIFY console command. If the argument is NULL then the \_\_console() function will not wait on operator console commands.
- concmd    If a console command was issued against the invoking job, the \_\_console() function will set the command type. Valid types are, \_CC\_modify (function received a modify request) and \_CC\_stop (function received a stop request).

The cons structure is defined in the <sys/\_\_messag.h> header and has the following format.

```
struct __cons_msg {
    short __reserved0;
    char __reserved1[2];
    union {
        struct {
            int __msg_length;
            char *__msg;
            char __reserved2[8];
        } __f1;
    } __format;
};
```

__reserved0	Reserved for future use.
__reserved1[2]	Reserved for future use.
__format.__f1.__msg_length	Length of message, not including the NULL terminator.

<code>__format.__f1.__msg</code>	A character string containing the message to be sent to the operator console.
<code>__format.__f1.__reserved2[8]</code>	Reserved for future use.

**Note:** The length of the message must be between 1 and 17850 characters for invokers with appropriate privileges, and between 1 and 17780 for invokers without appropriate privileges. The number of lines written to the console is limited to 255. In the case of an unprivileged user, one of those lines is used for the message ID and the invoker's login name. If the message length is exceeded, no lines are written and the service returns an EINVAL. If the number of lines is exceeded, the service returns an EINTR, but the first 255 lines are written to the console.

### Returned Value

The `__console()` function returns 0 if successful. Otherwise, it returns -1 and sets `errno` to indicate the error.

`errno` values defined for `__console()`.

<b><i>errno</i></b>	<b><i>Description</i></b>
EINVAL	The cons structure contains errors.
EINTR	The <code>__console()</code> function was interrupted by a signal.
EFAULT	One of the following errors was detected: <ul style="list-style-type: none"> <li>• All or part of the cons structure is not addressable by the caller.</li> <li>• All or part of the modstr string is not addressable by the caller.</li> </ul>
EMVSERR	OS/390 environmental or internal error has occurred.

### Related Information

- “sys/\_\_messag.h” on page 48

## ContinueWorkUnit() — Continue WLM Work Unit

### Standards

Standards / Extensions	C or C++	Dependencies
	both	

### Format

```
#include <sys/__wlm.h>
int ContinueWorkUnit(wlmetok_t *enclavetoken);
```

### General Description

The ContinueWorkUnit function provides the ability for an application to create a WLM work unit that represents a continuation of the work unit associated with the current home address space.

*\*enclavetoken* Points to a data field of type wlmetok\_t where the ContinueWorkUnit() function is to return the WLM work unit enclave token.

### Returned Value

Upon successful completion ContinueWorkUnit() returns a zero. If the function is unsuccessful, -1 is returned and errno is set to one of the following values:

EFAULT	An argument of this function contained an address that was not accessible to the caller.
EINVAL	An argument of this function contained an incorrect value.
EMVSWLMERROR	The WLM create enclave failed. Use __errno2() to obtain the WLM service reason code for the failure.
EPERM	The calling thread's address space is not permitted to the BPX.WLMSEVER Facility class. The caller's address space must be permitted to the BPX.WLMSEVER Facility class if it is defined. If BPX.WLMSEVER is not defined, the calling process is not defined as a superuser (UID=0).
EMVSSAF2ERR	An error occurred in the security product.

### Related Information

- “sys/\_\_wlm.h” on page 51
- “CheckSchEnv() — Check WLM Scheduling Environment” on page 172
- “ConnectServer() — Connect to WLM as a Server Manager” on page 219
- “ConnectWorkMgr() — Connect to WLM as a Work Manager” on page 221
- “CreateWorkUnit() — Create WLM Work Unit” on page 236
- “DeleteWorkUnit() — Delete a WLM Work Unit” on page 272
- “DisconnectServer() — Disconnect from WLM Server” on page 276
- “JoinWorkUnit() — Join a WLM Work Unit” on page 723
- “LeaveWorkUnit() — Leave a WLM Work Unit” on page 742
- “QueryMetrics() — Query WLM System Information” on page 1070
- “QuerySchEnv() — Query WLM Scheduling Environment” on page 1072

## \_\_convert\_id\_np() — Convert Between DCE UUID and Userid

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 5.2.2

### Format

```
#include <unistd.h>
```

```
int __convert_id_np(int function_code,
                   char *principal_uuid,
                   char *cell_uuid,
                   char *userid);
```

### General Description

The \_\_convert\_id\_np() function is used to retrieve the DCE UUID associated with a userid or the userid associated with a DCE UUID.

This function is intended for DCE servers which process requests from multiple clients. For example, DCE RPC requests from clients are identified by a DCE UUID only. This function enables servers to extract the userid of the requester.

The parameters supported are:

<i>function_code</i>	Identifies whether extracting a userid or UUID. Possible function codes are:  __GET_USERID Return the userid associated with the specified UUIDs.  __GET_UUID Return the UUIDs associated with the specified userid.
<i>principal_uuid</i>	When __GET_USERID is specified, <i>principal_uuid</i> contains the UUID of the user for the specified userid. When __GET_UUID is specified, <i>principal_uuid</i> returns the extracted UUID for the userid specified. The caller must provide a 36-byte field for the returned <i>principal_uuid</i> .
<i>cell_uuid</i>	When __GET_USERID is specified, <i>cell_uuid</i> should contain the cell UUID if known. If not known, <i>cell_uuid</i> must be NULL. When __GET_UUID is specified, <i>cell_uuid</i> will return the extracted cell UUID, if it is defined for the specified userid. The caller must provide a 36-byte field for the returned <i>cell_uuid</i> .
<i>userid</i>	When __GET_USERID is specified, <i>userid</i> will return the extracted userid for the specified UUID. The caller must provide a 9 byte field for the returned userid. When __GET_UUID is specified, <i>userid</i> contains the userid for whom the UUID should be extracted. The userid must be 1 to 8 characters in length.



## Returned Value

The \_\_convert\_id\_np() function returns 0 if successful. Otherwise, a -1 is returned and errno is set to indicate the error.

If any of the following conditions occurs, the \_\_convert\_id\_np() function returns -1 and sets errno to one of the following:

- EINVAL      One of the following errors was detected:
- functioncode specified is undefined.
  - \_\_GET\_UUID was specified and userid is not in the range 1 to 8 characters long.
  - \_\_GET\_USERID was specified and userid was not 9 character long
- ESRCH      One of the following errors was detected:
- No mapping exists between a UUID and Userid.
  - No mapping exists between a Userid and UUID.
  - The DCEUUIDS class is not active.
  - \_\_GET\_UUID was specified and no cell UUID is defined for the userid.
  - The userid is not defined to the security product.
- EMVSERR    An MVS environmental or internal error occurred.
- EMVSSAF2ERR
- One of the following errors was detected:
- Received an unexpected return code for the security product.
  - The security product detected an error in the input parameters.
  - An internal error occurred in the security product.
- ENOSYS      One of the following errors was detected:
- No security product is installed on the system.
  - The security product does not have support for this function.

## Related Information

- “unistd.h” on page 53

## copysign() — Copy Sign

### Standards

Standards/Extensions	C or C++	Dependencies;
	both	OS/390 V2R6

### Format

```
#include <math.h>
#include <float.h>
```

```
double copysign (x, y)
double x, y;
```

### General Description

The `copysign()` function returns the `x` parameter with the same sign as the `y` parameter.

Parameters	Description
------------	-------------

<code>x</code>	Specifies a long double floating-point value
----------------	--

<code>y</code>	Specifies a long double floating-point value
----------------	--

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

Long double floating-point value.

### Related Information

- “`nextafter()` — Next Representable Double Float” on page 860
- “`scalb()` — Load Exponent” on page 1154
- “`logb()` — Unbiased Exponent” on page 763
- “`ilogb()` — Integer Unbiased Exponent” on page 659

## cos() — Calculate Cosine

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double cos(double x);
```

### General Description

Calculates the cosine of  $x$ . The value  $x$  is expressed in radians.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

Returns the calculated value. If  $x$  is outside prescribed limits, the value is not calculated. Instead, `cos()` returns the value zero and sets the `errno` to `ERANGE`. If the correct value would cause an underflow, zero is returned and the value of the macro `ERANGE` is stored in `errno`.

### Special Behavior for XPG4.2

The following error is added:

**EDOM**        The argument exceeded an internal limit for the function (approximately  $2^{50}$ ).

### Example

#### CBC3BC26

```
/* CBC3BC26
   This example calculates y to be the cosine of x.
*/
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, y;

    x = 7.2;
    y = cos(x);

    printf("cos( %lf ) = %lf\n", x, y);
}
```

### Output

```
cos( 7.200000 ) = 0.608351
```

**Related Information**

- “math.h” on page 35
- “acos() — Calculate Arccosine” on page 85
- “acosh() — Hyperbolic Arccosine” on page 87
- “asin() — Calculate Arcsine” on page 108
- “asinh() — Hyperbolic Arcsine” on page 110
- “atan() - atan2() — Calculate Arctangent” on page 113
- “atanh() — Hyperbolic Arctangent” on page 115
- “cosh() — Calculate Hyperbolic Cosine” on page 231
- “sin() — Calculate Sine” on page 1360
- “sinh() — Calculate Hyperbolic Sine” on page 1362
- “tan() — Calculate Tangent” on page 1500
- “tanh() — Calculate Hyperbolic Tangent” on page 1502

## cosh() — Calculate Hyperbolic Cosine

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double cosh(double x);
```

### General Description

Calculates the hyperbolic cosine of  $x$ . The value  $x$  is expressed in radians.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

If the result overflows, `cosh()` returns the value `HUGE_VAL` and sets `errno` to `ERANGE`.

### Example

#### CBC3BC27

```
/* CBC3BC27
   This example calculates y to be the hyperbolic cosine of x.
*/
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x,y;

    x = 7.2;
    y = cosh(x);

    printf("cosh( %lf ) = %lf\n", x, y);
}
```

### Output

```
cosh( 7.200000 ) = 669.715755
```

### Related Information

- “math.h” on page 35
- “acos() — Calculate Arccosine” on page 85
- “acosh() — Hyperbolic Arccosine” on page 87
- “asin() — Calculate Arcsine” on page 108
- “asinh() — Hyperbolic Arcsine” on page 110
- “atan() - atan2() — Calculate Arctangent” on page 113
- “atanh() — Hyperbolic Arctangent” on page 115

## cosh

- “sinh() — Calculate Hyperbolic Sine” on page 1362
- “tan() — Calculate Tangent” on page 1500
- “tanh() — Calculate Hyperbolic Tangent” on page 1502

## creat() — Create a New File or Rewrite an Existing One

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <fcntl.h>
```

```
int creat(const char *pathname, mode_t mode);
```

### General Description

The function call: `creat(pathname, mode)` is equivalent to the call:

```
open(pathname, O_CREAT|O_WRONLY|O_TRUNC, mode);
```

Thus the file named by *pathname* is created, unless it already exists. The file is then opened for writing only, and is truncated to zero length. See “`open()` — Open a File” on page 872 for further information.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

The *mode* argument specifies the file permission bits to be used in creating the file. Here is a list of symbols that can be used for a mode:

<code>S_ISUID</code>	Privilege to set the user ID (UID) for execution. When this file is run through an exec function, the effective user ID of the process is set to the owner of the file, so that the process has the same authority as the file owner rather than the authority of the actual invoker.
<code>S_ISGID</code>	Privilege to set group ID (GID) for execution. When this file is run through an exec function, the effective group ID of the process is set to the group ID of the file, so that the process has the same authority as the file owner rather than the authority of the actual invoker.
<code>S_ISVTX</code>	Indicates shared text. Keep loaded as an executable file in storage.
<code>S_IRUSR</code>	Read permission for the file owner.
<code>S_IWUSR</code>	Write permission for the file owner.
<code>S_IXUSR</code>	Search permission (for a directory) or execute permission (for a file) for the file owner.
<code>S_IRWXU</code>	Read, write, and search, or execute, for the file owner; <code>S_IRWXG</code> is the bitwise inclusive OR of <code>S_IRUSR</code> , <code>S_IWUSR</code> , and <code>S_IXUSR</code> .
<code>S_IRGRP</code>	Read permission for the file's group.
<code>S_IWGRP</code>	Write permission for the file's group.
<code>S_IXGRP</code>	Search permission (for a directory) or execute permission (for a file) for the file's group.

S_IRWXG	Read, write, and search, or execute permission for the file's group. S_IRWXG is the bitwise inclusive <b>OR</b> of S_IRGRP, S_IWGRP, and S_IXGRP.
S_IROTH	Read permission for users other than the file owner.
S_IWOTH	Write permission for users other than the file owner.
S_IXOTH	Search permission for a directory, or execute permission for a file, for users other than the file owner.
S_IRWXO	Read, write, and search, or execute permission for users other than the file owner. S_IRWXO is the bitwise inclusive OR of S_IROTH, S_IWOTH, and S_IXOTH.

### Returned Value

If `creat()` succeeds, it returns a file descriptor for the open file. If unsuccessful, `creat()` returns `-1` and sets `errno` to one of:

#### EACCES

- The process did not have search permission on a component in *pathname*.
- The file exists but the process did not have appropriate permissions to open the file in the way specified by the flags.
- The file does not exist, and the process does not have write permission on the directory where the file is to be created.
- `O_TRUNC` was specified, but the process does not have write permission on the file.

EINTR *open()* was interrupted by a signal.

EISDIR *pathname* is a directory, and *options* specifies write or read/write access.

ELOOP A loop exists in symbolic links. This error is issued if the number of symbolic links detected in the resolution of *pathname* is greater than `POSIX_SYMLINK_MAX`.

EMFILE The process has reached the maximum number of file descriptors it can have open.

#### ENAMETOOLONG

*pathname* is longer than `PATH_MAX` characters or some component of *pathname* is longer than `NAME_MAX` characters while `_POSIX_NO_TRUNC` is in effect. For symbolic links, this error occurs if the length of a *pathname* string substituted for a symbolic link in the *pathname* argument exceeds `PATH_MAX`. The `PATH_MAX` and `NAME_MAX` values can be determined with `pathconf()`.

ENFILE The system has reached the maximum number of file descriptors it can have open.

ENOENT `O_CREAT` is specified, and either the path prefix does not exist or the *pathname* argument is an empty string.

ENOSPC The directory or file system intended to hold a new file has insufficient space.



ENOTDIR     A component of *pathname* is not a directory.  
 EROFS       *pathname* is on a read-only file system.

### Example CBC3BC28

```
/* CBC3BC28
   This example creates a file.
   */
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

main() {
    char fn[]="creat.file", text[]="This is a test";
    int fd;

    if ((fd = creat(fn, S_IRUSR | S_IWUSR)) < 0)
        perror("creat() error");
    else {
        write(fd, text, strlen(text));
        close(fd);
        unlink(fn);
    }
}
```

### Related Information

- “fcntl.h” on page 28
- “sys/stat.h” on page 48
- “sys/types.h” on page 49
- “close() — Close a File” on page 191
- “open() — Open a File” on page 872
- “unlink() — Remove a Directory Entry” on page 1660.

## CreateWorkUnit() — Create WLM Work Unit

### Standards

Standards / Extensions	C or C++	Dependencies
	both	

### Format

```
#include <sys/_wlm.h>
int CreateWorkUnit(wlmetok_t *enclavetoken,
                  server_classify_t classify,
                  char *arrival_time,
                  char *func_name);
```

### General Description

The CreateWorkUnit function provides the ability for an application to create a WLM work unit.

<i>*enclavetoken</i>	Points to a data field of type wlmetok_t where the CreateWorkUnit() function is to return the WLM work unit enclave token.
<i>*classify</i>	Points to a server_classify_t structure that contains the classification information for the work request macro.
<i>*arrival_time</i>	Points to a null terminated character string that represents the arrival time in STICK format of the associated work request.
<i>*func_name</i>	Points to a null terminated character string that represents the descriptive function name of the associated work request.

### Returned Value

Upon successful completion CreateWorkUnit() returns a pointer to work unit enclave token of type wlmetok\_t. If unsuccessful, -1 is returned and errno is set to one of the following values:

EFAULT	An argument of this function contained an address that was not accessible to the caller.
EINVAL	An argument of this function contained an incorrect value.
EMVSWLMERROR	The WLM create failed. Use __errno2() to obtain the WLM service reason code for the failure.
EPERM	The calling thread's address space is not permitted to the BPX.WLMSEVER Facility class. The caller's address space must be permitted to the BPX.WLMSEVER Facility class if it is defined. If BPX.WLMSEVER is not defined, the calling process is not defined as a superuser (UID=0).
EMVSSAF2ERR	An error occurred in the security product.

**Related Information**

- “sys/\_\_wlm.h” on page 51
- “CheckSchEnv() — Check WLM Scheduling Environment” on page 172
- “ConnectServer() — Connect to WLM as a Server Manager” on page 219
- “ConnectWorkMgr() — Connect to WLM as a Work Manager” on page 221
- “ContinueWorkUnit() — Continue WLM Work Unit” on page 225
- “DeleteWorkUnit() — Delete a WLM Work Unit” on page 272
- “DisconnectServer() — Disconnect from WLM Server” on page 276
- “JoinWorkUnit() — Join a WLM Work Unit” on page 723
- “LeaveWorkUnit() — Leave a WLM Work Unit” on page 742
- “QueryMetrics() — Query WLM System Information” on page 1070
- “QuerySchEnv() — Query WLM Scheduling Environment” on page 1072

## crypt() — String Encoding Function

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <unistd.h>
```

```
char *crypt(const char *key, const char *salt);
```

### General Description

The `crypt()` function encodes the string pointed to by the *key* argument. It perturbs the Data Encryption Standard (DES) encryption algorithm with the first two characters in the string pointed to by the *salt* argument to perform this encoding. The first two *salt* characters must be chosen from the set:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9 . /
```

This function can be called from any thread.

### Returned Value

Upon successful completion, `crypt()` returns a pointer to a thread specific encoded string. The first two characters of the returned value are those of the *salt* argument.

### Notes:

1. The return value of `crypt()` points to a thread specific buffer which is overwritten each time `crypt()` is called from the same thread.
2. The values returned by `crypt()` are not portable to other X/Open-conformant systems.

If unsuccessful, `crypt()` returns a null pointer and sets `errno` to indicate the error.

### Special Behavior for OS/390 UNIX Services

The `crypt()` function will fail if:

**ENOMEM** Storage for `crypt()` output buffer is not available for thread from which `crypt()` has been invoked.

**EINVAL** First two characters of *salt* argument are not from the *salt* set.

### Related Information

- “`unistd.h`” on page 53
- “`encrypt()` — Encoding Function” on page 305

## cs() — Compare and Swap

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <stdlib.h>
```

```
int cs(cs_t *oldptr, cs_t *curptr, cs_t newword);
```

### General Description

The `cs()` built-in function compares the 4-byte value pointed to by *oldptr* to the 4-byte value pointed to by *curptr*. If they are equal, the 4-byte value, *newword*, is copied into the location pointed to by *curptr*. If they are unequal, the value pointed to by *curptr* is copied into the location pointed to by *oldptr*.

To avoid infringing on the user's name-space, this non-standard function is exposed only when you use the compiler option `LANGLVL(EXTENDED)`. When you use `LANGLVL(EXTENDED)` any relevant information in the header is also exposed.

The function uses the System/370 CS instruction. For a detailed description of this function, refer to the appendixes in the *ESA/390 Principles of Operations*.

### Returned Value

Returns 0 if the 4-byte value pointed to by *oldptr* is equal to the 4-byte value pointed to by *curptr*. If it is not equal, the function returns 1.

### Related Information

- *ESA/390 Principles of Operations*
- “`stdlib.h`” on page 45
- “`cds()` — Compare Double and Swap” on page 151

## csid() — Character Set ID for Multibyte Character

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <stdlib.h>
```

```
int csid(const char *c)
```

### External Entry Point

```
@@CSID, __csid;
```

### General Description

Determines the character set identifier for the specified multibyte character pointed to by *c*, that begins in the initial shift state.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

### Returned Value

Returns the character-set identifier for the multibyte character, or  $-1$  if the character is not valid.

**Note:** The multibyte character passed must begin in the initial shift state.

### Example

#### CBC3BC29

```
/* CBC3BC29
   This example checks character set ID for a character.
*/
#include "locale.h"
#include "stdio.h"
#include "stdlib.h"

main() {
    char *string = "A";
    int    rc;

    rc = csid(string);
    printf("character '%s' is in character set id %i\n", string, rc);
}
```

### Output

```
character 'A' is in character set id 0
```

**Related Information**

- “stdlib.h” on page 45

## csnap() — Request a Condensed Dump

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <ctest.h>
```

```
int csnap(char *dumptitle);
```

### General Description

Creates a display of the activation stack, including the Dynamic Storage Area (DSA), for each presently active function. Other environmental control blocks that may be required by IBM Service are also displayed. Under Language Environment, these consist of the Common Anchor Area (CAA) and the OS/390 C/C++ CAA information. The output is identified with *dumptitle*. See the CEE3DMP Language Environment callable service in the *OS/390 Language Environment Programming Guide*, SC28-1939 to determine where the output is written to.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

### Returned Value

Returns 0 if successful and nonzero otherwise.

### Example

```
#include <ctest.h>

int main(void) {

    int rc;
    rc = csnap("Sample csnap output");
}
```

### Related Information

- *IBM Language Environment Programming Guide*
- “ctest.h” on page 24
- “cdump() — Request a Main Storage Dump” on page 152
- “ctrace() — Request a Traceback” on page 252



## \_\_csplist — Retrieve CSP Parameters

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	C only	

### Format

```
#include <csp.h>
__csplist;
```

### General Description

\_\_csplist is a macro intended to be used to access the parameter list passed from Cross System Product (CSP) to your C/MVS program. The macro evaluates to the address of the first element of the parameter list. You can use array indexing to extract the subsequent parameters, casting each parameter to the expected type, as shown in the example below. If no parameters are passed, \_\_csplist[0] equals NULL.

You must include the `#pragma runopts(plist(ims))` directive if CSP is used to invoke a OS/390 C program.

argc will always be 1. See the *OS/390 C/C++ User's Guide* for information about the PLIST compiler option.

If you are expecting an integer and then a structure of type `s_type`, you should have the statements:

```
int_var = (int *) __csplist[0];
s_var   = (s_type *) __csplist[1];
```

### Related Information

- “csp.h” on page 24

## ctdli() — Call to DL/I

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

#### C only

```
#pragma runopts(env(IMS),plist(OS))
#include <ims.h>
```

or

```
#include <cics.h>
```

```
int ctdli(int parmcount, const char *function, ...);
```

### General Description

Invokes DL/I facilities. The first argument, *parmcount*, is optional for C, but is mandatory for C++ applications. The *parmcount* argument specifies the number of *function* arguments for the ctdli() call. The *function* argument specifies the DL/I function you want to perform. Because the format of the ctdli() call depends on the function selected, all of the variations are not given here. For complete details on the available functions, refer to the COBOL publications.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LONGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with LONGLVL(EXTENDED). When you use LONGLVL(EXTENDED) any relevant information in the header is also exposed.

To invoke ctdli() from an IMS transaction you need either the #pragma runopts(env(ims),plist(os)), or you need to specify the compiler options TARGET(IMS) and PLIST(OS).

### Returned Value

The Program Control Block (PCB) status field (2 bytes) is stored as an unsigned int and used as the returned value for ctdli(). If the PCB status field contains blanks (hex '4040'), ctdli() returns 0.

### Example

```
/* The following program demonstrates the use of the ctdli() function.
   It is a skeleton of a message processing program that calls ctdli()
   to retrieve messages and data.
```

```
   Do use the TARGET(IMS) and PLIST(IMS) compile options for C++
   applications.
```

```
*/
#ifdef __cplusplus
```

```

#pragma runopts(env(ims),plist(os))
#endif

#include <stdlib.h>
#include <ims.h>
#define n      20          /* I/O area size - Application dependent */
typedef struct {PCB_STRUCT(10)} PCB_10_TYPE;

int main(void)
{
    /* Function codes for ctdli */
    static const char func_GU[4]   = "GU  ";
    static const char func_ISRT[4] = "ISRT";

    char ssa_name[] = "ORDER    (ORDERKEY    = 666666)";

    int rc;

    char msg_seg_io_area[n];
    char db_seg_io_area[n];
    char alt_msg_seg_out[n];

    PCB_STRUCT_8_TYPE *alt_pcb;
    PCB_10_TYPE *db_pcb;
    IO_PCB_TYPE *io_pcb;

    io_pcb = (IO_PCB_TYPE *) (__pcblist)[0];
    alt_pcb = __pcblist[1];
    db_pcb = (PCB_10_TYPE *) __pcblist[2];
    :
    /* Get first message segment from message region */
    rc = ctdli(func_GU, io_pcb, msg_seg_io_area);
    :
    /* Get the data from the database having the specified key value */
    rc = ctdli(func_GU, db_pcb, db_seg_io_area, ssa_name);
    :
    /* Build output message in program's I/O area */
    rc = ctdli(func_ISRT, alt_pcb, alt_msg_seg_out);
    :
}

```

## Related Information

- “ims.h” on page 30

## ctermid() — Generate Path Name for Controlling Terminal

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
char *ctermid(char *string);
```

### General Description

*string* points to a memory location where the `ctermid()` function stores the name of the current controlling terminal. The memory location must be able to hold at least `L_ctermid` characters, where `L_ctermid` is a symbol defined in the `stdio.h` header file.

`ctermid()` returns a string that can be used as a path name to refer to the controlling terminal for the current process. If *string* is not null, `ctermid()` stores the path name in the specified location and returns the value of *string*. Otherwise, `ctermid()` uses a location of its own and returns a pointer to that location.

The path name returned can be used to access the controlling terminal, if the process has a controlling terminal.

### Returned Value

`ctermid()` is always successful; it returns a string that can be used as a path name to refer to the controlling terminal for the current process.

There are no documented `errno`s for this function.

### Example

#### CBC3BC32

```
/* CBC3BC32
   This example refers to the controlling terminal for the current process.
*/
#define _POSIX_SOURCE
#include <unistd.h>
#include <stdio.h>

main() {
    char termid[1025];

    if (ctermid(termid) == NULL)
        perror("ctermid() error");
    else
        printf("The control terminal is %s\n", termid);
}
```

### Output

The control terminal is /dev/tty

**Related Information**

- “stdio.h” on page 43
- “unistd.h” on page 53
- “ttyname() — Get the Name of a Terminal” on page 1632

## ctest() — Start Debug Tool

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <ctest.h>
```

```
int ctest(char *command);
```

### General Description

Invokes the Debug Tool from your application program. The parameter *command* is a character pointer to a list of valid Debug Tool commands, that ctest() uses to invoke Debug Tool.

If you choose not to compile your program with hooks, you can use well-placed ctest() function calls instead. (A *hook* is a conditional exit that transfers control to the debugger, when the code is run under the debugger.) You would create a hook when you compile with the TEST option, causing the exit to be in your generated code waiting to run. A hook has minimal effect on a program that is running without the debugger.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANTLRVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with LANTLRVL(EXTENDED). When you use LANTLRVL(EXTENDED) any relevant information in the header is also exposed.

For more information on the Debug Tool, refer to *Debug Tool User's Guide and Reference*, SC09-2137.

### Returned Value

Returns 0 if successful and a nonzero value otherwise.

### Examples

To let the debug tool gain control of your program, issue the command: ctest(NULL).

To display the call chain from within a program and then let the program continue execution, issue the function call: ctest("list calls; go;"). To set a breakpoint from within a ctest() call, try:

```
char *cmd = "at line 17 list my_struct; go;";
ctest(cmd);
```

**Related Information**

- “ctest.h” on page 24

ctime() — Convert Time to a Character String

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

Format

```
#include <time.h>

char *ctime(const time_t *timer);
```

General Description

Converts the calendar time pointed to by *timer* to local time in the form of a character string. A value for *timer* is usually obtained by a call to the `time()` function.

The `ctime()` function is equivalent to the function call: `asctime(localtime(timer))`

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

Returned Value

Returns no value if an error occurs.

Returns a pointer to a date and time string. The string returned by `ctime()` contains exactly 26 characters and has the format:

```
"%.3s %.3s%3d %.2d:%.2d:%.2d %d\n"
```

For example: `Mon Jul 16 02:03:55 1987\n\0`

Notes:

- This function is sensitive to time zone information which is provided by:
  - The `TZ` environmental variable when `POSIX(ON)` and `TZ` is correctly defined, or by the `_TZ` environmental variable when `POSIX(OFF)` and `_TZ` is correctly defined.
  - The `LC_TOD` category of the current locale if `POSIX(OFF)` or `TZ` is not defined.

The time zone external variables `tzname`, `timezone`, and `daylight` declarations remain feature test protected in `time.h`.

- The calendar time returned by a call to the `time()` function begins at epoch, which was at 00:00:00 Coordinated Universal Time, January 1, 1970.
- The `ctime()` function uses a 24-hour clock format.
- The days are abbreviated to: Sun, Mon, Tue, Wed, Thu, Fri, and Sat.
- The months are abbreviated to: Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, and Dec.
- All fields have a constant width.



- Dates with only one digit are preceded either with a 0 or a blank space.
- The new-line character (`\n`) and the null character (`\0`) occupy the last two positions of the string.
- The `asctime()`, `ctime()`, and other time functions may use a common, statically allocated buffer for holding the return string. Each call to one of these functions may destroy the result of the previous call.

### Special Behavior for POSIX C

- Under C/MVS POSIX applications only, this function is sensitive to time zone information, which is provided by:
  - The TZ environment variable if `time()` is called from a POSIX program and TZ is defined. The names of the time zones are parsed out of TZ and placed in the `tzname` array.
  - The LC\_TOD locale category, either `time()` is called from a non-POSIX program, or if TZ is not defined.

See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information about using POSIX support.

### Example CBC3BC33

```
/* CBC3BC33
   This example polls the system clock by using the library function
   time(). It then prints a message giving the current date and time.
*/
#include <time.h>
#include <stdio.h>

int main(void)
{
    time_t ltime;

    time(&ltime);
    printf("the time is %s", ctime(&ltime));
}
```

### Output

the time is Fri Jun 16 16:03:38 1995

### Related Information

- “locale.h” on page 33
- “time.h” on page 51
- “asctime() — Convert Time to Character String” on page 106
- “gmtime() — Convert Time to Broken-Down UTC Time” on page 640
- “localdtconv() — Date/Time Formatting Convention Inquiry” on page 754
- “localtime() — Convert Time and Correct for Local Time” on page 758
- “mktime() — Convert Local Time” on page 828
- “strftime() — Convert to Formatted Time” on page 1432
- “time() — Determine Current Time” on page 1570
- “tzset() — Set the Time Zone” on page 1637

## ctrace() — Request a Traceback

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <ctest.h>
```

```
int ctrace(char *dumptime);
```

### General Description

Requests a traceback. The output is identified with *dumptime*. `ctrace()` invokes the CEE3DMP Language Environment callable service with the following options: TRACEBACK, NOFILE, NOBLOCK, NOVARIABLE, NOSTORAGE, STACKFRAME(ALL), NOCOND, NOENTRY. See the CEE3DMP Language Environment callable service in the *OS/390 Language Environment Programming Guide*, SC28-1939 to determine where the output is written to.

If you compile the code using the GONUMBER option, this function will display, along with the traceback, the statement numbers and the offset information.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use `LANGLVL(EXTENDED)`.

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with `LANGLVL(EXTENDED)`. When you use `LANGLVL(EXTENDED)` any relevant information in the header is also exposed.

**Note:** The offsets displayed by `ctrace()` are from the beginning of the functions, whereas by default, compiler listings show offsets from the beginning of the source file. You can override the displayed offsets with the `OFFSET` compile-time option.

### Returned Value

Returns 0 if successful and a nonzero value otherwise.

### Example

#### CBC3BC34

```
/* CBC3BC34
   This example shows how ctrace() is used and the output produced.
*/
#include <ctest.h>
int main(void) {

    int rc;
    rc = ctrace("Sample ctrace output");
}
```

### Output for C++

CEE3DMP: Sample ctrace output  
06/16/95 6:13:31 PMPage: 1

Language Environment for MVS & VM V1 R5.0

Information for enclave ????????

Information for thread 8000000000000000

Traceback:

DSA Addr	Program Unit	PU Addr	PU Offset	Entry	E Addr	E Offset	Statement	Status
00065280		05337708	+0000011C	__ctrace	05337708	+0000011C		Call
000651E0		052005A8	+0000006C	main	052005A8	+0000006C		Call
000650C8		0533FA26	+000000B4	@@MNINV	0533FA26	+000000B4		Call
00065018	CEEBEXT	000079D8	+0000013C	CEEBEXT	000079D8	+0000013C		Call

## Output for C

CEE3DMP: Sample ctrace output  
06/16/95 6:12:47 PMPage: 1

Language Environment for MVS & VM V1 R5.0

Information for enclave ????????

Information for thread 8000000000000000

Traceback:

DSA Addr	Program Unit	PU Addr	PU Offset	Entry	E Addr	E Offset	Statement	Status
00065268		05337708	+0000011C	__ctrace	05337708	+0000011C		Call
000651E0		052006B8	+0000005E	main	052006B8	+0000005E		Call
000650C8		0533FA26	+000000B4	@@MNINV	0533FA26	+000000B4		Call
00065018	CEEBEXT	000079D8	+0000013C	CEEBEXT	000079D8	+0000013C		Call

## Related Information

- *IBM Language Environment Programming Guide*
- “ctest.h” on page 24
- “cdump() — Request a Main Storage Dump” on page 152
- “csnap() — Request a Condensed Dump” on page 242

## cuserid() — Return Character Login of the User

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE
#include <stdio.h>
```

```
char *cuserid(char *s);
```

### General Description

The `cuserid()` function generates a character representation of the name associated with the real or effective user ID of the process.

If `s` is a null pointer, this representation is generated in an area that may be overwritten by subsequent calls to `cuserid()`. A pointer to the area is returned. If `s` is not a null pointer, `s` is assumed to point to an array of at least `{L_cuserid}` bytes; the representation is deposited in this array. The symbolic constant `{L_cuserid}` is defined in `<stdio.h>` and has a value greater than 0.

### Returned Value

If `s` is not a null pointer, `s` is returned. If `s` is not a null pointer and the login name cannot be found, the null byte `'\0'` will be placed at `*s`. If `s` is a null pointer and the login name cannot be found, `cuserid()` returns a null pointer. If `s` is a null pointer and the login name can be found, the address of a buffer local to the calling thread containing the login name is returned.

### Related Information

- “`stdlib.h`” on page 45
- “`getlogin()` — Get the User Login Name” on page 550
- “`getpwnam()` — Access the User Database by User Name” on page 587
- “`getpwuid()` — Access the User Database by User ID” on page 589
- “`getuid()` — Get the Real User ID” on page 618
- “`geteuid()` — Get the Effective User ID” on page 519

## dbm\_clearerr() — Clear Database Error Indicator

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ndbm.h>
```

```
int dbm_clearerr(DBM *db);
```

### General Description

The `dbm_clearerr()` function clears the error condition of the database. The argument `db` is a handle to a database previously obtained by `dbm_open()`. Note that this does not correct any problems with the database due to previous failures. It simply allows `dbm_` operations to proceed. The database may be in an inconsistent or damaged state.

### Special Behavior for OS/390 UNIX Services

In a multi-threaded environment, the database error indicator is global to all threads using the database handle. Thus, clearing the error indicator affects all threads using the database handle.

### Returned Value

The return value is unspecified by X/Open.

If successful, `dbm_clearerr()` returns a 0. If unsuccessful, a `-1` is returned, and `errno` is set to:

**EINVAL** Invalid database descriptor specified.

### Related Information

- “`ndbm.h`” on page 37
- “`dbm_close()` — Close a Database” on page 256
- “`dbm_delete()` — Delete Database Record” on page 257
- “`dbm_error()` — Check Database Error Indicator” on page 258
- “`dbm_fetch()` — Get Database Content” on page 259
- “`dbm_firstkey()` — Get First Key in Database” on page 260
- “`dbm_nextkey()` — Get Next Key in Database” on page 262
- “`dbm_open()` — Open a Database” on page 264
- “`dbm_store()` — Store Database Record” on page 266

## dbm\_close() — Close a Database

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ndbm.h>
```

```
void dbm_close(DBM * db);
```

### General Description

The `dbm_close()` function closes a database. The `db` argument is the database handle returned by a previous call to `dbm_open()`.

### Special Behavior for OS/390 UNIX Services

A `dbm_close()` function call removes access to the specified database handle to all threads within the process.

### Returned Value

`dbm_close()` does not have a return value.

### Related Information

- “`ndbm.h`” on page 37
- “`dbm_clearerr()` — Clear Database Error Indicator” on page 255
- “`dbm_delete()` — Delete Database Record” on page 257
- “`dbm_error()` — Check Database Error Indicator” on page 258
- “`dbm_fetch()` — Get Database Content” on page 259
- “`dbm_firstkey()` — Get First Key in Database” on page 260
- “`dbm_nextkey()` — Get Next Key in Database” on page 262
- “`dbm_open()` — Open a Database” on page 264
- “`dbm_store()` — Store Database Record” on page 266

## dbm\_delete() — Delete Database Record

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ndbm.h>
```

```
int dbm_delete(DBM *db, datum key);
```

### General Description

The `dbm_delete()` function deletes a record and its key from the database. The `db` argument specifies the database handle returned by a previous call to `dbm_open()`. The `key` argument identifies the record the program is deleting. The `key` datum must contain a `dptr` pointer to the key, and the key length in `dsize`.

After calling `dbm_delete()`, during a pass through the keys by `dbm_firstkey()` and `dbm_nextkey()`, the application positioning must be reset by calling `dbm_firstkey()`. If not, unpredictable results may occur including retrieval of the same key multiple times, or not at all.

File space is not physically reclaimed by a `dbm_delete()` operation. That is, the file size is not reduced. However, the space is available for reuse, subject to hashing.

### Special Behavior for OS/390 UNIX Services

In a multi-threaded environment, changes made to the database by a `dbm_delete()` operation affect all threads using the database handle. Thus, all other threads must also reset their positioning by using the `dbm_firstkey()` function prior to using `dbm_nextkey()`. A previously executed `dbm_fetch()` operation by **another** thread for the same `key` still has correct buffer pointers to the previous data. The `dbm_delete()` operation does not affect this. All other operations on other threads, such as `dbm_fetch()` to this (now) deleted `key` will fail.

### Returned Value

If successful, `dbm_delete()` returns a 0. Otherwise, if unsuccessful, `dbm_delete()` returns -1, and sets the error value in `errno`. Also, the database error indicator may be set.

### Related Information

- “`ndbm.h`” on page 37
- “`dbm_clearerr()` — Clear Database Error Indicator” on page 255
- “`dbm_close()` — Close a Database” on page 256
- “`dbm_error()` — Check Database Error Indicator” on page 258
- “`dbm_fetch()` — Get Database Content” on page 259
- “`dbm_firstkey()` — Get First Key in Database” on page 260
- “`dbm_nextkey()` — Get Next Key in Database” on page 262
- “`dbm_open()` — Open a Database” on page 264
- “`dbm_store()` — Store Database Record” on page 266

## dbm\_error() — Check Database Error Indicator

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ndbm.h>
```

```
int dbm_error(DBM *db);
```

### General Description

The `dbm_error()` function returns the error condition of the database. The argument `db` is a handle to a database previously obtained by `dbm_open()`.

### Special Behavior for OS/390 UNIX Services

In a multi-threaded environment, the database error indicator is global to all threads using the database handle. Thus, the database error indicator may be set as a result of a database operation by another thread.

### Returned Value

`dbm_error()` returns a 0 if the error condition is not set and a nonzero value if the error condition is set.

### Related Information

- “`ndbm.h`” on page 37
- “`dbm_clearerr()` — Clear Database Error Indicator” on page 255
- “`dbm_close()` — Close a Database” on page 256
- “`dbm_delete()` — Delete Database Record” on page 257
- “`dbm_error()` — Check Database Error Indicator”
- “`dbm_fetch()` — Get Database Content” on page 259
- “`dbm_firstkey()` — Get First Key in Database” on page 260
- “`dbm_nextkey()` — Get Next Key in Database” on page 262
- “`dbm_open()` — Open a Database” on page 264
- “`dbm_store()` — Store Database Record” on page 266



## dbm\_fetch() — Get Database Content

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ndbm.h>
```

```
datum dbm_fetch(DBM * db, datum key);
```

### General Description

The `dbm_fetch()` function reads a record from the database. The argument `db` is a handle to a database previously obtained by `dbm_open()`. The argument `key` is a datum that has been initialized by the application program to the value of the key that matches the key of the record the program is fetching. A datum is a structure that consists of two members, `dptr` and `dsize`. The member `dptr` is a char pointer to an array of data that is `dsize` bytes in length. (Note: The data is arbitrary binary data and is not null terminated.)

The `dptr` is valid only until the next `dbm_` operation by this thread.

### Special Behavior for OS/390 UNIX Services

In a multi-threaded environment, the `dbm_fetch()` function returns a `dptr` in the datum structure to a data area that is thread specific. This data area is not affected by other threads operations on the database, with the exception of a `dbm_close()` operation, which invalidates the *datum*.

### Returned Value

If successful, `dbm_fetch()` returns the a datum containing a pointer to the data content `dptr`, and the data length `dsize`.

If unsuccessful, `dbm_fetch()` returns a null pointer in `dptr`, and returns the error value in `errno`. Also, the database error indicator may be set.

### Related Information

- “`ndbm.h`” on page 37
- “`dbm_clearerr()` — Clear Database Error Indicator” on page 255
- “`dbm_close()` — Close a Database” on page 256
- “`dbm_delete()` — Delete Database Record” on page 257
- “`dbm_error()` — Check Database Error Indicator” on page 258
- “`dbm_firstkey()` — Get First Key in Database” on page 260
- “`dbm_nextkey()` — Get Next Key in Database” on page 262
- “`dbm_open()` — Open a Database” on page 264
- “`dbm_store()` — Store Database Record” on page 266

## dbm\_firstkey() — Get First Key in Database

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ndbm.h>
```

```
datum dbm_firstkey(DBM * db);
```

### General Description

The `dbm_firstkey()` function returns the first key in the database. The argument *db* is a handle to a database previously obtained by `dbm_open()`. Since the keys are arbitrary binary data, the order of key return by `dbm_firstkey()` and `dbm_nextkey()` does not reflect any lexical ordering. In addition, the return order does not reflect record insertion ordering. All keys can be retrieved from the database by executing a loop such as:

```
for (key = dbm_firstkey(db); key.dptr != NULL; key = dbm_nextkey(db))
```

That is, establish positioning to the beginning by use of the `dbm_firstkey()` function, then loop doing `dbm_nextkey()` function calls until a null *dptr* is returned in the *datum*.

The returned *dptr* is valid only until the next `dbm_` operation by this thread.

### Special Behavior for OS/390 UNIX Services

In a multi-threaded environment, the `dbm_firstkey()` function returns a pointer to data that is thread specific. In addition, each thread maintains its own positioning information for `dbm_nextkey()` operations. However, other threads making modifications to the database, for example using `dbm_store()` or `dbm_delete()` can cause unpredictable results for threads executing `dbm_nextkey()`, including keys retrieved multiple times or not at all. The application must reset positioning to the beginning using `dbm_firstkey()` if another thread has done a modification to the database.

### Returned Value

If successful, `dbm_firstkey()` returns the a datum containing a pointer to the key *dptr*, and the key length *dsize*.

If unsuccessful, `dbm_firstkey()` returns a null pointer in *dptr*, and returns the error value in *errno*. Also, the database error indicator may be set.

### Related Information

- “`ndbm.h`” on page 37
- “`dbm_clearerr()` — Clear Database Error Indicator” on page 255
- “`dbm_close()` — Close a Database” on page 256
- “`dbm_delete()` — Delete Database Record” on page 257
- “`dbm_error()` — Check Database Error Indicator” on page 258
- “`dbm_fetch()` — Get Database Content” on page 259
- “`dbm_nextkey()` — Get Next Key in Database” on page 262

- “dbm\_open() — Open a Database” on page 264
- “dbm\_store() — Store Database Record” on page 266

## dbm\_nextkey() — Get Next Key in Database

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ndbm.h>
```

```
datum dbm_nextkey(DBM * db);
```

### General Description

The `dbm_nextkey()` function returns the next key in the database. The argument *db* is a handle to a database previously obtained by `dbm_open()`. Since the keys are arbitrary binary data, the order of key return by `dbm_firstkey()` and `dbm_nextkey()` does not reflect any lexical ordering. In addition, the return order does not reflect record insertion ordering. All keys can be retrieved from the database by executing a loop such as:

```
for (key = dbm_firstkey(db); key.dptr !=NULL; key = dbm_nextkey(db))
```

That is, establish positioning to the beginning by use of the `dbm_firstkey()` function, then loop doing `dbm_nextkey()` function calls until a null *dptr* is returned in *datum*.

The returned *dptr* is valid only until the next `dbm_` operation by this thread.

### Special Behavior for OS/390 UNIX Services

In a multi-threaded environment, the `dbm_nextkey()` function returns a pointer to data that is thread specific. In addition, each thread maintains its own positioning information for `dbm_nextkey()` operations. However, other threads making modifications to the database, for example using `dbm_store()` or `dbm_delete()` can cause unpredictable results for threads executing `dbm_nextkey()`, including keys retrieved multiple times or not at all. The application must reset positioning to the beginning using `dbm_firstkey()` if another thread has done a modification to the database.

### Returned Value

If successful, `dbm_nextkey()` returns the a datum containing a pointer to the key *dptr*, and the key length *dsize*.

If unsuccessful, `dbm_nextkey()` returns a null pointer in *dptr*, and returns the error value in *errno*. Also, the database error indicator may be set.

### Related Information

- “`ndbm.h`” on page 37
- “`dbm_clearerr()` — Clear Database Error Indicator” on page 255
- “`dbm_close()` — Close a Database” on page 256
- “`dbm_delete()` — Delete Database Record” on page 257
- “`dbm_error()` — Check Database Error Indicator” on page 258
- “`dbm_fetch()` — Get Database Content” on page 259
- “`dbm_firstkey()` — Get First Key in Database” on page 260

- “dbm\_open() — Open a Database” on page 264
- “dbm\_store() — Store Database Record” on page 266

## dbm\_open() — Open a Database

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ndbm.h>
```

```
DBM *dbm_open(const char *file, int open_flags, mode_t file_mode);
```

### General Description

The `dbm_open()` function opens a database. The *file* argument is the pathname of the database, not including the filename suffix (the part after the `.`). The database is stored in two files. One file is a directory containing a bit map of blocks in use and has `.dir` as its suffix. The second file contains all the data and has `.pag` as its suffix. The *open\_flags* argument has the same meaning as the *flags* argument of `open()` except that a database opened for write-only access opens the files for read and write access. The *file\_mode* argument has the same meaning as the third argument of `open()`.

The number of records that can be stored in the database is limited by the file space available for the `.dir` and `.pag` files, and by the underlying key hashing. If multiple keys hash to the same 32 bit hash value, the number of keys for that hash value is limited to the amount of data (key sizes plus content sizes plus overhead) that can be stored in a single logical block of 1024 bytes.

### Special Behavior for OS/390 UNIX Services

In a multi-threaded environment, the `dbm_` functions have both POSIX process wide and thread specific characteristics. OS/390 UNIX services provide the following multi-threaded behavior:

1. A database handle returned by the `dbm_open()` function is a process wide resource. This means that multiple threads within the process can access the database using the same database handle.
2. Each thread using a given database handle has its own positioning information for `dbm_firstkey()` and `dbm_nextkey()` operations. This means that multiple threads can each be executing a `dbm_nextkey()` loop.
3. Each thread using a given database handle has its own buffering for `dbm_fetch()` operations. This means that a pointer to a keys content (as returned by `dbm_fetch()`) remains valid, even if other threads modify the database.
4. Database modifications are automatically reflected to all of the threads using the same database handle. For example, if a thread adds a key/data pair using `dbm_store()`, a `dbm_fetch()` of that key by another thread will be successful.
5. Operations which modify the database, such as `dbm_store()` and `dbm_delete()`, can cause unpredictable results to threads executing `dbm_nextkey()`. If a database modification is done, all threads should reset positioning via a `dbm_firstkey()` call prior to executing `dbm_nextkey()`.

6. A `dbm_close()` operation removes access to the database for all threads that use the database handle.
7. Multiple `dbm_open()` operations, whether by a single thread, multiple threads within a process, or by multiple processes are permitted, but for read access only. No protection is provided for database modification, and modification can result in unpredictable results, including database destruction.

### Returned Value

If successful, `dbm_open()` returns the a pointer to the database descriptor.

If unsuccessful, `dbm_open()` returns a null pointer, and returns the error value in `errno`.

### Related Information

- “`ndbm.h`” on page 37
- “`dbm_clearerr()` — Clear Database Error Indicator” on page 255
- “`dbm_close()` — Close a Database” on page 256
- “`dbm_delete()` — Delete Database Record” on page 257
- “`dbm_error()` — Check Database Error Indicator” on page 258
- “`dbm_fetch()` — Get Database Content” on page 259
- “`dbm_firstkey()` — Get First Key in Database” on page 260
- “`dbm_nextkey()` — Get Next Key in Database” on page 262
- “`dbm_store()` — Store Database Record” on page 266
- “`open()` — Open a File” on page 872

## dbm\_store() — Store Database Record

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ndbm.h>
```

```
int dbm_store(DBM * db, datum key, datum content, int store_mode);
```

### General Description

The `dbm_store()` function writes a record to a database. The `db` argument specifies the database handle returned by a previous call to `dbm_open()`. The `key` argument identifies the record the program is deleting. The `key` datum must contain a `dptr` pointer to the key, and the key length in `dsize`. The argument `content` is a datum. that describes the data record being stored. record the program is writing. The `content` datum. contains a `dptr` pointer to the data, and the data length in `dsize`.

The argument `store_mode` controls whether `dbm_store()` replaces a already existing record that has the same key. The `store_mode` argument may be any one of the following set of symbols defined in the `<ndbm.h>` include file:

DBM_INSERT	Do not add the <code>key</code> and <code>content</code> pair if the <code>key</code> already exists in the database. If the <code>key</code> doesn't already exist, add the new <code>key</code> and <code>content</code> pair.
DBM_REPLACE	Replace the <code>key</code> and <code>content</code> pair in the database with the new pair if the <code>key</code> already exists. If the <code>key</code> doesn't already exist, add the new <code>key</code> and <code>content</code> pair.

After calling `dbm_store()`, during a pass through the keys by `dbm_firstkey()` and `dbm_nextkey()`, the application positioning must be reset by calling `dbm_firstkey()`. If not, unpredictable results may occur including retrieval of the same key multiple times, or not at all.

The number of records that can be stored in the database is limited by the file space available for the `.dir` and `.pag` files, and by the underlying key hashing. If multiple keys hash to the same 32 bit hash value, the number of keys for that hash value is limited to the amount of data (key sizes plus content sizes plus overhead) that can be stored in a single logical block of 1024 bytes.

### Special Behavior for OS/390 UNIX Services

In a multi-threaded environment, changes made to the database by a `dbm_store()` operation affect all threads using the database handle. Thus, all other threads must also reset their positioning by using the `dbm_firstkey()` function prior to using `dbm_nextkey()`. A previously executed `dbm_fetch()` operation by **another** thread for the same `key` still has correct buffer pointers to the previous data. The `dbm_store()` operation does not affect this. All other operations, such as `dbm_fetch()` or `dbm_delete()`, will automatically have access to the new `key` and `content` pair.



## Returned Value

If successful, `dbm_store()` returns a 0. If `DBM_INSERT` is specified, and the *key* already exists, a 1 is returned. Otherwise, if unsuccessful, `dbm_store()` returns -1, and sets the error value in `errno`. Also, the database error indicator may be set.

**EFBIG** Seek/Write operation failed attempting to write new block. This `errno` is not part of the `errno` set described by X/Open for this function. You may be able to store other *key* and *content* pairs when the *key* hashes to a different value.

**ENOSPC** *key* plus *content* plus block overhead does not fit into a block. This `errno` is not part of the `errno` set described by X/Open for this function. The *key* plus *content* underlying data lengths need be less or equal to 1012 bytes in length. block

## Related Information

- “`ndbm.h`” on page 37
- “`dbm_clearerr()` — Clear Database Error Indicator” on page 255
- “`dbm_close()` — Close a Database” on page 256
- “`dbm_delete()` — Delete Database Record” on page 257
- “`dbm_error()` — Check Database Error Indicator” on page 258
- “`dbm_fetch()` — Get Database Content” on page 259
- “`dbm_firstkey()` — Get First Key in Database” on page 260
- “`dbm_nextkey()` — Get Next Key in Database” on page 262
- “`dbm_open()` — Open a Database” on page 264

## decabs() — Decimal Absolute Value

### Standards

Standards / Extensions	C or C++	Dependencies
C/MVS	C only	

### Format

```
#include <decimal.h>
```

```
decimal(n,p) decabs(decimal(n,p) pdec);
```

### General Description

The built-in function `decabs()` accepts a decimal type expression as an argument and returns the absolute value of the decimal argument, in the same decimal type as the argument. The function does not change the content of the argument.

The parameter *n* can be any integral value between 1 and `DEC_DIG`. The parameter *p* can be any integral value between 0 and `DEC_PRECISION`, although it must be less than or equal to *n*. `DEC_DIG` and `DEC_PRECISION` are defined inside `decimal.h`.

If the content of the given argument is not in native packed decimal format, behavior is undefined.

### Example

#### CBC3BD01

```
/* CBC3BD01 */
#include <decimal.h>

decimal(10,2) p1, p2;
int main(void) {
    p2 = -1234.56d;
    p1 = decabs(p2);
    printf("p1 = %D(10,2), p2 = %D(10,2)\n", p1, p2);
    return(0);
}
```

### Output

```
p1 = 1234.56, p2 = -1234.56
```

### Related Information

- “`decimal.h`” on page 24
- “`decchk()` — Check for Valid Decimal Types” on page 269
- “`decfix()` — Fix Up a Nonpreferred Sign Variable” on page 271

## decchk() — Check for Valid Decimal Types

### Standards

Standards / Extensions	C or C++	Dependencies
C/MVS	C only	

### Format

```
#include <decimal.h>
```

```
int decchk(decimal(n,p) pdec);
```

### General Description

The built-in function `decchk()` accepts a decimal type expression as an argument and returns a status value of type `int`.

The status can be interpreted as follows:

**DEC\_VALUE\_OK** A valid decimal representation value (including the less-preferred but valid sign, A-F).

**DEC\_BAD\_NIBBLE**

The leftmost half-byte is not 0 in a decimal type number that has an even number of digits. For example, 123 is stored in `decimal(2,0)`. If such a number is packed, then it is used.

**DEC\_BAD\_DIGIT** Digits not allowed (not 0-9). If such a number is packed, then it is used.

**DEC\_BAD\_SIGN** Sign not allowed (not A-F). If such a number is packed, then it is used.

The function return status can be masked to return multiple status.

The parameter *n* can be any integral value between 1 and `DEC_DIG`. The parameter *p* can be any integral value between 0 and `DEC_PRECISION`, although it must be less than or equal to *n*. `DEC_DIG` and `DEC_PRECISION` are defined inside `decimal.h`.

If the content of the given argument is not in native packed decimal format, the behavior is undefined.

### Example

```
#include <decimal.h>
```

```
decimal(10,2) p1;
char mem2[3] = { 0x12, 0x34, 0x5c }; /* bad half-byte */
char mem3[3] = { 0x02, 0xa4, 0x5c }; /* bad digit */
char mem4[3] = { 0x02, 0x34, 0x56 }; /* bad sign */
char mem5[3] = { 0x12, 0xa4, 0x56 }; /* bad half-byte, digit and sign */
decimal(4,0) *pp2;
decimal(4,0) *pp3;
decimal(4,0) *pp4;
decimal(4,0) *pp5;
int main(void) {
    p1 = 123456.78d;
    pp2 = (decimal(4,0) *) mem2;
```

```

pp3 = (decimal(4,0) *) mem3;
pp4 = (decimal(4,0) *) mem4;
pp5 = (decimal(4,0) *) mem5;

if (decchk(p1) == DEC_VALUE_OK) {
    printf("p1 is a valid decimal representation value.\n");
}
if (decchk(*pp2) == DEC_BAD_NIBBLE) {
    printf("pp2 points to a bad half-byte value!\n");
}
if (decchk(*pp3) == DEC_BAD_DIGIT) {
    printf("pp3 points to an illegal digit!\n");
}
if (decchk(*pp4) == DEC_BAD_SIGN) {
    printf("pp4 points to an illegal sign!\n");
}
/* The wrong way ----- */
if (decchk(*pp5) == DEC_BAD_SIGN) {
    printf("YOU SHOULD NOT GET THIS!!!!\n");
}
/* The right way ----- */
if ((decchk(*pp5) & DEC_BAD_SIGN) == DEC_BAD_SIGN) {
    printf("pp5 points to an illegal sign!\n");
}
return(0);
}

```

### Output

```

p1 is a valid decimal representation value.
pp2 points to a bad half-byte value!
pp3 points to an illegal digit!
pp4 points to an illegal sign!
pp5 points to an illegal sign!

```

### Related Information

- “decimal.h” on page 24
- “decabs() — Decimal Absolute Value” on page 268
- “decfix() — Fix Up a Nonpreferred Sign Variable” on page 271

## decfix() — Fix Up a Nonpreferred Sign Variable

### Standards

Standards / Extensions	C or C++	Dependencies
C/MVS	C only	

### Format

```
#include <decimal.h>
```

```
decimal(n,p) decfix(decimal(n,p) pdec);
```

### General Description

The built-in function `decfix()` accepts a decimal type expression as an argument and returns a decimal value that has the same type and same value as the argument with the correct preferred sign. The function does not change the content of the argument.

The parameter *n* can be any integral value between 1 and `DEC_DIG`. The parameter *p* can be any integral value between 0 and `DEC_PRECISION`, though it must be less than or equal to *n*. `DEC_DIG` and `DEC_PRECISION` are defined inside `decimal.h`.

If the content of the given argument is not in native packed decimal format, behavior is undefined.

### Example

```
#include <decimal.h>

char *ptr;
char mem[3] = { 0x01, 0x23, 0x4A };
decimal(4,0) *pp;
decimal(4,0) p;
int main(void) {
    pp = (decimal(4,0) *) mem;
    p = decfix(*pp);
    ptr = (char *) +;
    printf("Before decfix : %X%X%X\n", mem[0], mem[1], mem[2]);
    printf("After decfix : %X%X%X\n", ptr[0], ptr[1], ptr[2]);
    return(0);
}
```

### Output

```
Before decfix : 1234A
After decfix : 1234C
```

### Related Information

- “`decimal.h`” on page 24
- “`decabs()` — Decimal Absolute Value” on page 268
- “`decchk()` — Check for Valid Decimal Types” on page 269

## DeleteWorkUnit() — Delete a WLM Work Unit

### Standards

Standards / Extensions	C or C++	Dependencies
	both	

### Format

```
#include <sys/__wlm.h>
int DeleteWorkUnit(wlmetok_t *enclavetoken);
```

### General Description

The DeleteWorkUnit() function provides the ability for an application to delete a WLM work unit.

*\*enclavetoken* Points to a work unit enclave token that was returned from a call to CreateWorkUnit() or ContinueWorkUnit().

### Returned Value

Upon successful completion DeleteWorkUnit() returns a zero. If the function is unsuccessful, -1 is returned and errno is set on to one of the following values:

EFAULT	An argument of this function contained an address that was not accessible to the caller.
EINVAL	An argument of this function contained an incorrect value.
EMVSWLMERROR	The WLM delete enclave failed. Use __errno2() to obtain the WLM service reason code for the failure.
EPERM	The calling thread's address space is not permitted to the BPX.WLMSEVER Facility class. The caller's address space must be permitted to the BPX.WLMSEVER Facility class if it is defined. If BPX.WLMSEVER is not defined, the calling process is not defined as a superuser (UID=0).
EMVSSAF2ERR	An error occurred in the security product.

### Related Information

- “sys/\_\_wlm.h” on page 51
- “CheckSchEnv() — Check WLM Scheduling Environment” on page 172
- “ConnectServer() — Connect to WLM as a Server Manager” on page 219
- “ConnectWorkMgr() — Connect to WLM as a Work Manager” on page 221
- “ContinueWorkUnit() — Continue WLM Work Unit” on page 225
- “DisconnectServer() — Disconnect from WLM Server” on page 276
- “JoinWorkUnit() — Join a WLM Work Unit” on page 723
- “LeaveWorkUnit() — Leave a WLM Work Unit” on page 742
- “QueryMetrics() — Query WLM System Information” on page 1070
- “QuerySchEnv() — Query WLM Scheduling Environment” on page 1072

## difftime() — Compute Time Difference

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <time.h>
```

```
double difftime(time_t time2, time_t time1);
```

### General Description

Computes the difference in seconds between *time2* and *time1*, which are calendar times returned by `time()`.

The `difftime()` function returns the difference between two calendar times as a double. The return value is hexadecimal floating-point or IEEE floating-point format depending on the floating-point mode of the thread invoking `difftime()`. The `difftime()` function uses `__isBFP()` to determine which floating-point format (hexadecimal floating-point or IEEE floating-point) to return on the invoking thread.

### Returned Value

Returns the elapsed time in seconds from *time1* to *time2* as a double.

### Example CBC3BD04

```
/* CBC3BD04
This example shows a timing application using difftime(). The example
calculates how long, on average, it takes a user to input some data
to the program.
*/
#include <time.h>
#include <stdio.h>

int main(void)
{
    time_t start, finish;
    int i, n, num;
    int answer;

    printf("11 x 55 = ? Enter your answer below\n");
    time(&start);
    scanf("%d",&answer);
    time(&finish);
    printf("You answered %s in %.0f seconds.\n",
        answer == 605 ? "correctly" : "incorrectly",
        difftime(finish,start));
}
```

### Output

```
11 x 55 = ? Enter your answer below
605
You answered correctly in 20 seconds
```

## **Related Information**

- “time.h” on page 51
- “asctime() — Convert Time to Character String” on page 106
- “ctime() — Convert Time to a Character String” on page 250
- “gmtime() — Convert Time to Broken-Down UTC Time” on page 640
- “\_\_isBFP() — Determine Application Floating-Point Format” on page 705
- “localtime() — Convert Time and Correct for Local Time” on page 758
- “mktime() — Convert Local Time” on page 828
- “strftime() — Convert to Formatted Time” on page 1432
- “time() — Determine Current Time” on page 1570



## dirname() — Report the Parent Directory of a Pathname

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <libgen.h>
```

```
char *dirname(char *path);
```

### General Description

The `dirname()` function takes a pointer to a character string that contains a pathname, and returns a pointer to a string that is a pathname of the parent directory of that file. Trailing '/' characters in the path are not counted as part of the path.

If *path* does not contain a '/' then `dirname()` returns a pointer to the string ".". If *path* is a null pointer or points to an empty string, `dirname()` returns a pointer to the string ".".

The `dirname()` function may modify the string pointed to by *path*.

Examples:

Input String	Output String
"/usr/lib"	"/usr"
"/usr/"	"/"
"usr"	."
"/"	"/"
."	."
.."	."

### Returned Value

The `dirname()` function returns a pointer to a string that is the parent directory of *path*. If *path* is a null pointer or points to an empty string, a pointer to a string "." is returned.

There are no `errno` values defined for `dirname()`.

### Related Information

- "libgen.h" on page 31
- "basename() — Return the Last Component of a Pathname" on page 125

## DisconnectServer() — Disconnect from WLM Server

### Standards

Standards / Extensions	C or C++	Dependencies
	both	

### Format

```
#include <sys/_wlm.h>
int DisconnectServer(unsigned long *conn_tkn);
```

### General Description

The DisconnectServer function provides the ability for an application to disconnect from WLM.

*\*conn\_tkn* Specifies the connect token that represents the WLM connection that is to be disconnected.

### Returned Value

Upon successful completion DisconnectServer() returns a zero. If the function is unsuccessful, -1 is returned and errno is set on to one of the following values:

EFAULT	An argument of this function contained an address that was not accessible to the caller.
EINVAL	An argument of this function contained an incorrect value.
EMVSWLMERROR	The WLM disconnect failed. Use __errno2() to obtain the WLM service reason code for the failure.
EPERM	The calling thread's address space is not permitted to the BPX.WLMSEVER Facility class. The caller's address space must be permitted to the BPX.WLMSEVER Facility class if it is defined. If BPX.WLMSEVER is not defined, the calling process is not defined as a superuser (UID=0).
EMVSSAF2ERR	An error occurred in the security product.

### Related Information

- “sys/\_wlm.h” on page 51
- “CheckSchEnv() — Check WLM Scheduling Environment” on page 172
- “ConnectServer() — Connect to WLM as a Server Manager” on page 219
- “ConnectWorkMgr() — Connect to WLM as a Work Manager” on page 221
- “ContinueWorkUnit() — Continue WLM Work Unit” on page 225
- “CreateWorkUnit() — Create WLM Work Unit” on page 236
- “DeleteWorkUnit() — Delete a WLM Work Unit” on page 272
- “JoinWorkUnit() — Join a WLM Work Unit” on page 723
- “LeaveWorkUnit() — Leave a WLM Work Unit” on page 742
- “QueryMetrics() — Query WLM System Information” on page 1070
- “QuerySchEnv() — Query WLM Scheduling Environment” on page 1072

## div() — Calculate Quotient and Remainder

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
div_t div(int numerator, int denominator);
```

### General Description

Calculates the quotient and remainder of the division of *numerator* by *denominator*.

### Returned Value

Returns a structure of type `div_t`, containing both the quotient `int quot` and the remainder `int rem`. This structure is defined in `stdlib.h`. If the returned value cannot be represented, the behavior of `div()` is undefined. If *denominator* is 0, the same exception will be raised as if you divided by 0. That is, you get the error CEE3209S (Fixed point divide exception).

### Related Information

- “`stdlib.h`” on page 45
- “`ldiv()` — Compute Quotient and Remainder of Integral Division” on page 740

## dllfree() — Free the Supplied DLL

### Standards

Standards / Extensions	C or C++	Dependencies
C/MVS	both	

### Format

```
#include <dll.h>
```

```
int dllfree(dllhandle* dllHandle);
```

### General Description

Frees the supplied DLL. It also deletes the DLL from memory if the handle was the last handle accessing the DLL.

### Notes:

- This function is not available under SPC, MTF and CSP environments.
- If a DLL is loaded implicitly, it cannot be deleted with `dllfree()`. For more information on the implicit use of DLLs, see the *OS/390 C/C++ Programming Guide*.
- DLLs that are loaded explicitly, that is with `dllload()`, and are not freed with a corresponding call to `dllfree()`, are freed automatically at enclave termination in LIFO sequence.
- C++ destructors are executed only once, when the DLL load module is physically deleted.

### Returned Value

The function returns one of the following values and set `errno` if the return code is not 0:

Value	Meaning
0	Successful
1	The <code>dllHandle</code> supplied is NULL or <code>dllhandle</code> is inactive.
2	There are no DLLs to be deleted.
3	DLL is not physically deleted because there is another <code>dllHandle</code> for this DLL or there is an implicit reference to the DLL.
4	Delete of DLL failed.
5	No match is found for input <code>dllHandle</code> .
6	Not supported under this environment.
7	C++ destructors are currently running for this DLL. A <code>dllfree()</code> is already in progress.

## Example

### CBC3BDL4

```

/* CBC3BDL4
   The following example shows how to use dllfree() to free the
   dllhandle for the DLL stream.
*/
#include <stdio.h>
#include <dll.h>
#include <stdlib.h>

int main() {
    dllhandle *handle;
    char *name="stream";
    int (*fptr1)(int);
    int *ptr1_var1;
    int rc=0;

    handle = dllload(name);          /* call to stream DLL */
    if (handle == NULL) {
        perror("failed on call to stream DLL");
        exit(-1);
    }

    fptr1 = (int (*)(int)) dllqueryfn(handle,"f1");
                                   /* retrieving f1 function */
    if (fptr == NULL) {
        perror("failed on retrieving f1 function");
        exit(-2);
    }

    ptr_var1 = dllqueryvar(handle,"var1");
                                   /* retrieving var1 variable */
    if (ptr_var1 == NULL) {
        perror("failed on retrieving var1 variable");
        exit(-3);
    }

    rc = fptr(*ptr_var1);           /* execute DLL function f1 */
    *ptr_var++;                     /* increment value of var1 */

    rc = dllfree(handle);           /* freeing handle to stream DLL */
    if (rc != 0) {
        perror("failed on dllfree call");
    }

    return (0);
}

```

## Related Information

- “dllload() — Load the DLL and Connect it to the Application” on page 280
- “dllqueryfn() — Obtain a Pointer to a DLL Function” on page 282
- “dllqueryvar() — Obtain a Pointer to a DLL Variable” on page 284

## dllload() — Load the DLL and Connect it to the Application

### Standards

Standards / Extensions	C or C++	Dependencies
C/MVS	both	

### Format

```
#include <dll.h>
```

```
dllhandle* dllload(const char * dllName);
```

### General Description

Loads the Dynamic Link Library (DLL) into memory (if it has not been previously loaded) and connects it to the application. The function that called the DLL receives a handle that uniquely identifies the requested DLL for subsequent explicit requests for that DLL.

A different handle is returned for each successful call to `dllload()`. A DLL is physically loaded only once, even though there may be many calls to `dllload()`. C++ constructors are run only once.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

The `dllName` identifies the DLL load module to be loaded. It must be a character string terminated with the NULL character. The DLL module must be a member of a PDS or an alias to it. When a DLL is loaded, MVS uses the following search sequence to load the module:

- Steplib data set
- Joblib data set
- Link pack area or extended link pack area (LPA/ELPA).

HFS search uses the path defined by the `LIBPATH` environment variable or the current directory by default. `LIBPATH` allows an absolute or relative pathname to be searched when loading a DLL. If the input filename contains a slash, it is used as is to locate the DLL. If the input filename does not contain a slash, then `LIBPATH` is used to determine the pathname to load. `LIBPATH` specifies a list of directories separated by colons. If the `LIBPATH` begins or ends with a colon, then the working directory is also searched first or last, depending on the position of the stand-alone colon. The “`::`” specification can only occur at the beginning or end of the list of directories. If you are running `POSIX(ON)` then the HFS is searched first followed by MVS. If you are running `POSIX(OFF)`, then MVS is searched first followed by HFS. This double search can be avoided by using unambiguous DLL names. For example, “`//mvsname`” for MVS names, and “`./hfsname`” for HFS names. See *OS/390 C/C++ Programming Guide* for more information.

Under the CICS environment, the search sequence for DLL load modules is the same as that used for dynamically loaded CICS modules.

**Note:** The `AMODE` of the application must be the same as the `AMODE` of the DLL load module.

This function is not available under SPC, MTF and CSP environments.

Returned Value

dlload() returns a unique handle that identifies the DLL if the call is successful. If unsuccessful, NULL is returned and errno is set.

ENOEXEC           The new process image file has the appropriate access permission but is not in the proper format.

**Note:**

Reason codes further qualify the errno. For most of the reason codes, see *OS/390 UNIX System Services Messages and Codes*.

For ENOEXEC, the reason codes are:

Reason Code	Explanation
X'xxxx0C27'	The target HFS file is not in the correct format to be an executable file.
X'xxxx0C31'	The target HFS file is built at a level that is higher than that supported by the running system.

Example  
CBC3BDL1

```
/* CBC3BDL1
   The following example shows how to invoke dlload() functions
   from a simple C application.
*/
#include <stdio.h>
#include <dll.h>

main() {
    dllhandle *handle;
    char *name="stream";
    int (*fptr1)();
    int *ptr1_var1;
    int rc=0;

    handle = dlload(name);
    if (handle == NULL) {
        perror("failed on dlload of stream DLL");
        exit (-1);
    }
}
```

Related Information

- “dlqueryfn() — Obtain a Pointer to a DLL Function” on page 282
- “dlqueryvar() — Obtain a Pointer to a DLL Variable” on page 284
- “dlfree() — Free the Supplied DLL” on page 278

## dllqueryfn() — Obtain a Pointer to a DLL Function

### Standards

Standards / Extensions	C or C++	Dependencies
C/MVS	both	

### Format

```
#include <dll.h>
```

```
void (* dllqueryfn(dllhandle *dllHandle, const char *funcName)) ();
```

### General Description

Obtains a pointer to a DLL function (funcName). It uses the dllHandle returned from a previous successful call to dllload() for input. funcName represents the name of an exported function from the DLL. It must be a character string terminated with the NULL character.

This function is not available under the SPC, MTF, and CSP environments.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

dllqueryfn() returns a pointer to a function, funcName, that can be used to invoke the desired function in a DLL. If unsuccessful, it returns NULL and sets errno.

### Example

#### CBC3BDL2

```
/* CBC3BDL2
   The following example shows how to use dllqueryfn() to obtain
   a pointer to a function, f1 that is in DLL load module stream.
*/
#include <stdio.h>
#include <dll.h>

main() {
    dllhandle *handle;
    char *name="stream";
    int (*fptr1)();
    int *ptr1_var1;
    int rc=0;

    handle = dllload(name);
    if (handle == NULL) {
        perror("failed on dllload of stream DLL");
        exit (-1);
    }

    fptr1 = (int (*)( )) dllqueryfn(handle,"f1");
    if (fptr1 == NULL) {
        perror("failed on retrieving f1 function");
        exit (-2);
    }
}
```



**Related Information**

- “dllload() — Load the DLL and Connect it to the Application” on page 280
- “dllqueryvar() — Obtain a Pointer to a DLL Variable” on page 284
- “dllfree() — Free the Supplied DLL” on page 278

## dllqueryvar() — Obtain a Pointer to a DLL Variable

### Standards

Standards / Extensions	C or C++	Dependencies
C/MVS	both	

### Format

```
#include <dll.h>
```

```
void* dllqueryvar(dllhandle *dllHandle, const char *varName);
```

### General Description

Obtains a pointer to a DLL variable (*varName*). It uses the *dllHandle* returned from a previous successful call to *dllload()* for input. *varName* represents the name of an exported variable from the DLL. It must be a character string terminated with the NULL character.

This function is not available under SPC, MTF and CSP environments.

### Returned Value

*dllqueryvar()* returns a pointer to a variable in the storage of the DLL if successful. If unsuccessful, it returns NULL and sets *errno*.

### Example

#### CBC3BDL3

```
/* CBC3BDL3
   The following example shows how to use dllqueryvar() to obtain a
   pointer to a variable, var1, that is in DLL load module stream.
*/
#include <stdio.h>
#include <dll.h>

main() {
    dllhandle *handle;
    char *name="stream";
    int (*fptr1)();
    int *ptr1_var1;
    int rc=0;

    handle = dllload(name);
    if (handle == NULL) {
        perror("failed on dllload of stream DLL");
        exit (-1);
    }

    fptr1 = (int (*)( )) dllqueryfn(handle,"f1");
    if (fptr1 == NULL) {
        perror("failed on retrieving f1 function");
        exit (-2);
    }

    ptr_var1 = dllqueryvar(handle,"var1");
    if (ptr_var1 == NULL) {
        perror("failed on retrieving var1 variable");
        exit (-3);
    }
}
```

**Related Information**

- “dllload() — Load the DLL and Connect it to the Application” on page 280
- “dllqueryfn() — Obtain a Pointer to a DLL Function” on page 282
- “dllfree() — Free the Supplied DLL” on page 278

## drand48() — Pseudo-random Number Generator

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>
```

```
double drand48(void);
```

### General Description

The `drand48()`, `erand48()`, `jrand48()`, `lrand48()`, `mrnd48()` and `nrnd48()` functions generate uniformly distributed pseudorandom numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The functions `drand48()` and `erand48()` return non-negative, double-precision, floating-point values, uniformly distributed over the interval  $[0.0, 1.0)$ . These functions have been extended to determine floating-point format (hexadecimal floating-point or IEEE floating-point) of the returned value using the `__isBFP()` function.

The functions `drand48()` and `erand48()` return non-negative, double-precision, floating-point values, uniformly distributed over the interval  $[0.0, 1.0)$ .

The functions `lrand48()` and `nrnd48()` return non-negative, long integers, uniformly distributed over the interval  $[0, 2^{**31})$ .

The functions `mrnd48()` and `jrand48()` return signed long integers, uniformly distributed over the interval  $[-2^{**31}, 2^{**31})$ .

The `drand48()` function generates the next 48-bit integer value in a sequence of 48-bit integer values,  $X(i)$ , according to the linear congruential formula:

$$X(n+1) = (aX(n) + c) \bmod (2^{**48}) \quad n \geq 0$$

The initial values of  $X$ ,  $a$ , and  $c$  are:

```
X(0) = 1
a    = 5deece66d (base 16)
c    = b          (base 16)
```

C/370 provides storage to save the most recent 48-bit integer value of the sequence,  $X(i)$ . This storage is shared by the `drand48()`, `lrand48()` and `mrnd48()` functions. The value,  $X(n)$ , in this storage may be reinitialized by calling the `lcong48()`, `seed48()` or `srand48()` function. Likewise, the values of  $a$  and  $c$ , may be changed by calling the `lcong48()` function. Thereafter, whenever the `seed48()` or `srand48()` function is called to change  $X(n)$ , the initial values of  $a$  and  $c$  are also reestablished.

### Special Behavior for OS/390 UNIX Services

You can make the `drand48()` function and other functions in the `drand48` family thread specific by setting the environment variable `_RAND48` to the value `THREAD` before calling any function in the `drand48` family.

If you do not request thread specific behavior for the `drand48` family, C/370 serializes access to the storage for  $X(n)$ ,  $a$  and  $c$  by functions in the `drand48` family when they are called by a multithreaded application.

If thread specific behavior is requested, and the `drand48()` function is called from thread  $t$ , the `drand48()` function generates the next 48-bit integer value in a sequence of 48-bit integer values,  $X(t,i)$ , for the thread  $t$ . The sequence of values for a thread is generated according to the linear congruential formula:

$$X(t,n+1) = (a(t)X(t,n) + c(t)) \bmod (2^{**}48) \quad n \geq 0$$

The initial values of  $X(t)$ ,  $a(t)$  and  $c(t)$  for the thread  $t$  are:

```
X(t,0) = 1
a(t)   = 5deece66d (base 16)
c(t)   = b         (base 16)
```

C/370 provides storage which is specific to the thread  $t$  to save the most recent 48-bit integer value of the sequence,  $X(t,i)$ , generated by the `drand48()`, `lrand48()` or `mrnd48()` function. The value,  $X(t,n)$ , in this storage may be reinitialized by calling the `lcng48()`, `seed48()` or `srand48()` function from the thread  $t$ . Likewise, the values of  $a(t)$  and  $c(t)$  for thread  $t$  may be changed by calling the `lcng48()` function from the thread. Thereafter, whenever the `seed48()` or `srand48()` function is called from the thread  $t$  to change  $X(t,n)$ , the initial values of  $a(t)$  and  $c(t)$  are also reestablished.

## Returned Value

The `drand48()` function transforms the generated 48-bit value,  $X(n+1)$ , to a double-precision, floating-point value on the interval  $[0.0,1.0)$  and returns this transformed value.

## Special Behavior for OS/390 UNIX Services

If thread specific behavior is requested for the `drand48` family and the `drand48()` function is called on thread  $t$ , the `drand48()` function transforms the generated 48-bit value,  $X(t,n+1)$ , to a double-precision, floating-point value on the interval  $[0.0,1.0)$  and returns this transformed value.

## Related Information

- “`stdio.h`” on page 43
- “`erand48()` — Pseudo-random Number Generator” on page 314
- “`__isBFP()` — Determine Application Floating-Point Format” on page 705
- “`jrand48()` — Pseudo-random Number Generator” on page 724
- “`lcng48()` — Pseudo-random Number Initializer” on page 736
- “`lrnd48()` — Pseudo-random Number Generator” on page 773
- “`mrnd48()` — Pseudo-random Number Generator” on page 843
- “`nrnd48()` — Pseudo-random Number Generator” on page 868
- “`seed48()` — Pseudo-random Number Initializer” on page 1156
- “`srand48()` — Pseudo-random Number Initializer” on page 1401

## dup() — Duplicate an Open File Descriptor

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
int dup(int fildes);
```

### General Description

Returns a new file descriptor that is the lowest numbered available descriptor. The new file descriptor refers to the same open file as *fildes* and shares any locks that may be associated with *fildes*.

The following operations are equivalent:

```
fd = dup(fildes);
fd = fcntl(fildes, F_DUPFD, 0);
```

For further information, see “fcntl() — Control Open File Descriptors” on page 350.

### Returned Value

If successful, dup() returns a new file descriptor. If unsuccessful, it returns –1 and sets errno to one of the following:

**EBADF** *fildes* is not a valid open file descriptor.

**EMFILE** The process has already reached its maximum number of open file descriptors.

### Example CBC3BD05

```
/* CBC3BD05
   This example duplicates an open file descriptor, using dup().
*/
#define _POSIX_SOURCE
#include <errno.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

void print_inode(int fd) {
    struct stat info;
    if (fstat(fd, &info) != 0)
        fprintf(stderr, "fstat() error for fd %d: %s\n", fd, strerror(errno));
    else
        printf("The inode of fd %d is %d\n", fd, (int) info.st_ino);
}

main() {
```

```
int fd;
if ((fd = dup(0)) < 0)
    perror("&dupf error");
else {
    print_inode(0);
    print_inode(fd);
    puts("The file descriptors are different but");
    puts("they point to the same file.");
    close(fd);
}
}
```

### Output

```
The inode of fd 0 is 30
The inode of fd 3 is 30
The file descriptors are different but
they point to the same file.
```

### Related Information

- “unistd.h” on page 53
- “close() — Close a File” on page 191
- “creat() — Create a New File or Rewrite an Existing One” on page 233
- “dup2() — Duplicate an Open File Descriptor to Another” on page 290
- “exec Functions” on page 322
- “fcntl() — Control Open File Descriptors” on page 350
- “open() — Open a File” on page 872
- “pipe() — Create an Unnamed Pipe” on page 907

## dup2() — Duplicate an Open File Descriptor to Another

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
int dup2(int fd1, int fd2);
```

### General Description

Returns a file descriptor with the value *fd2*. *fd2* now refers to the same file as *fd1*, and the file that was previously referred to by *fd2* is closed. The following conditions apply:

- If *fd2* is less than 0 or greater than OPEN\_MAX, dup2() returns -1 and sets errno to EBADF.
- If *fd1* is a valid file descriptor and is equal to *fd2*, dup2() returns *fd2* without closing it.
- If *fd1* is not a valid file descriptor, dup2() fails and does not close *fd2*.
- If a file descriptor does not already exist, dup2() can be used to create one, a duplicate of *fd1*.

### Returned Value

If successful, dup2() returns *fd2*. If unsuccessful, it returns -1 and sets errno to one of the following:

**EBADF**     *fd1* is not a valid file descriptor, or *fd2* is less than 0 or greater than OPEN\_MAX.

**EINTR**     dup2() was interrupted by a signal.

### Example CBC3BD06

```
/* CBC3BD06
   This example duplicates an open file descriptor, using dup2().
*/
#define _POSIX_SOURCE
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

void print_inode(int fd) {
    struct stat info;
    if (fstat(fd, &info) != 0)
        fprintf(stderr, "fstat() error for fd %d: %s\n", fd, strerror(errno));
    else
        printf("The inode of fd %d is %d\n", fd, (int) info.st_ino);
}
```



```

main() {
    int fd;
    char fn[]="dup2.file";

    if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        print_inode(fd);
        if ((fd = dup2(0, fd)) < 0)
            perror("dup2() error");
        else {
            puts("After dup2()...");
            print_inode(0);
            print_inode(fd);
            puts("The file descriptors are different but they");
            puts("point to the same file which is different than");
            puts("the file that the second fd originally pointed to.");
            close(fd);
        }
        unlink(fn);
    }
}

```

### Output

```

The inode of fd 3 is 3031
After dup2()...
The inode of fd 0 is 30
The inode of fd 3 is 30
The file descriptors are different but they
point to the same file which is different than
the file that the second fd originally pointed to.

```

### Related Information

- “unistd.h” on page 53
- “close() — Close a File” on page 191
- “creat() — Create a New File or Rewrite an Existing One” on page 233
- “dup() — Duplicate an Open File Descriptor” on page 288
- “exec Functions” on page 322
- “fcntl() — Control Open File Descriptors” on page 350
- “open() — Open a File” on page 872
- “pipe() — Create an Unnamed Pipe” on page 907

## dynalloc() — Allocate a Data Set

### Standards

Standards / Extensions	C or C++	Dependencies
C/MVS	both	

### Format

```
#include <dynit.h>
int dynalloc(__dyn_t *dyn_parms);
```

### General Description

Dynamically allocates a data set using the SVC 99 service on MVS, by building an SVC 99 parameter list based on parameters specified in *dyn\_parms*. *dynalloc()* corresponds to verb code 1 for SVC 99.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use `LANGLVL(EXTENDED)`.

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with `LANGLVL(EXTENDED)`. When you use `LANGLVL(EXTENDED)` any relevant information in the header is also exposed.

For a description of the `__dyn_t` structure, see Table 18 on page 293. For more information on SVC 99, the SVC 99 extension block, and the text unit keys and values, refer to *OS/390 MVS Programming: Authorized Assembler Services Guide*.

The Request Block Extension and the Error Message Parameter list can be used to process the messages returned by SVC99 when an error occurs. To use this feature you must allocate and initialize these structures using the processes described in the MVS manuals. You must also inform the *dynalloc()* function that they are present by assigning their addresses to `__rbx` or `__msgpar`.

Because additional fields have been added to the `__dyn_t` structure, you should recompile existing source code with the latest *dynit.h* header file to access the new fields.

Some values, such as *ddname* and *dsname*, will be converted to uppercase internally when they are used by the *dynalloc()* function.

To dynamically allocate a data set on MVS, you should:

- Invoke *dyninit()* with a variable of type `__dyn_t`.
- Assign values to the appropriate fields in the variable that will satisfy the *svc99()* request.
- Invoke *dynalloc()* with this variable.

**\_\_dyn\_t Data Structure Elements***Table 18 (Page 1 of 5). Description of \_\_dyn\_t Data Structure Elements*

<b>Element</b>	<b>Text Unit Key</b>	<b>Text Unit Value</b>	<b>Type</b>	<b>Description</b>
__ddname	DALDDNAM	0001	char *	ddname (maximum length of 8) <sup>1</sup> . If 8 question marks (???????) are specified, it means that the request expects a system-generated ddname returned.
__dsname	DALDSNAM	0002	char *	Fully qualified data-set name (maximum length of 44 <sup>1</sup> ).
__sysout	DALYSOU	0018	char	The class of the system output data set (for example, SYSOUT=A). Values are: alphabetic character, asterisk (*), or the macro __DEF_CLASS, to specify the default class.
__sysoutname	DALSPGNM	0019	char *	Program name for sysout. The __sysout field must be specified with this field (maximum length of 44 <sup>1</sup> ).
__member	DALMEMBR	0003	char *	Member of a partitioned data set to be allocated (maximum length of 8 <sup>1</sup> ).
__status	DALSTATS	0004	char	Data set status. Values are: __DISP_OLD, __DISP_NEW, __DISP_MOD, and __DISP_SHR, which are defined in dynit.h.
__normdisp	DALNDISP	0005	char	Specifies the normal disposition of a data set. Values are: __DISP_CATLG, __DISP_UNCATLG, __DISP_DELETE, and __DISP_KEEP, which are defined in dynit.h.
__conddisp	DALCDISP	0006	char	Specifies the conditional disposition of a data set. Values are: __DISP_CATLG, __DISP_UNCATLG, __DISP_DELETE, and __DISP_KEEP, which are defined in dynit.h.
__unit	DALUNIT	0015	char *	Unit name of the device that the data set will (or does, if it already exists) reside on (maximum length of 8 <sup>1</sup> ).
__volser	DALVLSER	0010	char *	Volume serial number of the device a data set will (or does, if it already exists) reside on (maximum length of 6 <sup>1</sup> ).

Table 18 (Page 2 of 5). Description of \_\_dyn\_t Data Structure Elements

Element	Text Unit Key	Text Unit Value	Type	Description
__dsorg	DALDSORG	003C	char	<p>Data-set organization of a data set. Values are:</p> <p>__DSORG_unknown Unknown</p> <p>__DSORG_VSAM VSAM</p> <p>__DSORG_GS Graphics</p> <p>__DSORG_PO Partitioned organization</p> <p>__DSORG_POU Partitioned organization unmovable</p> <p>__DSORG_DA Direct access</p> <p>__DSORG_DAU Direct access unmovable</p> <p>__DSORG_PS Physical sequential</p> <p>__DSORG_PSU Physical sequential unmovable.</p>
__alcunit	DALCYL, DALTRK	0008, 0007	char	Unit of space allocation for a data set. Values are: __CYL and __TRK. To specify allocation units in blocks, use the field __avgbk.
__primary	DALPRIME	000A	int	Primary space allocation for a data set.
__secondary	DALSECND	000B	int	Secondary space allocation for a data set.
__dirblk	DALDIR	000C	int	Number of directory blocks for a partitioned data set.
__avgbk	DALBLKLN	0009	int	Specifies the unit of space allocation to be blocks and sets the average block length.
__recfm		0049	short	<p>Record format of a data set. The following macros in dynit.h can be added together to determine the __recfm value:</p> <p><b>_M_</b> Machine-code printer-control characters</p> <p><b>_A_</b> ASA printer-control characters</p> <p><b>_S_</b> Standard fixed, spanned variable</p> <p><b>_B_</b> Blocked</p> <p><b>_D_</b> Variable ASCII records</p> <p><b>_V_</b> Variable</p> <p><b>_F_</b> Fixed</p> <p><b>_U_</b> Undefined</p> <p><b>_FB_</b> Fixed blocked</p> <p><b>_VB_</b> Variable blocked</p> <p><b>_FBS_</b> Fixed blocked standard</p> <p><b>_VBS_</b> Variable blocked standard.</p> <p>e.g., to specify a recfm of FBA, set: __recfm = <b>_FB_</b> + <b>_A_</b></p>
__blksize	DALBLKSZ	0030	short	Block size of a data set.

Table 18 (Page 3 of 5). Description of \_\_dyn\_t Data Structure Elements

Element	Text Unit Key	Text Unit Value	Type	Description
__lrecl	DALLRECL	0042	unsigned short	Record length of a data set.
__volrefds	DALLVLRDS	0014	char *	Fully qualified name of a cataloged data set to be used as a model for obtaining volume serial information (maximum length of 441).
__dcbrefds	DALDCBDS	002C	char *	Fully qualified name of a cataloged data set to be used as a model for obtaining DCB information (maximum length of 441).
__dcbrefdd	DALLDCBDD	002D	char *	ddname of a data set to be used as a model for obtaining DCB information (maximum length of 91).
__misc_flags			unsigned char	Specifies the attributes. See example CBC3BD07 for instructions on how to specify the flags shown below using a logical   (OR).
__CLOSE	DALCLOSE	001C	unsigned char	A flag: deallocate data set when file is closed.
__RELEASE	DALRLSE	000D	unsigned char	A flag: release unused space when file is closed.
__CONTIG	DALSPRFRM	000E	unsigned char	A flag: allocate space contiguously.
__ROUND	DALROUND	000F	unsigned char	A flag: allocate space in whole cylinders when blocks are requested.
__TERM	DALTERM	0028	unsigned char	A flag: time-sharing terminal is to be used as I/O device.
__DUMMY_DSN	DALDUMMY	0024	unsigned char	A flag: dummy data set is to be allocated.
__HOLDQ	DALSHOLD	0059	unsigned char	A flag: hold queue routing for sysout data set.
__PERM	DALPERMA	0052	unsigned char	A flag: set permanent allocation attribute.
__password	DALPASSW	0050	char *	Password for a password-protected data set. The dsname field must be specified with this field (maximum length of 81).
__miscitems			char **	For all other text unit keys not available in __dyn_t, this pointer will let you specify an array of text unit strings. If you specify this field, you must turn the high bit on the last item (as in svc99()). Use the bitwise inclusive or ( ) operand with the last item and the hexadecimal value 0x80000000.
__infocode			short	Returns the information code returned by the MVS dynamic allocation functions. For more information, refer to <i>OS/390 MVS Programming: Authorized Assembler Services Guide</i> .
__errcode			short	Returns the error code returned by the MVS dynamic allocation functions. For more information, refer to <i>OS/390 MVS Programming: Authorized Assembler Services Guide</i> .
__storclass	DALSTCL	8004	char *	Specifies the storage class of system managed storage.

Table 18 (Page 4 of 5). Description of \_\_dyn\_t Data Structure Elements

Element	Text Unit Key	Text Unit Value	Type	Description
__mgntclass	DALMGCL	8005	char *	Specifies the management class of a data set.
__dataclass	DALDACL	8006	char *	Specifies the data class of a data set.
__recorg	DALRECO	800B	char	Specifies the record organization of a VSAM data set. Values are: __KS, __ES, __RR, __LS.
__keyoffset	DALKEYO	800C	short	Specifies the key offset. The position of the first byte of the key in records of the specified VSAM data set.
__keylength	DALKYLEN	0040	short	Specifies the length in bytes of the keys used in the data set.
__refdd	DALREFD	800D	char *	Specifies the name of the JCL DD statement from which the attributes are to be copied.
__like	DALLIKE	800F	char *	Specifies the name of the model data set from which the attributes are to be copied.
__dsntype	DALDSNT	8012	char	Specifies the type attributes of a data set. Valid types include __DSNT_HFS, __DSNT_LIBRARY, __DSNT_PDS, and __DSNT_PIPE.
__pathname	DALPATH	8017	char *	Path name (maximum length is 2551). See <i>OS/390 UNIX System Services User's Guide</i> for path name format.
__pathopts	DALPOPT	8018	int	Specifies file options for the HFS file. Values are: __PATH_OCREAT, __PATH_OAPPEND, __PATH_OEXCL, __PATH_ONOCTTY, __PATH_OTRUNC, __PATH_ONONBLOCK, __PATH_ORDONLY, __PATH_OWONLY, __PATH_ORDWR. For information on the file options, refer to <i>OS/390 MVS JCL Reference</i> . For information on DYNALLOC, refer to <i>OS/390 MVS Programming: Authorized Assembler Services Guide</i> .
__pathmode	DALPMDE	8019	int	Specifies the file access attributes for the HFS file. Values are: __PATH_SIRUSR, __PATH_SIWUSR, __PATH_SIXUSR, __PATH_SIRWXU, __PATH_SIRGRP, __PATH_SIWGRP, __PATH_SIXGRP, __PATH_SIRWXG, __PATH_SIROTH, __PATH_SIWOTH, __PATH_SIXOTH, __PATH_SIRWXS, __PATH_SISUID, __PATH_SISGID. For information on the file attributes, refer to <i>OS/390 MVS JCL Reference</i> . For information on DYNALLOC, refer to <i>OS/390 MVS Programming: Authorized Assembler Services Guide</i> .
__pathndisp	DALPNDS	801A	char	Specifies the normal HFS file disposition desired. It is either __DISP_KEEP or __DISP_DELETE

Table 18 (Page 5 of 5). Description of \_\_dyn\_t Data Structure Elements

Element	Text Unit Key	Text Unit Value	Type	Description
__pathcdisp	DALPCDS	801B	char	Specifies the abnormal HFS file disposition desired. It is either __DISP_KEEP or __DISP_DELETE
__rbx			s99rbx_t *	For users who make use of the Request Block Extension.
__emsgparmlist			__s99emparms_t *	For users who want to process associated messages with the dynamic allocation.
__rls	DALRLS	801C	char	Specifies the type of record level sharing (RLS) being done for a specific data set. The valid values are __RLS_NRI and __RLS_CR. Refer to the <i>OS/390 C/C++ Programming Guide</i> and <i>DFSMS/MVS Using Data Sets</i> for a description of these VSAM RLS access modes.

[1] If an element exceeds its maximum allowable length, it is truncated to that length.

### Special Behavior for POSIX C

For POSIX C programs, allocations established by dynalloc() persist neither after an exec nor in the child process after fork(). See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information about using POSIX support.

### Returned Value

Under MVS, dynalloc() returns 0 if successful. If SVC 99 is not supported on your system, or if a text string passed to SVC 99 cannot be built from a field in the *dyn\_parms* structure, a negative value is returned. The value -1 is returned if there is not sufficient storage to process all the text units. Otherwise, the return code is the value returned from SVC 99, and the error and information codes are found in those fields of the *dyn\_parms* structure. For example, if you pass NULL to dynalloc(), the return code is nonzero.

For more information on return codes, refer to *OS/390 MVS Programming: Authorized Assembler Services Guide*.

### Example CBC3BD07

```

/* CBC3BD07
   This example dynamically allocates a data set.
*/
#include <dynit.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <dynit.h>

int main() {

```

```

__dyn_t ip;

dyninit(&ip);                                /* Initialize control block */

ip.__ddname="MYDD";                          /* //MYDD DD */
ip.__dsname="MYUSERID.MYDSN";               /* // DSN=MYUSERID.MYDSN, */
ip.__status=__DISP_NEW;                      /* // DISP=(NEW, */
ip.__normdisp=__DISP_CATLG;                 /* // CATLG), */
ip.__alcunit=__CYL;                         /* // SPACE=(CYL, */
ip.__primary=2;                             /* // (2, */
ip.__secondary=1;                           /* // 1), */
ip.__misc_flags=__RELEASE |                 /* // RLSE, */
                 __CONTIG;                  /* // CONTIG), */
ip.__dsorg=__DSORG_PS;                      /* // DSORG=PS, */
ip.__recfm=__F + __B + __A;                 /* // RECFM=FBA, */
ip.__lrecl=121;                             /* // LRECL=121, */
ip.__blksize=12100;                         /* // BLKSIZE=12100 */

if (dynalloc(&ip)!=0) {
    printf("Dynalloc failed with error code=%hX, info code=%hX\n",
           ip.__errcode,ip.__infocode);
}
}

```

## Related Information

- “dynit.h” on page 25
- “dynfree() — Deallocate a Data Set” on page 299
- “dyninit() — Initialize \_\_dyn\_t Structure” on page 301
- “svc99() — Access Supervisor Call” on page 1467



## dynfree() — Deallocate a Data Set

### Standards

Standards / Extensions	C or C++	Dependencies
C/MVS	both	

### Format

```
#include <dynit.h>
```

```
int dynfree(__dyn_t *dyn_parms);
```

### General Description

Dynamically deallocates an OS/390 data set in accordance with the attributes defined in *dyn\_parms*.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

The only fields in *\_\_dyn\_t* that are used by *dynfree()* are:

```
char *__ddname
char *__dsname
char *__member
char *__pathname
char __normdisp
char __pathndisp
char **__miscitems
```

If any other fields are specified, they will be ignored. For more information on the *\_\_dyn\_t* structure, see Table 18 on page 293.

To dynamically deallocate a data set on OS/390, you should:

- Invoke *dyninit()* with a variable of type *\_\_dyn\_t*
- Assign values to the appropriate fields that will satisfy the *svc99()* request
- Invoke *dynfree()* with this variable.

### Returned Value

Under OS/390, *dynfree()* returns 0 if successful and nonzero otherwise. The value -1 is returned if there is not sufficient storage to process all the text units.

**Example**

```
/*
   This example dynamically deallocates a data set.
 */
#include <dynit.h>

int main(void) {
    :
    __dyn_t ip;
    :
    dyninit(ip);
    ip.__ddname = "mydd";
    :
    dynfree(&ip);
}
```

**Related Information**

- “dynit.h” on page 25
- “dynalloc() — Allocate a Data Set” on page 292
- “dyninit() — Initialize \_\_dyn\_t Structure” on page 301
- “svc99() — Access Supervisor Call” on page 1467

## dyninit() — Initialize \_\_dyn\_t Structure

### Standards

Standards / Extensions	C or C++	Dependencies
C/MVS	both	

### Format

```
#include <dynit.h>
```

```
int dyninit(__dyn_t *dyn_parms);
```

### General Description

Initializes the \_\_dyn\_t structure that is used to build the parameter lists that are passed to the dynalloc() function and the dynfree() function. If you do not initialize the \_\_dyn\_t structure using dyninit(), undefined behavior may result.

The \_\_dyn\_t structure is defined in the dynit.h header file. A description of the elements is found in “dynalloc() — Allocate a Data Set” on page 292.

### Returned Value

Under MVS, dyninit() returns 0 if successful and a nonzero value otherwise.

### Example CBC3BD09

```
/* CBC3BD09
   This example initializes a __dyn_t structure, called ip.
*/
#include <stdio.h>
#include <string.h>
#include <dynit.h>

main() {
    char dsn[]="USER.TEST.DATASET";
    __dyn_t ip;
    int ret;

    dyninit(&ip);
    ip.__ddname = "TEST";
    ip.__dsname = dsn;
    ip.__status = __DISP_NEW;
    ip.__normdisp = __DISP_DELETE;
    ip.__alcunit = __TRK;
    ip.__primary = 1;
    ip.__unit = "SYSALLDA";

    if ((ret = dynalloc(&ip)) != 0)
        printf("dynalloc() ret=%d, error code %04x, info code %04x\n",
            ret, ip.__errcode, ip.__infocode);

    else {
        dyninit(&ip);
        ip.__ddname = "TEST";

        if ((ret = dynfree(&ip)) != 0)
            printf("dynfree() ret=%d, error code %04x, info code %04x\n",
                ret, ip.__errcode, ip.__infocode);
    }
}
```

```
        else puts("success!");  
    }  
}
```

**Related Information**

- “dynit.h” on page 25
- “dynalloc() — Allocate a Data Set” on page 292
- “dynfree() — Deallocate a Data Set” on page 299
- “svc99() — Access Supervisor Call” on page 1467

## ecvt() — Convert Double to String

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>
```

```
char *ecvt(double x, int ndigit,
           int *decpt, int *sign);
```

### General Description

The `ecvt()` function converts double floating-point argument values to floating-point output strings. The `ecvt()` function has been extended to determine the floating-point format (hexadecimal floating-point or IEEE floating-point) of double argument values by using `__isBFP()`.

OS/390 (C/C++) formatted output functions, including the `ecvt()` function, convert IEEE floating-point infinity and NaN argument values to special infinity and NaN floating-point number output sequences. See “*printf* Family of Formatted Output Functions” on page 441 for a description of the special infinity and Nan output sequences.

The `ecvt()` function converts `x` to a null-terminated string of *ndigit* digits (where *ndigit* is reduced to an unspecified limit determined by the precision of a double) and returns a pointer to the string. The high-order digit is nonzero, unless the value is 0. The low-order digit is rounded. The position of the radix character relative to the beginning of the string is stored in the integer pointed to by *decpt* (negative means left of the returned digits). The radix character is not included in the returned string. If the sign of the result is negative, the integer pointed to by *sign* is nonzero, otherwise it is 0.

The function returns a pointer to a buffer used only by the calling thread which may be overwritten by subsequent calls to `ecvt()`, “*fcvt()* — Convert Double to String” on page 358 and “*gcvt()* — Convert Double to String” on page 498.

If the converted value is out of range or is not representable, the function returns NULL.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

If it succeeds, `ecvt()` returns the character equivalent of `x` as specified above.

If it is unable to allocate the return buffer, or the conversion fails, `ecvt()` returns NULL.

### **Related Information**

- “stdlib.h” on page 45
- “fcvt() — Convert Double to String” on page 358
- “gcvt() — Convert Double to String” on page 498
- “gcvt() — Convert Double to String” on page 498
- “\_\_isBFP() — Determine Application Floating-Point Format” on page 705

## encrypt() — Encoding Function

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <unistd.h>
```

```
void encrypt(char block[64], int edflag);
```

### General Description

The `encrypt()` function uses an array of 16 48-bit keys produced by the `setkey()` function to encode bytes specified by the *block* argument according to the Data Encryption Standard (DES) encryption algorithm or to decode argument bytes according to the DES decryption algorithm.

The *block* argument of `encrypt()` is an array of length 64 bytes containing only the bytes with numerical value of 0 and 1. The array is modified in place using keys produced by `setkey()`. If *edflag* is 0, the argument is encoded using the DES encryption algorithm. If *edflag* is 1 and the OS/390 Language Environment Decryption feature is installed, the argument is decoded using the DES decryption algorithm. If the OS/390 Language Environment Decryption feature is not installed, decoding fails.

### Special Behavior for OS/390 UNIX Services

The `encrypt()` function is thread specific. Thus, for each thread from which the `encrypt()` function is called by a threaded application, the `setkey()` function must first be called from the thread to establish a DES key array for the thread.

### Returned Value

No values are returned by the `encrypt()` function.

### Special Behavior for OS/390 UNIX Services

The `encrypt()` function will fail if:

**ENOMEM** If `setkey()` has not been called or failed to produce a DES key array for the thread from which `encrypt()` is called.

**EINVAL** 64 byte input array contains bytes with values other than 0x00 or 0x01.

**ENOSYS** If DES key array exists for thread from which `encrypt()` is called to decode data but the OS/390 Language Environment Decryption feature is not installed.

**Note:** Because `encrypt()` does not return a value, applications wishing to check for errors should set `errno` to 0, call `encrypt()`, then test `errno` and, if it is nonzero, assume an error has occurred.

**Related Information**

- “unistd.h” on page 53
- “crypt() — String Encoding Function” on page 238



## endgrent() — Group Database Entry Functions

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <grp.h>
void endgrent (void),
struct group *getgrent (void);
void setgrent (void);
```

### General Description

The `getgrent()` function returns a pointer to the broken-out fields of a line in the group database, mapped by the **group** structure defined in the `<grp.h>` header file. Repeated calls to `getgrent()` return a pointer to the next **group** structure in the database, until end of file, at which point a null pointer is returned. `setgrent()` interrupts this sequential search and rewinds the user database to the beginning, such that the next `getgrent()` returns a pointer to the first **group** structure. Use of `setgrent()` is optional after an end of file, as the next `getgrent()` after end of file again returns a pointer to the first **group** structure. `endgrent()` is optionally used to close the user database when searching is complete.

The `setgrent()` function effectively rewinds the group database to allow repeated searches.

The `endgrent()` function may be called to close the group database when processing is complete.

### Returned Value

When first called, `getgrent()` returns a pointer to the next group structure in the group database. Upon subsequent calls it returns a pointer to a group structure, or it returns a null pointer on either end-of-file or an error. The return value may point to static data that is overwritten by each call. There are no documented errors for this function.

### Related Information

- “`getgrgid()` — Access the Group Database by ID” on page 523
- “`getgrnam()` — Access the Group Database by Name” on page 525
- “`getlogin()` — Get the User Login Name” on page 550
- “`getpwent()` — Get User Database Entry” on page 586
- “`getpwnam()` — Access the User Database by User Name” on page 587
- “`getpwuid()` — Access the User Database by User ID” on page 589

## endhostent() — Work with a Host Entry

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>
```

```
void endhostent(void);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <netdb.h>
```

```
void endhostent();
```

### General Description

The `endhostent()` call closes the `/etc/hosts` or the `tcpip.HOSTS.SITEINFO` data set, which contains information about known hosts.

You can use the **X\_SITE** environment variable to specify a data set other than `tcpip.HOSTS.SITEINFO`. For more information on these data sets and environment variables, see *OS/390 UNIX System Services Planning*.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Related Information

- “`gethostbyaddr()` — Get a Host Entry by Address” on page 531
- “`gethostbyname()` — Get a Host Entry by Name” on page 534
- “`gethostent()` — Get the Next Host Entry” on page 537
- “`sethostent()` — Open the Host Information Data Set” on page 1224

## endnetent() — Close Network Information Data Sets

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>
```

```
void endnetent(void);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <netdb.h>
```

```
void endnetent();
```

### General Description

The `endnetent()` call closes the `tcpip.HOSTS.ADDRINFO` data set. The `tcpip.HOSTS.ADDRINFO` data set contains information about known networks.

You can use the **X\_ADDR** environment variable to specify a data set other than `tcpip.HOSTS.ADDRINFO`. For more information on these data sets and environment variables, see *OS/390 UNIX System Services Planning*.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Related Information

- “`getnetbyaddr()` — Get a Network Entry by Address” on page 558
- “`getnetbyname()` — Get a Network Entry by Name” on page 560
- “`getnetent()` — Get the Next Network Entry” on page 562
- “`setnetent()` — Open the Network Information Data Set” on page 1251

## endprotoent() — Work with a Protocol Entry

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>
```

```
void endprotoent(void);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <netdb.h>
```

```
void endprotoent();
```

### General Description

The `endprotoent()` call closes the */etc/protocol* or the *tcpip.ETC.PROTO* data set, which contains information about the networking protocols (IP, ICMP, TCP, and UDP).

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Related Information

- “`getprotobyname()` — Get a Protocol Entry by Name” on page 580
- “`getprotoent()` — Get the Next Protocol Entry” on page 584
- “`setprotoent()` — Open the Protocol Information Data Set” on page 1259

## endpwent() — User Database Functions

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <pwd.h>
void endpwent (void),
struct passwd *getpwent(void);
void setpwent (void);
```

### General Description

The `getpwent()` function returns a pointer to the broken-out fields of a line in the user database, mapped by the **passwd** structure defined in the `<pwd.h>` header file. Repeated calls to `getpwent()` return a pointer to the next **passwd** structure in the database, until end of file, at which point a null pointer is returned. `setpwent()` interrupts this sequential search and rewinds the user database to the beginning, such that the next `getpwent()` returns a pointer to the first **passwd** structure. Use of `setpwent()` is optional after an end of file, as the next `getpwent()` after end of file again returns a pointer to the first **passwd** structure. `endpwent()` is optionally used to close the user database when searching is complete.

The `setpwent()` function effectively rewinds the user database to allow repeated searches.

The `endpwent()` function may be called to close the user database when processing is complete.

### Returned Value

When first called, `getpwent()` returns a pointer to the next **passwd** structure in the user database. Upon subsequent calls it returns a pointer to a **passwd** structure, or it returns a null pointer on either end-of-file or an error. The return value may point to static data that is overwritten by each call. There are no documented errors for this function.

### Related Information

- “`getgrent()` — Get Group Database Entry” on page 522
- “`getgrgid()` — Access the Group Database by ID” on page 523
- “`getgrnam()` — Access the Group Database by Name” on page 525
- “`getpwnam()` — Access the User Database by User Name” on page 587
- “`getlogin()` — Get the User Login Name” on page 550
- “`getpwent()` — Get User Database Entry” on page 586
- “`getpwuid()` — Access the User Database by User ID” on page 589

## endservent() — Close Network Services Information Data Sets

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>
```

```
void endservent(void);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <netdb.h>
```

```
void endservent();
```

### General Description

The `endservent()` call closes the */etc/services* or the *tcpip.ETC.SERVICES* data set, which contains information about network services. Example services are name server, File Transfer Protocol (FTP), and telnet.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Related Information

- “`getservbyname()` — Get a Server Entry by Name” on page 597
- “`getservbyport()` — Get a Service Entry by Port” on page 599
- “`getservent()` — Get the Next Service Entry” on page 601
- “`setservent()` — Open the Network Services Information Data Set” on page 1267

## endutxent() — Close the utmpx Database

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <utmpx.h>
```

```
void endutxent(void);
```

### General Description

The `endutxent()` function closes the utmpx database for the current thread. The database may be opened by `getutxent()`, `getutxid()`, `getutxline()`, or `pututxline()`.

Because the `endutxent()` function processes thread specific data the `endutxent()` function can be used safely from a multi-threaded application. If multiple threads in the same process open the database, then each thread opens the database with a different file descriptor. The thread's database file descriptor is closed when the calling thread terminates or the `endutxent()` function is called by the calling thread.

Programs must not reference the data passed back by `getutxline()`, `getutxid()`, `getutxent()`, or `pututxline()` after `endutxent()` has been called (the storage has been freed.)

After `getutxline()`, `getutxent()`, `getutxid()`, or `pututxline()`, the utmpx database is open. No other process can do `pututxline()` to this utmpx database until this process issues `endutxent()` or `__utmpxname()` to close the utmpx database, or this process ends. You can cause all OS/390 UNIX user logins/logouts to hang if you fail to `exit()` or issue `endutxent()` or `__utmpxname()`, and you have the main `/etc/utmpx` database open in your process. `endutxent()` resets the name of the next utmpx file to open back to the default. If you want to do additional utmpx operations using a non-standard utmpx file name, you must re-issue `__utmpxname()` after closing the utmpx database with `endutxent()`.

### Returned Value

The `endutxent()` function returns no value.

### Related Information

- “`__utmpxname()` — Change the utmpx Database Name” on page 1669
- “`getutxent()` — Read Next Entry in utmpx Database” on page 620
- “`getutxline()` — Search by Line utmpx Database” on page 624
- “`getutxid()` — Search by ID utmpx Database” on page 622
- “`pututxline()` — Write Entry to utmpx Database” on page 1061
- “`setutxent()` — Reset to Start of utmpx Database” on page 1282

## erand48() — Pseudo-random Number Generator

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>
```

```
double erand48(unsigned short int x16v[3]);
```

### General Description

The drand48(), erand48(), jrand48(), lrand48(), mrand48() and nrand48() functions generate uniformly distributed pseudorandom numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The functions drand48() and erand48() return non-negative, double-precision, floating-point values, uniformly distributed over the interval [0.0,1.0]. These functions have been extended to determine the floating-point format (hexadecimal floating-point or IEEE floating-point) of the returned value using the \_\_isBFP() function.

The functions drand48() and erand48() return non-negative, double-precision, floating-point values, uniformly distributed over the interval [0.0,1.0].

The functions lrand48() and nrand48() return non-negative, long integers, uniformly distributed over the interval [0,2\*\*31].

The functions mrand48() and jrand48() return signed long integers, uniformly distributed over the interval [-2\*\*31,2\*\*31].

The erand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(i), according to the linear congruential formula:

$$X(n+1) = (aX(n) + c) \bmod (2^{**}48) \quad n \geq 0$$

The erand48() function uses storage provided by the argument array, x16v[3], to save the most recent 48-bit integer value in the sequence, X(i). The erand48() function uses x16v[0] for the low order (rightmost) 16 bits, x16v[1] for the middle order 16 bits, and x16v[2] for the high order 16 bits of this value.

The initial values of a, and c are:

```
a   = 5deece66d (base 16)
c   = b         (base 16)
```

The values a and c, may be changed by calling the lcong48() function. The initial values of a and c are restored if either the seed48() or srand48() function is called.

### Special Behavior for OS/390 UNIX Services



You can make the `erand48()` function and other functions in the `drand48` family thread specific by setting the environment variable `_RAND48` to the value `THREAD` before calling any function in the `drand48` family.

If you do not request thread specific behavior for the `drand48` family, C/370 serializes access to the storage for `X(n)`, `a` and `c` by functions in the `drand48` family when they are called by a multithreaded application.

If thread specific behavior is requested and the `erand48()` function is called from thread `t`, the `erand48()` function generates the next 48-bit integer value in a sequence of 48-bit integer values, `X(t,i)`, for the thread according to the linear congruential formula:

$$X(t,n+1) = (a(t)X(t,n) + c(t)) \bmod (2^{**}48) \quad n \geq 0$$

The `erand48()` function uses storage provided by the argument array, `x16v[3]`, to save the most recent 48-bit integer value in the sequence, `X(t,i)`. The `erand48()` function uses `x16v[0]` for the low order (rightmost) 16 bits, `x16v[1]` for the middle order 16 bits, and `x16v[2]` for the high order 16 bits of this value.

The initial values of `a(t)` and `c(t)` on the thread `t` are:

```
a(t)   = 5deece66d (base 16)
c(t)   = b          (base 16)
```

The values `a(t)` and `c(t)` may be changed by calling the `lcg48()` function from the thread `t`. The initial values of `a(t)` and `c(t)` are restored if either the `seed48()` or `srand48()` function is called from the thread.

## Returned Value

The `erand48()` function saves the generated 48-bit value, `X(n+1)`, in storage provided by the argument array, `x16v[3]`. The `erand48()` function transforms the generated 48-bit value to a double-precision, floating-point value on the interval `[0.0,1.0]` and returns this transformed value.

## Special Behavior for OS/390 UNIX Services

If thread specific behavior is requested for the `drand48` family and the `erand48()` function is called on thread `t`, the `erand48()` function saves the generated 48-bit value, `X(t,n+1)`, in storage provided by the argument array, `x16v[3]`. The `erand48()` function transforms the generated 48-bit value to a double-precision, floating-point value on the interval `[0.0,1.0]` and returns this transformed value.

## Related Information

- “`stdlib.h`” on page 45
- “`drand48()` — Pseudo-random Number Generator” on page 286
- “`__isBFP()` — Determine Application Floating-Point Format” on page 705
- “`jrand48()` — Pseudo-random Number Generator” on page 724
- “`lcg48()` — Pseudo-random Number Initializer” on page 736
- “`lrand48()` — Pseudo-random Number Generator” on page 773
- “`mrnd48()` — Pseudo-random Number Generator” on page 843
- “`nrnd48()` — Pseudo-random Number Generator” on page 868
- “`seed48()` — Pseudo-random Number Initializer” on page 1156
- “`srand48()` — Pseudo-random Number Initializer” on page 1401

## erf() - erfc() — Calculate Error and Complementary Error Functions

### Standards

Standards / Extensions	C or C++	Dependencies
SAA XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double erf(double x);
double erfc(double x);
```

### Compiler Option

LANGVLV(EXTENDED), LANGVLV(SAA), or LANGVLV(SAA2)

### General Description

Calculates the error function:

$$2\pi^{-1/2} \int_0^x e^{-t^2} dt$$

Because the erfc() function calculates the value of  $1.0 - \text{erf}(x)$ , it is used in place of erf() for large values of  $x$ .

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

Both erf() and erfc() return the calculated value. If the correct value would cause underflow, 0 is returned and the value of the macro ERANGE is stored in errno.

### Example

#### CBC3BE01

```
/* CBC3BE01
   This example uses erf() and erfc() to compute the error function of two
   numbers.
*/
#include <stdio.h>
#include <math.h>

double smallx, largex, value;

int main(void)
{
    smallx = 0.1;
    largex = 10.0;

    value = erf(smallx);          /* value = 0.112463 */
```

```
printf("Error value for 0.1: %f\n", value);

value = erfc(largex);          /* value = 2.088488e-45 */
printf("Error value for 10.0: %e\n", value);
}
```

### Output

```
Error value for 0.1: 0.112463
Error value for 10.0: 2.088488e-45
```

### Related Information

- “math.h” on page 35
- “gamma() — Calculate Gamma Function” on page 497
- “j0() - j1() - jn() — Bessel Functions of the First Kind” on page 726
- “y0() - y1() - yn() — Bessel Functions of the Second Kind” on page 1793

## \_\_err2ad() — Return Address of Reason Code of Last Failure

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
int * __err2ad(void);
```

### General Description

The \_\_err2ad() function returns the address of the reason code of the last failing OS/390 UNIX system call serviced by the OS/390 C Library.

\_\_err2ad() provides assistance in diagnosing problems by allowing an application to reset the value returned by \_\_errno2() before a function is invoked.

### Returned Value

The returned value is intended for diagnostic display purposes *only*. The \_\_err2ad() function call is always successful.

### Related Information

- “\_\_errno2() — Return Reason Code Information” on page 319

## \_\_errno2() — Return Reason Code Information

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
int __errno2(void);
```

### General Description

Returns the reason code of the last failing OS/390 UNIX system service called by the OS/390 C Library. It provides OS/390 UNIX programs access to diagnostic information returned to C from an underlying OS/390 UNIX callable service.

`__errno2()` is provided to help you determine more about the cause of failure of your application program. Your program should not check for specific values from `__errno2()` and make processing decisions based on the value. Use it for diagnostic purposes only. If you use `__errno2()` in programs you expect to port to other platforms, you should use `#ifdef/#endif` logic to avoid calling the function on other platforms.

### Returned Value

The returned value is intended for diagnostic display purposes *only*. The `__errno2()` function call is always successful.

For information on the contents of the integer returned, you will find extensive information in *OS/390 UNIX System Services Programming: Assembler Callable Services Reference*.

### Example

#### CBC3BE02

```
/* CBC3BE02
   This example's output only occurs if the buffer is flushed.
*/
#include <errno.h>
#include <stdio.h>
FILE *myfopen(const char *fn, const char *mode) {
    FILE *f;
    f = fopen(fn,mode);
    if (f==NULL) {
        perror("fopen() failed");
        printf("__errno2 = %08x\n", __errno2());
    }
    return(f);
}
```

### Related Information

- “errno.h” on page 25
- “\_\_err2ad() — Return Address of Reason Code of Last Failure” on page 318

## \_\_etoa() — EBCDIC to ISO8859-1 String Conversion

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#include <unistd.h>
```

```
int __etoa(char *string);
```

### General Description

The `__etoa()` function converts an ebcdic character string *string* to its ISO8859-1 equivalent. The conversion is performed using the codeset page associated with the current locale. The input character string up to, but not including, the null character is changed from the current locale to an ISO8859-1 representation.

The argument, *string*, points to the ebcdic character string to be converted to its ISO8859-1 equivalent.

### Returned Value

If successful, `__etoa()` converts the input ebcdic string to its equivalent ISO8859-1 value, and returns the length of the converted string.

If unsuccessful, `__etoa()` returns `-1`, and stores the error value in `errno`. The following are the possible values of `errno`: (This function internally may call `iconv_open()` and `iconv()`. The `errno`'s returned by these functions are propagated without modification.)

**EINVAL** The current locale does not describe a single-byte character set.

**ENOMEM** There is insufficient storage to complete the conversion process.

### Related Information

- “sys/msg.h” on page 48

## \_\_etoa\_l() — EBCDIC to ISO8859-1 Conversion Operation

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#include <unistd.h>
```

```
int __etoa_l(char *bufferptr, int leng);
```

### General Description

The `__etoa_l()` function converts *leng* ebcdic bytes in the buffer pointed to by *bufferptr* to their ISO8859-1 equivalent. The conversion is performed using the codeset page associated with the current locale.

The argument, *bufferptr*, points to a buffer containing the ebcdic bytes to be converted to their ISO8859-1 equivalent. The input buffer is treated as a sequence of bytes, and all bytes in the input buffer are converted, including any imbedded nulls.

### Returned Value

If successful, `__etoa_l()` converts the input ebcdic bytes to their equivalent ISO8859-1 value, and returns the number of bytes converted.

If unsuccessful, `__etoa_l()` returns `-1`, and returns the error value in `errno`. The following are the possible values of `errno`: (This function may internally call `iconv_open()` and `iconv()`. The `errno`'s returned by these functions are propagated without modification.)

**EINVAL** The current locale does not describe a single-byte character set.

**ENOMEM** There is insufficient storage to complete the conversion process.

### Related Information

- “sys/msg.h” on page 48

exec Functions

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

Format

```
#define _POSIX_SOURCE
#include <unistd.h>
extern char **environ;

int execl(const char *path, const char *arg, ..., NULL);
int execl(const char *path, const char *arg, ..., NULL,
          char *const envp[]);
int execlp(const char *file, const char *arg, ..., NULL);
int execl(const char *path, char *const argv[]);
int execve(const char *path, char *const argv[],
          char *const envp[]);
int execvp(const char *file, char *const argv[]);
```

**Note:** Although POSIX.1 does not require that the `unistd.h` include file be included, it is recommended that you include it for portability.

General Description

All `exec` functions run a new program by replacing the current process image with a new process image obtained from a file in the HFS (hierarchical file system).

For information on specifying names for MVS data sets and HFS files, see the *OS/390 C/C++ Programming Guide*.

| To provide an ASCII input/output format for applications using `execv()`, `execve()` or  
| `execvp()` functions, define feature test macro `__LIBASCII` as described on page 22.

A successful `exec` function never returns control because the calling process is overwritten with the new process.

The argument *path* is a string giving the absolute or relative path name of a file. This file contains the image of the process to be run.

*file* is a string that is used in determining the path name of the file containing the image of the process to be run. If *file* contains a slash character (/), it is assumed to be the absolute or relative path name of the file. If *file* does not contain a slash, the system searches for the given file name under the list of directories given by the `PATH` environment variable. The system checks under directories in the order they appear in the `PATH` variable, and executes the first file whose name matches the *file* string. The file must reside in the HFS.

The `exec` functions use the following environment variables:

**STEPLIB** Supports the creation and propagation of a STEPLIB environment to the new process image. The following are the accepted values for the STEPLIB environment variable and the actions taken for each value:



- STEPLIB=NONE. No Steplib DD is to be created for the new process image.
- STEPLIB=CURRENT. The TASKLIB, STEPLIB or JOBLIB DD data set allocations that are active for the calling task at the time of the call to exec() are propagated to the new process image, if they are found to be cataloged. Uncataloged data sets are not propagated to the new process image.
- STEPLIB=Dsn1:Dsn2:...DsnN. The specified data sets, Dsn1:Dsn2:...DsnN, are built into a STEPLIB DD in the new process image.

**Note:** The actual name of the DD is not STEPLIB, but is a system-generated name that has the same effect as a STEPLIB DD. The data sets are concatenated in the order specified. The specified data sets must follow standard MVS data set naming conventions. Data sets found to be in violation of this standard are ignored. If the data sets do follow the standard, but:

- The caller does not have the proper security access to a data set
- A data set is uncataloged or is not in load library format

then the data set is ignored. Because the data sets in error are ignored, the executable file may run without the proper STEPLIB environment. If a data set is in error due to improper security access, a X'913' abend is generated. The dump for this abend can be suppressed by your installation.

If the STEPLIB environment variable is not specified, the exec() default behavior is the same as if STEPLIB=CURRENT were specified.

If the program to be invoked is a set-user-ID or set-group-ID file and the user-ID or group-ID of the file is different from that of the current process image, the data sets to be built into the STEPLIB environment for the new process image must be found in the system sanction list for set-user-id and set-group-id programs. Only those data sets that are found in the sanction list are built into the STEPLIB environment for the new process image. For detailed information regarding the sanction list, and for information on STEPLIB performance considerations, see *OS/390 UNIX System Services Planning*.

#### **\_BPX\_JOBNAME**

Used to change the jobname of the new process image. The jobname change is allowed only if the invoker has appropriate privileges and is running in an address space created by fork. If these conditions are not met, the environment variable is ignored. Accepted values are strings of 1–8 alphanumeric characters. Incorrect specifications are ignored.

**\_BPX\_ACCT\_DATA**

Used to change the account data of the new process image. Rules for specifying account data:

- Up to 142 actual account data characters are allowed, including any commas
- Sub-parameters must be separated by commas.
- There is no restriction on the character set.
- If the account data is greater than 142 characters, the data is ignored.

**Special Behavior for XPG4**

If this file is not a valid executable object, the `execlp()` and `execvp()` functions invoke **/bin/sh** with the invoker's pathname and the rest of the input arguments. It is similar to invoking:

```
execl("/bin/sh",
      "sh",
      "--",
      fully_expanded_pathname,
      arg1, arg2, ..., argn,
      NULL
    );
```

where `arg1`, `arg2`, ..., `argn` are the caller's arguments to `execlp()` or `execvp()`, and `fully_expanded_pathname` is the pathname of the shell script found by searching the directories in the current `PATH`.

*arg*, ..., `NULL` is a series of pointers to null-terminated character strings specifying arguments for the process being invoked. If the new process is a `main()`, these strings are stored in an array, and a pointer to the array is passed in the *argv* parameter. The first argument is required, and it should point to a string containing the name of the file that is associated with the process that `exec` is starting. A `NULL` pointer must follow the last argument string pointer.

*argv[ ]* is a pointer to an array of pointers to null-terminated character strings. There must be a `NULL` pointer after the last character string to mark the end of the array. These strings are used as arguments for the process being invoked. *argv[0]* should point to a string containing the name of a file associated with the process being started by `exec`. *envp[ ]* is a pointer to an array of pointers to null-terminated character strings. There must be a `NULL` pointer after the last character string to mark the end of the array. The strings of *envp* provide the environment variables for the new process.

All the forms of `exec` functions provide a way to locate the file containing the new process you want to run and a collection of arguments that should be passed to the new process. Each form of `exec` has its own method for specifying this information.

Some `exec` calls explicitly pass an environment using an *envp* argument. In versions where an environment is not passed explicitly—`execl()`, `execlp()`, `execv()`, and `execvp()`—the system uses the entire environment of the caller. The caller's environment is assumed to be the *environment variables* that the *external variable* `**environ` points to.

The variable `ARG_MAX`, obtained from OS/390 UNIX services by an invocation of `sysconf(_SC_ARG_MAX)`, specifies the maximum number of bytes that can be used

for arguments and environment variables passed to the process being invoked. The number of bytes includes the null terminator on each string.

A process started by an `exec` function has all of the open file descriptors that were present in the caller, except for those files opened with the close-on-exec flag `FD_CLOEXEC`. See “`fcntl()` — Control Open File Descriptors” on page 350 for more information about this flag. In file descriptors that remain open, all attributes remain unchanged (including file locks).

Directory streams that are open in the calling process image are closed in the new process image.

The state of conversion descriptors and message catalogue descriptors is undefined.

Signals set to be ignored in the caller, `SIG_IGN`, are set to be ignored in the new process image. Be careful to take care of signals that are being ignored. Although `sigaction()` specifying a handler is not passed by, `SIG_IGN` is. Blocking of signals is also passed by. All other signals are set to the default action, `SIG_DFL`, in the new process image, no matter how the caller handled such signals.

The real user ID (UID), real group ID (GID), and supplementary group IDs of the new process are the same as those of the caller. If the set-user-ID mode bit of the program file is on, the effective user ID of the new process is set to the file's owner. Similarly, if the set-group-ID mode bit of the program file is on, the effective group ID of the new process is set to the file's group. The effective user ID of the new process image is saved as the saved set-user-ID, and the effective group ID of the new process image is saved as the saved set-group-ID.

Any shared memory segments attached to the calling process image will not be attached to the new process image, see “`shmat()` — Shared Memory Attach Operation” on page 1285. Any shared memory segments attached to the calling process image will be detached (i.e., the value of `shm_nattch` decremented by one). If this is the last thread attached to the shared memory segment and a `shmctl()` `RMD` has been issued, the segment will be removed from the system.

### Special Behavior for XPG4.2

Interval timers are preserved across an `exec`.

The new process also inherits the following from the caller:

- Controlling terminal (**XPG4.2**)
- Nice value (see “`nice()` — Change Priority of a Process” on page 864) (**XPG4**)
- `semadj` values (see “`semop()` — Semaphore Operations” on page 1175) (**XPG4**)
- Process ID
- Parent process ID
- Process group ID
- Resource limits (see “`setrlimit()` — Control Maximum Resource Consumption” on page 1264 and “`ulimit()` — Get/Set Process File Size Limits” on page 1645) (**XPG4.2**)
- Session membership
- Time left until an alarm clock signal
- Working directory

- Root directory
- File mode creation mask
- File size limit (see “ulimit() — Get/Set Process File Size Limits” on page 1645) (**XPG4**)
- Process signal mask
- Pending signals
- `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime`. See “times() — Get Process and Child Process Times” on page 1572 for more about these qualities.

A successful `exec` function automatically opens the specified program file, and updates the access time `st_atime` for that file. The program file is closed automatically after the program has been read from the file. The precise time of this close operation is undefined.

### Special Behavior for OS/390 UNIX Services

#### Notes:

1. A prior loaded copy of an HFS program in the same address space is reused under the same circumstances that apply to the reuse of a prior loaded MVS unauthorized program from an unauthorized library by the MVS XCTL service with the following exceptions:
  - If the calling process is in Ptrace debug mode, a prior loaded copy is not reused.
  - If the calling process is not in Ptrace debug mode, but the only prior loaded usable copy found of the HFS program is in storage modifiable by the caller, the prior copy is not reused.
2. If the specified file name represents an external link or a sticky bit file, the program is loaded from the caller's MVS load library search order. For an external link, the external name is only used if the name is eight characters or less, otherwise the caller receives an error from the `loadhfs` service. For a sticky bit program, the file name is used if it is eight characters or less. Otherwise, the program is loaded from the HFS.
3. If the calling task is in a WLM enclave, the resulting task in the new process image is joined to the same WLM enclave. This allows WLM to manage the old and new process images as one ‘business unit of work’ entity for system accounting and management purposes.

### Returned Value

If successful, an `exec` function never returns control because the calling process is overwritten with the new process. If unsuccessful, an `exec` function returns `-1` and sets `errno` to one of the following:

- |               |  |
|---------------|--|
| <b>E2BIG</b>  | The combined argument list and environment list of the new process has more bytes than the system-defined length. See “ <code>sysconf()</code> — Determine System Configuration Options” on page 1483 for information about the system-defined length. |
| <b>EACCES</b> | The process did not have appropriate permissions to run the specified file, for one of these reasons: <ul style="list-style-type: none"> <li>• The process did not have permission to search a directory named in your <i>path</i>.</li> </ul>         |

- The process did not have execute permission for the file to be run.
  - The system cannot run files of this type.
- EINVAL** The new process image file has the appropriate permission and has a recognized format, but the system does not support execution of a file with this format.
- ELOOP** A loop exists in symbolic links. This error is issued if the number of symbolic links detected in the resolution of the *path* or *file* argument is greater than POSIX\_SYMLINK\_MAX (a value defined in the limits.h header file)
- ENAMETOOLONG** All or part of the file name is too long. This can happen if:
- A *path* or *file* argument exceeds the value of PATH\_MAX, or an element of your *path* exceeds PATH\_MAX.
  - Any *pathname* component is greater than NAME\_MAX, and \_POSIX\_NO\_TRUNC is in effect.
  - The length of a path name string substituted for a symbolic link in the *path* argument exceeds PATH\_MAX.
- The PATH\_MAX and NAME\_MAX values are determined with pathconf().
- ENOENT** One or more *pathname* components in *path* or *file* does not exist. This error is also issued if *path* or *file* is a null string.
- ENOEXEC** The new process image file has the appropriate access permission but has an unrecognized format. This errno can be returned from any one of the exec family of functions, except for execlp() and execvp().

**Note:**

Reason codes further qualify the errno. For most of the reason codes, see *OS/390 UNIX System Services Messages and Codes*.

For ENOEXEC, the reason codes are:

Reason Code	Explanation
X'xxxx0C27'	The target HFS file is not in the correct format to be an executable file.
X'xxxx0C31'	The target HFS file is built at a level that is higher than that supported by the running system.
<b>ENOMEM</b>	The new process requires more memory than is permitted by the operating system.
<b>ENOTDIR</b>	A directory component of <i>path</i> or <i>file</i> is not really a directory.
<b>EFAULT</b>	A bad address was received as an argument of the call, or the user exit program checked.
	Consult Reason Code to determine the exact reason the error occurred. The following reason code can accompany the return code: JRExecParmErr and JRExitRtnError.

**EMVSSAF2ERR**

The executable file is a set-user-ID or set-group-ID file, and the file owner's UID or GID is not defined to RACF.

**Example**  
**CBC3BE03**

```
/* CBC3BE03
   This example runs a program, using the execl() function.
*/
#define _POSIX_SOURCE
#include <stdio.h>
#include <wait.h>
#include <sys/types.h>
#include <unistd.h>

main() {
    pid_t pid;
    int status;

    if ((pid = fork()) == 0) {
        execl("/bin/false", NULL);
        perror("The execl() call must have failed");
        exit(255);
    }
    else {
        wait(&status);
        if (WIFEXITED(status))
            printf("child exited with status of %d\n", WEXITSTATUS(status));
        else
            puts("child did not exit successfully\n");
    }
}
```

**Output**

child exited with status of 1

**Related Information**

- “limits.h” on page 32
- “signal.h” on page 41
- “unistd.h” on page 53
- “alarm() — Set an Alarm” on page 102
- “chmod() — Change the Mode of a File or Directory” on page 174
- “\_exit() — End a Process and Bypass the Cleanup” on page 332
- “fcntl() — Control Open File Descriptors” on page 350
- “fork() — Create a New Process” on page 422
- “getrlimit() — Control Maximum Resource Consumption” on page 591
- “nice() — Change Priority of a Process” on page 864
- “putenv() — Change or Add an Environment Variable” on page 1054
- “semop() — Semaphore Operations” on page 1175
- “shmat() — Shared Memory Attach Operation” on page 1285
- “setuid() — Set the Effective User ID” on page 1278
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sigpending() — Examine Pending Signals” on page 1337
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “stat() — Get File Information” on page 1404
- “times() — Get Process and Child Process Times” on page 1572
- “system() — Execute a Command” on page 1488

- “ulimit() — Get/Set Process File Size Limits” on page 1645
- “umask() — Set and Retrieve File Creation Mask” on page 1647

## exit() — End Program

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
void exit(int status);
```

### General Description

The `exit()` function:

1. Calls all functions registered with the `atexit()` function, in last-in-first-out (LIFO) order.
2. Flushes all buffers, and closes all open files.
3. All files opened with `tmpfile()` are deleted.
4. Returns control to the host environment from the program.

Process termination in `_exit()` is equivalent to program termination in `exit()`.

The argument *status* can have a value from 0 to 255 inclusive or be one of the macros `EXIT_SUCCESS` or `EXIT_FAILURE`. The value of `EXIT_SUCCESS` is defined in `stdlib.h` as 0; the value of `EXIT_FAILURE` is 8.

This function is also available to C applications in a stand-alone Systems Programming Environment.

In a POSIX C program, `exit()` returns control to the kernel with the value of *status*. The kernel then performs normal process termination. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information about using POSIX support.

POSIX-level thread cleanup routines are *not* executed. These includes cleanup routines created with `pthread_cleanup_push()` and destructor routines created with `pthread_key_create()`.

### Special Behavior for C++

If `exit()` is called in a OS/390 C++ program, the program terminates without leaving the current block, and therefore destructors are not called for local (automatic) variables. Destructors for initialized static objects will be called.



## Returned Value

No returned value. The `exit()` function returns control to its host environment, with the returned value status. For example, if program A invokes program B via a call to the `system()` function, and program B calls the `exit()` function, then program B returns to its host environment, which is program A.

## Example

```
/* This example flushes all buffers, closes any open files, and ends the
   program if it cannot open the file myfile.
*/
#include <stdio.h>
#include <stdlib.h>

FILE *stream;

int main(void)
{
    :
    if ((stream = fopen("myfile.dat", "r")) == NULL)
    {
        printf("Could not open data file\n");
        exit(EXIT_FAILURE);
    }
}
```

## Related Information

- “System Programming Facilities” in *OS/390 C/C++ Programming Guide*
- “Using Runtime User Exits” in *OS/390 C/C++ Programming Guide*
- “stdlib.h” on page 45
- “abort() — Stop a Program” on page 71
- “atexit() — Register Program Termination Function” on page 116
- “\_exit() — End a Process and Bypass the Cleanup” on page 332
- “signal() — Handle Interrupts” on page 1330
- “wait() — Wait for a Child Process to End” on page 1687
- “waitpid() — Wait for a Specific Child Process to End” on page 1692

## `_exit()` — End a Process and Bypass the Cleanup

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
void _exit(int status);
```

### General Description

Ends the current process and makes an exit status value for the process available to the system.

The argument *status* specifies a return status for the process that is ending. Ending the process has the following results:

- `_exit()` closes all open file descriptors and directory streams in the caller.
- If the caller's parent is currently suspended because of `wait()` or `waitpid()`, the low-order 8 bits of *status* become available to the parent. For a discussion on accessing those 8 bits, refer to “`waitpid()` — Wait for a Specific Child Process to End” on page 1692.
- If the caller's parent is not currently suspended because of `wait()` or `waitpid()`, `_exit()` saves the *status* value so that it can be returned to the parent if the parent calls `wait()` or `waitpid()`.
- A `SIGCHILD` signal is sent to the parent process.
- If the process calling `_exit()` is a controlling process, the `SIGHUP` signal is sent to each process in the foreground process group of the controlling terminal belonging to the caller.
- If the process calling `_exit()` is a controlling process, `_exit()` disassociates the associated controlling terminal from the session. A new controlling process can then acquire the terminal.
- Exiting from a process does not end its child processes directly. The `SIGHUP` signal may end children in some cases. Children that survive when a process ends are assigned a new parent process ID. The new parent process ID is always 1, indicating the root ancestor of all processes.
- If a process ends and orphans a process group and if a member of that group is stopped, each member of the group is sent a `SIGHUP` signal, followed by a `SIGCONT` signal.
- All threads are ended, and their resources cleaned up. (*Threads* are MVS tasks that call an OS/390 UNIX callable service.) POSIX-level thread cleanup routines are *not* executed. These include cleanup routines created with `pthread_cleanup_push()` and destructor routines created with `pthread_key_create()`.

These results occur whenever a process ends. `_exit()` does not cause C runtime library cleanup to be performed; therefore, stream buffers are not necessarily flushed.

**Note:** If `_exit()` is issued from a TSO/E address space, it ends the calling task and all its subtasks.

### Special Behavior for C++

If `_exit()` is called in a C++/MVS program, the program terminates without leaving the current block, and destructors are not called for local (automatic) variables. In addition, unlike `exit()`, destructors for global (static) variables are not called.

### Returned Value

`_exit()` is always successful and does not return a value. No value is stored in `errno` for this function.

### Example CBC3BE05

```
/* CBC3BE05
   This example ends a process.
*/
#define _POSIX_SOURCE
#include <unistd.h>
#include <stdio.h>

main() {
    puts("Remember that stream buffers are not automatically");
    puts("flushed before _exit()!");
    fflush(NULL);
    _exit(0);
}
```

### Output

```
Remember that stream buffers are not automatically
flushed before _exit()!
```

### Related Information

- “`stdlib.h`” on page 45
- “`unistd.h`” on page 53
- “`abort()` — Stop a Program” on page 71
- “`atexit()` — Register Program Termination Function” on page 116
- “`close()` — Close a File” on page 191
- “`exit()` — End Program” on page 330
- “`fork()` — Create a New Process” on page 422
- “`sigaction()` — Examine or Change a Signal Action” on page 1295
- “`signal()` — Handle Interrupts” on page 1330
- “`wait()` — Wait for a Child Process to End” on page 1687

## exp() — Calculate Exponential Function

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double exp(double x);
```

### General Description

Calculates the exponent of  $x$ , defined as  $e^{**}x$ , where  $e$  equals 2.17128128....).

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

Returns the calculated value if successful. If an overflow occurs, it returns HUGE\_VAL. If an underflow occurs, the function returns 0. Both overflow and underflow set errno to ERANGE.

### Example

#### CBC3BE06

```
/* CBC3BE06
   This example calculates y as the exponential function of x.
*/
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, y;

    x = 5.0;
    y = exp(x);

    printf("exp( %f ) = %f\n", x, y);
}
```

### Output

```
exp( 5.000000 ) = 148.413159
```

### Related Information

- “math.h” on page 35
- “log() — Calculate Natural Logarithm” on page 762
- “log10() — Calculate Base 10 Logarithm” on page 767
- “pow() — Raise to Power” on page 916

# expm1() — Exponential Minus One

## Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

## Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>
```

```
double expm1(double x);
```

## General Description

The expm1() function calculates the function  $e^x - 1.0$ .

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

## Returned Value

If it succeeds, expm1() returns the above function calculated on x. expm1() may fail as follows:

- If x is negative and exceeds an internally defined large value, expm1() will return -1.0.
- If the value of the function overflows, expm1() returns HUGE\_VAL and sets errno to ERANGE.

## Related Information

- “math.h” on page 35
- “exp() — Calculate Exponential Function” on page 334
- “ilogb() — Integer Unbiased Exponent” on page 659
- “log1p() — Natural Log of x + 1” on page 766

## extlink\_np() — Create an External Symbolic Link

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 4.3

### Format

```
#include <unistd.h>
```

```
int extlink_np(const char *ename, const char *elink);
```

### General Description

Creates the external symbolic link file named by *elink* with the object specified by *ename*. The *ename* is not resolved, and refers to an object outside the HFS (hierarchical file system). The variable *elink* is the name of the external symbolic link file created, and *ename* is the name of the object contained within that file.

### Returned Value

If successful, `extlink_np()` returns 0. If unsuccessful, it returns `-1` and does not affect any file it names. `extlink_np()` sets `errno` to one of the following:

- EACCES** A component of the *elink* path prefix denies search permission.
- EINVAL** *elink* has a slash as its last component, which indicates that the preceding component will be a directory. An external link cannot be a directory.
- ENAMETOOLONG** *pathname* is longer than `PATH_MAX` characters, or some component of *pathname* is longer than `NAME_MAX` characters while `_POSIX_NO_TRUNC` is in effect. For symbolic links, the length of the *pathname* string substituted for a symbolic link exceeds `PATH_MAX`. The `PATH_MAX` and `NAME_MAX` values can be determined with `pathconf()`.
- ENOTDIR** A component of the path prefix of *elink* is not a directory.
- ELOOP** A loop exists in symbolic links. This error is issued if the number of symbolic links encountered during resolution of the *elink* argument is greater than `POSIX_SYMLoop`.
- EEXIST** The file named by *elink* already exists.
- ENOSPC** The new external link cannot be created because there is no space left on the file system to contain it.
- EROFS** The file named by *elink* cannot be created on a read-only file system.

### Example

#### CBC3BE07

```
/* CBC3BE07
   This example creates an external symbolic link.
*/
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/stat.h>
```

```

#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>

main( argc, argv )
    int  argc ;
    char *argv ;
{
    int  i_rc ;
    int  i_fd ;
    char ac_mvds[] = "SYS1.LINKLIB" ;
    char ac_mvdsextsymlnk[] = "sys1.linklib.extsymlnk" ;

    i_rc = unlink( ac_mvdsextsymlnk ) ;

    if (( i_rc == -1 ) && ( errno == ENOENT )) {
    }
    else {
        perror( "unlink() error" ) ;
        return( -1 ) ;
    }

    printf( "Before extlink_np() call ...\n" ) ;
    system( "ls -il sys1.*" ) ;

    i_rc = extlink_np( ac_mvds, ac_mvdsextsymlnk ) ;

    if ( i_rc == -1 ) {
        perror( "extlink_np() error" ) ;
        return( -1 ) ;
    }

    printf( "After extlink_np() call ...\n" ) ;
    system( "ls -il sys1.*" ) ;

    i_rc = unlink( ac_mvdsextsymlnk ) ;
}

```

## Related Information

- “unistd.h” on page 53
- “link() — Create a Link to a File” on page 749
- “lstat() — Get Status of File or Symbolic Link” on page 778
- “readlink() — Read the Value of a Symbolic Link” on page 1091
- “symlink() — Create a Symbolic Link to a Path Name” on page 1479
- “unlink() — Remove a Directory Entry” on page 1660

## fabs() — Calculate Floating-Point Absolute Value

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <math.h>
double fabs(double x);
```

### General Description

The built-in fabs() function calculates the absolute value of a floating-point argument.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

Returns the absolute value of the float input.

### Example

```
/* This example calculates y as the absolute value of x. */
#include <math.h>

int main(void)
{
    double x, y;

    x = -5.6798;
    y = fabs(x);

    printf("fabs( %f ) = %f\n", x, y);
}
```

### Output

```
fabs( -5.679800 ) = 5.679800
```

### Related Information

- “math.h” on page 35
- “abs() — Calculate Integer Absolute Value” on page 73
- “labs() — Calculate Long Absolute Value” on page 733



## fattach() — Attach a STREAMS-based File Descriptor to a File in the File System Name Space

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stropts.h>
```

```
int fattach(int fildev, const char *path);
```

### General Description

The `fattach()` function attaches a STREAMS-based file descriptor to a file, effectively associating a pathname with *fildev*. The *fildev* argument must be a valid open file descriptor associated with a STREAMS file. The *path* argument points to a pathname of an existing file. The process must have appropriate privileges, or must be the owner of the file named by *path* and have write permission. A successful call to `fattach()` causes all pathnames that name the file named by *path* to name the STREAMS file associated with *fildev*, until the STREAMS file is detached from the file. A STREAMS file can be attached to more than one file and can have several pathnames associated with it.

The attributes of the named STREAMS file are initialized as follows: the permissions, user ID, group ID, and times are set to those of the file named by *path*, the number of links is set to 1, and the size and device identifier are set to those of the STREAMS file associated with *fildev*. If any attributes of the named STREAMS file are subsequently changed (for example, by `chmod()`), neither the attributes of the underlying file nor the attributes of the STREAMS file to which *fildev* refers are affected.

File descriptors referring to the underlying file, opened prior to an `fattach()` call, continue to refer to the underlying file.

### Returned Value

Upon successful completion, `fattach()` returns 0. Otherwise, -1 is returned and `errno` is set to indicate the error.

**Note:** OS/390 UNIX services do not supply any STREAMS devices or pseudodevices. It is impossible for `fattach()` to attach a STREAMS-based file descriptor to a file. It will always return -1 with `errno` set to indicate the failure. See “`open()` — Open a File” on page 872 for more information.

The following are the possible values of `errno`:

- EACCES** Search permission is denied for a component of the path prefix, or the process is the owner of *path* but does not have write permissions on the file named by *path*.
- EBADF** The *fildev* argument is not a valid open file descriptor.
- ENOENT** A component of *path* does not name an existing file or *path* is an empty string.

ENOTDIR	A component of the path prefix is not a directory.
EPERM	The effective user ID of the process is not the owner of the file named by <i>path</i> and the process does not have appropriate privilege.
EBUSY	The file named by <i>path</i> is currently a mount point or has a STREAMS file attached to it.
ENAMETOOLONG	The size of <i>path</i> exceeds PATH_MAX, or a component of <i>path</i> is longer than NAME_MAX, or pathname resolution of a symbolic link produced an intermediate result whose length exceeds PATH_MAX.
ELOOP	Too many symbolic links were encountered in resolving <i>path</i> .
EINVAL	The <i>fildev</i> argument does not refer to a STREAMS file.

**Related Information**

- “stropts.h” on page 46
- “fdetach() — Detach a Name from a STREAMS-based File Descriptor” on page 361
- “isastream() — Test a File Descriptor” on page 702

## fchaudit() — Change Audit Flags for a File by Descriptor

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 4.3

### Format

```
#include <sys/stat.h>
```

```
int fchaudit(int fildes, unsigned int flags, unsigned int option);
```

### General Description

Changes the audit flags of a file. The parameter *fildes* is the file descriptor for the open file whose audit flags are to be changed. *flags* specifies what the audit flags should be changed to:

AUDTREADFAIL

Audit failing read requests.

AUDTREADSUCC

Audit successful read requests.

AUDTWRITEFAIL

Audit failing write requests.

AUDTWritesucc

Audit successful write requests.

AUDTEXECFAIL

Audit failing execute or search requests.

AUDTEXECsucc

Audit successful execute or search requests. The bitwise inclusive OR of any or all of these can be used to set more than one type of auditing.

The parameter *option* specifies whether the user audit flags or the security auditor audit flags should be changed:

AUDT\_USER (0)

User audit flags are changed. The user must be the file owner or have appropriate authority to change the user audit flags for a file.

AUDT\_AUDITOR (1)

Security-auditor audit flags are changed. The user must have security auditor authority to change the security auditor audit flags for a file.

### Returned Value

If successful, fchaudit() returns 0. If unsuccessful, it returns -1 and sets errno to one of the following:

EBADF *fildes* is not a valid open file descriptor.

EINVAL *option* does not contain a 0 or 1.

- EPERM**      The effective user ID (UID) of the calling process does not match the owner of the file, and the calling process does not have appropriate privileges.
- EROFS**      *files* is associated with a file that is on a read-only file system.

### Example

#### CBC3BF02

```

/* CBC3BF02
   The following program changes the audit flags of a file.
*/
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

main() {
    int fd;
    char fn[]="fchaudit.file";

    if ((fd = creat(fn, S_IRUSR|S_IWUSR)) < 0)
        perror("creat() error");
    else {
        if (fchaudit(fd, AUDTREADSUCC, AUDT_USER) != 0)
            perror("fchaudit() error");
        close(fd);
        unlink(fn);
    }
}

```

### Related Information

- “sys/stat.h” on page 48
- “access() — Determine Whether a File Can be Accessed” on page 82
- “chaudit() — Change Audit Flags for a File by Path” on page 165
- “fchmod() — Change the Mode of a File or Directory by Descriptor” on page 344
- “fchown() — Change the Owner or Group by File Descriptor” on page 346
- “fstat() — Get Status Information about a File” on page 479

## fchdir() — Change Working Directory

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
int fchdir(int fildes);
```

### General Description

The `fchdir()` function has the same effect as `chdir()` except that the directory that is to be new current working directory is specified by the file descriptor *fildes*.

### Returned Value

If successful, `fchdir()` changes the working directory and returns 0. If unsuccessful, `fchdir()` does not change the working directory, returns -1, and sets `errno` to one of the following:

`EACCES` Search permission is denied for the directory referenced by *fildes*.

`EBADF` The *fildes* arguments is not an open file descriptor.

`ENOTDIR` The open file descriptor *fildes* does not refer to a directory.

### Related Information

- “`chdir()` — Change the Working Directory” on page 167
- “`chroot()` — Change Root Directory” on page 182

## fchmod() — Change the Mode of a File or Directory by Descriptor

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX1_SOURCE 2
#include <sys/stat.h>
```

```
int fchmod(int fildev, mode_t mode);
```

### General Description

Sets the S\_ISUID, S\_ISGID, and file permission bits of the open file identified by *fildev*, its file descriptor.

The *mode* argument is created with one of the symbols defined in the sys/stat.h header file. For more information on these symbols, refer to “chmod() — Change the Mode of a File or Directory” on page 174.

### Returned Value

If successful, fchmod() marks for update the st\_ctime field of the file and returns 0. If unsuccessful, it returns -1 and sets errno to one of the following:

EBADF        *fildev* is not a valid open file descriptor.

EPERM        The effective user ID (UID) does not match the owner of the file, and the calling process does not have appropriate privileges.

EROFS        The file resides on a read-only file system.

### Example

#### CBC3BF03

```
/* CBC3BF03
   This example changes a file permission.
*/
#define _POSIX1_SOURCE 2
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

main() {
    char fn[]="temp.file";
    int fd;
    struct stat info;

    if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        stat(fn, &info);
        printf("original permissions were: %08x\n", info.st_mode);
        if (fchmod(fd, S_IRWXU|S_IRWXG) != 0)
            perror("fchmod() error");
        else {
```

```
        stat(fn, &info);
        printf("after fchmod(), permissions are: %08x\n", info.st_mode);
    }
    close(fd);
    unlink(fn);
}
```

### Output

original permissions were: 03000080  
after fchmod(), permissions are: 030001f8

### Related Information

- “sys/stat.h” on page 48
- “chmod() — Change the Mode of a File or Directory” on page 174
- “chown() — Change the Owner or Group of a File or Directory” on page 177
- “fchown() — Change the Owner or Group by File Descriptor” on page 346
- “mkdir() — Make a Directory” on page 817
- “mkfifo() — Make a FIFO Special File” on page 820
- “open() — Open a File” on page 872
- “stat() — Get File Information” on page 1404

## fchown() — Change the Owner or Group by File Descriptor

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX1_SOURCE 2
#include <unistd.h>
```

```
int fchown(int fildes, uid_t owner, gid_t group);
```

### General Description

Changes the owner or group (or both) of a file. *fildes* is the file descriptor for the file. *owner* is the user ID (UID) of the new owner of the file. *group* is the group ID of the new group for the file.

If `_POSIX_CHOWN_RESTRICTED` is defined in the `unistd.h` header file, a process can change the group of a file only if one of the following conditions is true:

1. The process has appropriate privileges.
- Or
2. All of the following are true:
  - a. The effective user ID of the process is equal to the user ID of the file owner.
  - b. The *owner* argument is equal to the user ID of the file owner or `(uid_t)-1`,
  - c. The *group* argument is either the effective group ID or a supplementary group ID of the calling process.

If *fildes* points to a regular file and one or more of the `S_IXUSR`, `S_IXGRP`, or `S_IXOTH` bits of the file mode are set when `fchown()` returns successfully, it clears the set-user-ID (`S_ISUID`) and set-group-ID (`S_ISGID`) bits of the file mode.

If the file referred to by *fildes* is not a regular file and one or more of the `S_IXUSR`, `S_IXGRP`, or `S_IXOTH` bits of the file mode are set, the set-user-ID (`S_ISUID`) and set-group-ID (`S_ISGID`) bits of the file are cleared.

When `fchown()` completes successfully, it marks the `st_ctime` field of the file to be updated.

### Returned Value

If successful, `fchown()` updates the change time for the file and returns 0. If unsuccessful, it returns `-1` and sets `errno` to one of the following:

<code>EBADF</code>	<i>fildes</i> is not a valid open file descriptor.
<code>EPERM</code>	Either the effective user ID does not match the owner of the file, or the calling process does not have appropriate privileges, and <code>POSIX_CHOWN_RESTRICTED</code> indicates that such privilege is required.



EROFS      The file resides on a read-only system.

### Example CBC3BF04

```
/* CBC3BF04
   This example changes the owner ID and group ID.
*/
#define _POSIX1_SOURCE 2
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <stdio.h>

main() {
    char fn[]="temp.file";
    FILE *stream;
    int fd;
    struct stat info;

    if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        stat(fn, &info);
        printf("original owner was %d and group was %d\n", info.st_uid,
               info.st_gid);
        if (fchown(fd, 25, 0) != 0)
            perror("fchown() error");
        else {
            stat(fn, &info);
            printf("after fchown(), owner is %d and group is %d\n",
                   info.st_uid, info.st_gid);
        }
        close(fd);
        unlink(fn);
    }
}
```

### Output

```
original owner was 0 and group was 500
after fchown(), owner is 25 and group is 0
```

### Related Information

- “unistd.h” on page 53
- “chown() — Change the Owner or Group of a File or Directory” on page 177
- “chmod() — Change the Mode of a File or Directory” on page 174
- “fchmod() — Change the Mode of a File or Directory by Descriptor” on page 344
- “mkdir() — Make a Directory” on page 817
- “mkfifo() — Make a FIFO Special File” on page 820
- “open() — Open a File” on page 872
- “stat() — Get File Information” on page 1404

## fclose() — Close File

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
int fclose(FILE *stream);
```

### General Description

Flushes a stream, and then closes the file associated with that stream. Afterwards, the function releases any buffers associated with the stream. To *flush* means that unwritten buffered data is written to the file, and unread buffered data is discarded.

**Note:** The storage pointed to by the FILE pointer is freed by the fclose() function. An attempt to use the FILE pointer to a closed file is invalid. This restriction is true even when fclose() fails.

A pointer to a closed file *cannot* be used as an input value to the freopen() function.

### Returned Value

Returns 0 if it successfully closes the stream. If a failure occurs in flushing buffers or in outputting data, EOF will be returned. An attempt will still be made to close the file.

### Special Behavior for XPG4

The following are possible errno values returned by fclose():

EAGAIN	The O_NONBLOCK flag is set and output cannot be written immediately.
EBADF	The underlying file descriptor is not valid.
EFBIG	Writing to the output file would exceed the maximum file size or the process' file size supported by the implementation.
EINTR	The fclose() function was interrupted by a signal before it had written any output.
EIO	The process is in a background process group and is attempting to write to its controlling terminal, but TOSTOP (defined in the termio.h include file) is set, the process is neither ignoring nor blocking SIGTTOU signals, and the process group of the process is orphaned.
ENOSPC	There is no free space left on the output device
EPIPE	fclose() is trying to write to a pipe or FIFO that is not open for reading by any process. This error also generates a SIGPIPE signal.
ENXIO	A request was made of a non-existent device, or the request was outside the device.

**Example**

```
/* This example opens a file myfile.dat for reading as a stream and then
   closes the file.
*/
#include <stdio.h>

int main(void)
{
    FILE *stream;

    stream = fopen("myfile.dat", "r");
    :
    if (fclose(stream)) /* Close the stream. */
        printf("fclose error\n");
}
```

**Related Information**

- The “Closing Files” sections and the “Opening Files” sections, in the *OS/390 C/C++ Programming Guide*
- “stdio.h” on page 43
- “fopen() — Open a File” on page 417
- “freopen() — Redirect an Open File” on page 460

## fcntl() — Control Open File Descriptors

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <fcntl.h>
```

```
int fcntl(int fildes, int action, ...);
```

### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
```

```
int fcntl(int socket, int cmd, ...);
```

### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
```

```
int fcntl(int socket, int cmd, ...);
```

### General Description

Performs various actions on open file descriptors.

The argument *fildes* is a file descriptor for the file you want to manipulate. *action* is a symbol indicating the action you want to perform on *fildes*. These symbols are defined in the `fcntl.h` header file. If needed, “...” indicates a third argument. The type of the third argument depends on *action*, and some actions do not need an additional argument.

### Behavior for Sockets

The operating characteristics of sockets can be controlled with the `fcntl()` call. The operations to be controlled are determined by *cmd*. The *arg* parameter is a variable with a meaning that depends on the value of the *cmd* parameter.

Parameter	Description
<i>socket</i>	The socket descriptor.
<i>cmd</i>	The command to perform.
<i>arg</i>	The data associated with <i>cmd</i> .

The *action* argument can be one of the following symbols:

F_CLOSF	Closes a range of file descriptors. A third <code>int</code> argument must be specified to indicate the upper limit for the range of the file descriptors to be closed, while <i>fil</i> des specifies the lower limit. If <code>-1</code> is specified for the third argument, all file descriptors greater than or equal to the lower limit are closed.
F_DUPFD	Duplicates the file descriptor. A third <code>int</code> argument must be specified. <code>fcntl()</code> returns the lowest file descriptor greater than or equal to this third argument that is not already associated with an open file. This file descriptor refers to the same file as <i>fil</i> des and shares any locks. The flags <code>FD_CLOEXEC</code> and <code>FD_CLOFORK</code> are turned off in the new file descriptor, so that the file is kept open if an <code>exec</code> function is called.
F_DUPFD2	Duplicates the file descriptor. A third <code>int</code> argument must be specified to indicate which file descriptor to use as the duplicate. This file descriptor is closed if already open and then used as the new file descriptor. The new file descriptor refers to the same file as <i>fil</i> des and shares any locks. The flags <code>FD_CLOEXEC</code> and <code>FD_CLOFORK</code> are turned off in the new file descriptor, so that the file is kept open if an <code>exec</code> function is called.
F_GETFD	Obtains the file descriptor flags for <i>fil</i> des. <code>fcntl()</code> returns these flags as its result. For a list of supported file descriptor flags, see “File Flags” on page 352.
F_SETFD	Sets the file descriptor flags for <i>fil</i> des. You must specify a third <code>int</code> argument, giving the new file descriptor flag settings. <code>fcntl()</code> returns 0 if it successfully sets the flags.
F_GETFL	Obtains the file status flags and file access mode flags for <i>fil</i> des. <code>fcntl()</code> returns these flags as its result. For a list of supported file status and file access mode flags, see “File Flags” on page 352  <b>Behavior for Sockets:</b> This command gets the status flags of socket descriptor <i>socket</i> . With the <code>_OPEN_SYS</code> feature test macro you can query the <code>FNDELAY</code> flag. With the <code>_XOPEN_SOURCE_EXTENDED 1</code> feature test macro you can query the <code>O_NDELAY</code> flag. The <code>FNDELAY</code> and <code>O_NDELAY</code> flags mark <i>socket</i> as being in non-blocking mode. If data is not present on calls that can block, such as <code>read()</code> , <code>readv()</code> , and <code>recv()</code> , the call returns with <code>-1</code> , and the error code is set to <code>EWOULDBLOCK</code> .
F_SETFL	Sets the file status flags for <i>fil</i> des. You must specify a third <code>int</code> argument, giving the new file descriptor flag settings. <code>fcntl()</code> does not change the file access mode, and file access bits in the third argument are ignored. <code>fcntl()</code> returns 0 if it successfully sets the flags.  <b>Behavior for Sockets:</b> This command sets the status flags of socket descriptor <i>socket</i> . With the <code>_OPEN_SYS</code> feature test macro you can set the <code>FNDELAY</code> flag. With the <code>_XOPEN_SOURCE_EXTENDED 1</code> feature test macro you can set the <code>O_NDELAY</code> flag.
F_GETLK	Obtains locking information for a file. See “File Locking” on page 353
F_SETLK	Sets or clears a file segment lock. See “File Locking” on page 353

- F\_SETLKW** Sets or clears a file segment lock; but if a shared or exclusive lock is blocked by other locks, `fcntl()` waits until the request can be satisfied. See “File Locking” on page 353
- F\_GETOWN** **Behavior for Sockets:** Obtains the PID for the filedes and returns this value. The value returned will be either the process ID or the process group ID that is associated with the socket. If it is a positive integer, it specifies a process ID. If it is a negative integer (other than -1), it specifies a process group ID.
- F\_SETOWN** **Behavior for Sockets:** Sets either the process ID or the process group ID that is to receive either the SIGIO or SIGURG signals for the socket associated with filedes. The SIGURG signal is generated as a result of receiving out-of-band data. Refer to `send()`, `sendto()`, `sendmsg()`, and `recv()`, `recvfrom()` and `recvmsg()` for more information on sending and receiving out-of-band data.
- You must specify a third `int` argument, giving the PID requested. This value can be either a positive integer, specifying a process ID, or a negative integer (other than -1), specifying a process group ID. The difference between specifying a process ID or a process group ID is that in the first case only a single process will receive the signal, while in the second case all processes in the process group will receive the signal.

## File Flags

There are several types of flags associated with each open file. Flags for a file are represented by symbols defined in the `fcntl.h` header file.

The following *file descriptor* flags can be associated with a file:

- FD\_CLOEXEC** If this flag is 1, the file descriptor is closed if the process executes one of the `exec` function calls. If it is 0, the file remains open.
- FD\_CLOFORK** If this flag is 1 when a fork occurs, the file descriptor will be closed for the child process. If it is 0, the file remains open for the child.

The following *file status* flags can be associated with a file:

- O\_APPEND** Append mode. If this flag is 1, every write operation on the file begins at the end of the file.
- O\_ASYNC** If this flag is 1, then asynchronous I/O will be used for the file.
- O\_NONBLOCK** No blocking. If this flag is 1, read and write operations on the file return with an error status if they cannot perform their I/O immediately. If this flag is 0, read and write operations on the file wait (or “block”) until the file is ready for I/O. For more details, see “`read()` — Read From a File or Socket” on page 1080 and “`write()` — Write Data on a File or Socket” on page 1780.
- O\_SYNC** Force synchronous update. This flag is supported on MVS 5.1 but ignored on MVS 4.3. If the flag is 1, every `write()` operation on the file is written to permanent storage. That is, the file system buffers are forced to permanent storage. (See “`fsync()` — Write Changes to Direct-Access Storage” on page 483.) If this flag is 0, update operations on the file will not be completed until the data has been

written to permanent storage. On return from a function that performs a synchronous update, the program is assured that all data for the file has been written to permanent storage.

The following *file access mode* flags can be associated with a file:

O\_RDONLY    The file is opened for reading only.  
 O\_RDWR      The file is opened for reading and writing.  
 O\_WRONLY    The file is opened for writing only.

Two masks can be used to extract flags:

O\_ACCMODE   Extracts file access mode flags.  
 O\_GETFL      Extracts file status flags and file access mode flags.

## File Locking

A process can use `fcntl()` to lock out other processes from a part of a file, so that the process can read or write to that part of the file without interference from others. File locking can ensure data integrity when several processes have a file accessed concurrently. File locking can only be performed on file descriptors that refer to regular files. Locking is not permitted on file descriptors that refer to directories, FIFOs, pipes, character special files, or any other type of files.

A structure that has the type `struct flock` (defined in the `fcntl.h` header file) controls locking operations. This structure has the following members:

`short l_type`

Indicates the type of lock, using one of the following symbols (defined in the `fcntl.h` header file):

F\_RDLCK    Indicates a *read lock*, also called a *shared lock*. The process can read the locked part of the file, and other processes cannot obtain write locks for that part of the file in the meantime. More than one process can have a read lock on the same part of a file simultaneously.

To establish a read lock, a process must have the file accessed for reading.

F\_WRLCK    Indicates a *write lock*, also called an *exclusive lock*. The process can write on the locked part of the file, and no other process can establish a read lock or write lock on that same part or on an overlapping part of the file. A process cannot put a write lock on part of a file if there is already a read lock on an overlapping part of the file. To establish a write lock, a process must have accessed the file for writing.

F\_UNLCK    Unlocks a lock that was set previously.

`short l_whence`

One of three symbols used to determine the part of the file that is affected by this lock. These symbols are defined in the `unistd.h` header file and are the same as symbols used by `lseek()`:

SEEK\_CUR    The current file offset in the file  
 SEEK\_END    The end of the file  
 SEEK\_SET    The start of the file.

off\_t l\_start

Gives the byte offset used to identify the part of the file that is affected by this lock. The part of the file affected by the lock begins at this offset from the location given by l\_whence. For example, if l\_whence is SEEK\_SET and l\_start is 10, the locked part of the file begins at an offset of 10 bytes from the beginning of the file.

off\_t l\_len Gives the size of the locked part of the file in bytes. This size should not be negative. If l\_len is 0, the locked part of the file begins at the position specified by l\_whence and l\_start, and extends to the end of the file. Together, l\_whence, l\_start, and l\_len are used to describe the part of the file that is affected by this lock.

pid\_t l\_pid Specifies the process ID of the process that holds the lock. This is an output field used only with F\_GETLK actions.

You can set locks by specifying F\_SETLK as the *action* argument for fcntl(). Such a function call requires a third argument pointing to a struct flock structure, as in this example:

```
struct flock lock_it;
lock_it.l_type = F_RDLCK;
lock_it.l_whence = SEEK_SET;
lock_it.l_start = 0;
lock_it.l_len = 100;
fcntl(filides, F_SETLK, &lock_it);
```

This example sets up an flock structure describing a read lock on the first 100 bytes of a file, and then calls fcntl() to establish the lock. You can unlock this lock by setting l\_type to F\_UNLCK and making the same call. If an F\_SETLK operation cannot set a lock, it returns immediately with an error saying that the lock cannot be set.

The F\_SETLKW operation is similar to F\_SETLK, except that it waits until the lock can be set. For example, if you want to establish an exclusive lock and some other process already has a lock established on an overlapping part of the file, fcntl() waits until the other process has removed its lock. If fcntl() is waiting in an F\_SETLKW operation when a signal is received, fcntl() is interrupted. After handling the signal, fcntl() returns -1 and sets errno to EINTR.

F\_SETLKW operations can encounter *deadlocks* when process A is waiting for process B to unlock a region, and B is waiting for A to unlock a different region. If the system detects that an F\_SETLKW might cause a deadlock, fcntl() fails with errno set to EDEADLK.

A process can determine locking information about a file by using F\_GETLK as the *action* argument for fcntl(). In this case, the call to fcntl() should specify a third argument pointing to an flock structure. The structure should describe the lock operation you want. When fcntl() returns, the structure indicated by the flock pointer is changed to show the first lock that would prevent the proposed lock operation from taking place. The returned structure shows the type of lock that is set, the part of the file that is locked, and the process ID of the process that holds the lock. In the returned structure:

- l\_whence is always SEEK\_SET.
- l\_start gives the offset of the locked portion from the beginning of the file.
- l\_len is the length of the locked portion.



If there are no locks that prevent the proposed lock operation, the returned structure has `F_UNLCK` in `l_type`, and is otherwise unchanged.

A process can have several locks on a file simultaneously but only one type of lock set on a given byte. Therefore, if a process puts a new lock on part of a file that it had locked previously, the process has only one lock on that part of the file: the type of the lock is the one specified in the most recent locking operation.

All of a process's locks on a file are removed when the process closes any file descriptor that refers to the locked file. Locks are not inherited by child processes created with `fork()`.

All locks are advisory only. Processes can use locks to inform each other that they want to protect parts of a file, but locks do not prevent I/O on the locked parts. If a process has appropriate permissions on a file, it can perform whatever I/O it chooses, regardless of what locks are set. Therefore, file locking is only a convention, and it works only when all processes respect the convention.

## Returned Value

If successful, the value returned will depend on the *action* that was specified. If unsuccessful, `fcntl()` returns `-1` and sets `errno` to one of the following:

EAGAIN	The process tried to set a lock with <code>F_SETLK</code> , but the lock is in conflict with a lock already set by some other process on an overlapping part of the file.
EBADF	<i>fildev</i> is not a valid open file descriptor; or the process tried to set a read lock on a file descriptor open for writing only; or the process tried to set a write lock on a file descriptor open for reading only; or the <i>socket</i> parameter is not a valid socket descriptor.  In an <code>F_DUPFD2</code> operation, the third argument is negative, or greater than or equal to <code>OPEN_MAX</code> , which is the highest file descriptor value allowed for the process.
EDEADLK	The system detected the potential for deadlock in a <code>F_SETLKW</code> operation.
EINTR	<code>fcntl()</code> was interrupted by a signal during a <code>F_SETLKW</code> operation.
EINVAL	In an <code>F_DUPFD</code> operation, the third argument is negative or greater than or equal to <code>OPEN_MAX</code> , the highest file descriptor value allowed for the process. The <code>OPEN_MAX</code> value can be determined using <code>pathconf()</code> .  In a locking operation, <i>fildev</i> refers to a file with a type that does not support locking, or the <code>struct flock</code> pointed to by the third argument has an incorrect form.  If an <code>F_CLOSF</code> operation, the third argument, which specifies the upper limit, is less than <i>filedev</i> but is not equal to <code>-1</code> .  <b>Behavior for Sockets:</b> The <i>arg</i> parameter is not a valid flag, or the <i>cmd</i> parameter is not a valid command.
EMFILE	In an <code>F_DUPFD</code> operation, the process has already reached its maximum number of file descriptors, or there are no available file descriptors greater than the specified third argument.

ENOLCK	In an F_SETLK or F_SETLKW operation, the specified file has already reached the maximum number of locked regions allowed by the system.
EPERM	The operation was F_CLOSF, but all the requested file descriptors were not closed.

## Examples

### CBC3BF06

```

/* CBC3BF06
   This example illustrates one use of fcntl().
   The example will compile only with OS/390 C
*/
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>

void catcher(int signum) {
    puts("inside catcher...");
}

main() {
    int p[2], flags;
    struct sigaction sact;
    char c;

    if (pipe(p) != 0)
        perror("pipe() error");
    else {
        sigemptyset(&sact.sa_mask);
        sact.sa_flags = 0;
        sact.sa_handler = catcher;
        sigaction(SIGALRM, &sact, NULL);

        alarm(10);

        if (read(p[0], &c, 1) == -1)
            perror("first read() failed");

        if ((flags = fcntl(p[0], F_GETFL)) == -1)
            perror("first fcntl() failed");
        else if (fcntl(p[0], F_SETFL, flags | O_NONBLOCK) == -1)
            perror("second fcntl() failed");
        else {
            alarm(10);

            if (read(p[0], &c, 1) == -1)
                perror("second read() failed");

            alarm(0);
        }
        close(p[0]);
        close(p[1]);
    }
}

```

## Output

```

inside catcher...
first read() failed: Interrupted function call
second read() failed: Resource temporarily unavailable

```

Sockets example:

```

#define _OPEN_SYS
int s;
int rc;
int flags;
:
/* Place the socket into nonblocking mode */
rc = fcntl(s, F_SETFL, FNDELAY);

/* See if asynchronous notification is set */
flags = fcntl(s, F_GETFL, 0);
if (flags & FNDELAY)
    /* it is set */
else
    /* it is not */

```

## Related Information

- “fcntl.h” on page 28
- “sys/types.h” on page 49
- “unistd.h” on page 53
- “close() — Close a File” on page 191
- “dup() — Duplicate an Open File Descriptor” on page 288
- “dup2() — Duplicate an Open File Descriptor to Another” on page 290
- “exec Functions” on page 322
- “fsync() — Write Changes to Direct-Access Storage” on page 483
- “getsockopt() — Get the Options Associated with a Socket” on page 606
- “ioctl() — Control Device” on page 672
- “lseek() — Change the Offset of a File” on page 776
- “open() — Open a File” on page 872
- “setsockopt() — Set Options Associated with a Socket” on page 1270
- “socket() — Create a Socket” on page 1371

## fcvt() — Convert Double to String

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>
```

```
char *fcvt(double x, int ndigit,
           int *decpt, int *sign);
```

### General Description

The `fcvt()` function converts double floating-point argument values to floating-point output strings. The `fcvt()` function has been extended to determine the floating-point format (hexadecimal floating-point or IEEE floating-point) of double argument values by using `__isBFP()`.

OS/390 (C/C++) formatted output functions, including the `fcvt()` function, convert IEEE floating-point infinity and NaN argument values to special infinity and NaN floating-point number output sequences. See “*fprintf* Family of Formatted Output Functions” on page 441 for a description of the special infinity and Nan output sequences.

The `fcvt()` function converts `x` to a null-terminated string which has `ndigit` digits to the right of the radix point (where the total number of digits in the output string is restricted by the precision of a double) and returns a pointer to the string. The function behaves identically to “`ecvt()` — Convert Double to String” on page 303 in all respects other than the number of digits in the return value.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

If it succeeds, `fcvt()` returns the character equivalent of `x` as specified above.

If it is unable to allocate the return buffer, or the conversion fails, `fcvt()` returns `NULL`.

### Related Information

- “`stdlib.h`” on page 45
- “`ecvt()` — Convert Double to String” on page 303
- “`gcvt()` — Convert Double to String” on page 498
- “`__isBFP()` — Determine Application Floating-Point Format” on page 705

## fdelrec() — Delete a VSAM Record

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <stdio.h>
```

```
int fdelrec(FILE *stream);
```

### General Description

Removes the record previously read by `fread()` from the VSAM file associated with *stream*. The `fdelrec()` function can only be used after an `fread()` call has been performed and before any other operation on that file pointer. For example, if you need to acquire the file position using `ftell()` or `fgetpos()`, you can do it either before the `fread()` or after the `fdelrec()`. An `fread()` after an `fdelrec()` will retrieve the next record.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use `LANGLVL(EXTENDED)`.

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with `LANGLVL(EXTENDED)`. When you use `LANGLVL(EXTENDED)` any relevant information in the header is also exposed.

The `fdelrec()` function can be used with *key sequenced data sets* (KSDS), KSDS PATHs, and *relative record data set* (RRDS) opened in an update mode (that is, `rb+/r+b`, `wb+/w+b`, or `ab+/a+b`), with `type=record`.

VSAM does not support deletions from ESDSs.

### Returned Value

Returns 0 if the delete operation was successful and nonzero if the operation was not successful.

### Example

```
/* This example shows how a VSAM record is deleted using the fdelrec()
   function.
*/
#include <stdio.h>
FILE *stream;
char buf[80];
int num_read;
int rc;
stream = fopen("DD:MYCLUS", "rb+", type=record);
:
num_read = fread(buf, 1, sizeof(buf), stream);
rc = fdelrec(stream);
:
```

### **Related Information**

- “Performing VSAM I/O Operations” in the *OS/390 C/C++ Programming Guide*
- “stdio.h” on page 43
- “flocate() — Locate a VSAM Record” on page 405
- “fupdate() — Update a VSAM Record” on page 493

## fdetach() — Detach a Name from a STREAMS-based File Descriptor

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stropts.h>
```

```
int fdetach(const char *path);
```

### General Description

The `fdetach()` function detaches a STREAMS-based file from the file to which it was attached by a previous call to `fattach()`. The *path* argument points to the pathname of the attached STREAMS file. The process must have appropriate privileges or be the owner of the file. A successful call to `fdetach()` causes all pathnames that named the attached STREAMS file to again name the file to which the STREAMS file was attached. All subsequent operations on *path* will operate on the underlying file and not on the STREAMS file.

All open file descriptions established while the STREAMS file was attached to the file referenced by *path*, will still refer to the STREAMS file after the `fdetach()` has taken effect.

If there are no open file descriptors or other references to the STREAMS file, then a successful call to `fdetach()` has the same effect as performing the last `close()` on the attached file.

### Returned Value

If successful, `fdetach()` returns 0.

If unsuccessful, `fdetach()` returns `-1`, and returns an error value in `errno`.

**Note:** OS/390 UNIX services do not supply any STREAMS devices or pseudodevices. It is impossible for `fdetach()` to detach a file from a STREAMS-based file descriptor. See “`open()` — Open a File” on page 872 for more information.

The following are the possible values of `errno`:

- EACCES** Search permission is denied on a component of the path prefix.
- EPERM** The effective user ID is not the owner of *path* and the process does not have appropriate privileges.
- ENOTDIR** A component of the path prefix is not a directory.
- ENOENT** A component of *path* does not name an existing file or *path* is an empty string.
- EINVAL** The *path* argument names a file that is not currently attached.

### ENAMETOOLONG

The size of a pathname exceeds `PATH_MAX`, or a pathname component is longer than `NAME_MAX`, or pathname resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

ELOOP      Too many symbolic links were encountered in resolving *path*.

### Related Information

- “stropts.h” on page 46
- “fattach() — Attach a STREAMS-based File Descriptor to a File in the File System Name Space” on page 339



## fdopen() — Associate a Stream with an Open File Descriptor

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	

### Format

```
#define _POSIX_SOURCE
#include <stdio.h>
```

```
FILE *fdopen(int fildes, const char *options);
```

### General Description

Associates a stream with an open file descriptor. A *stream* is a pointer to a FILE structure that contains information about a file. A stream permits user-controllable buffering and formatted input and output. For a discussion of the OS/390 UNIX services implementation of buffering, see the *OS/390 C/C++ Programming Guide*.

The specified *options* must be permitted by the current mode of the file descriptor. For example, if the file descriptor is open-read-only (O\_RDONLY), the corresponding stream cannot be opened write-only (w).

These options are the same as for an fopen() operation:

To provide an ASCII input/output format for applications using this function, define feature test macro \_\_LIBASCII as described on page 22.

### Special Behavior for XPG4.2

The values for options are changed to include binary streams.

Mode	Description
r or rb	Open for reading.
w or wb	Open for writing.
a or ab	Open for appending.
r+ or rb+ or r+b	Open for update (reading and writing).
w+ or wb+ or w+b	Open for update (reading and writing).
a+ or ab+ or a+b	Open for update at end-of-file (reading and writing).

All these options have the same behavior as the corresponding *fopen()* options, except that w, wb, w+, wb+ and w+b do not truncate the file.

The file position indicator of the new stream is the file offset associated with the file descriptor. The error indicator and end-of-file indicator for the stream are cleared.

## Returned Value

If successful, `fdopen()` returns a FILE pointer to the control block for the new stream. If unsuccessful, it returns NULL and sets `errno` to one of the following:

- |        |   |
|--------|---|
| EINVAL | The specified mode is incorrect or does not match the mode of the open file descriptor. |
| EBADF  | <i>fd</i> is not a valid open file descriptor.  |

## Example

### CBC3BF08

```
/* CBC3BF08
   This example associates stream with the file descriptor fd, which is
   open for the file fdopen.file. The association is made in write mode.
*/
#define _POSIX_SOURCE
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

main() {
    char fn[]="fdopen.file";
    FILE *stream;
    int fd;

    if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        if ((stream = fdopen(fd, "w")) == NULL) {
            perror("fdopen() error");
            close(fd);
        }
        else {
            fputs("This is a test", stream);
            fclose(stream);
        }
        unlink(fn);
    }
}
```

## Related Information

- “stdio.h” on page 43
- “fopen() — Open a File” on page 417
- “fileno() — Get the File Descriptor from an Open Stream” on page 399
- “open() — Open a File” on page 872

## feof() — Test End-of-File Indicator

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
int feof(FILE *stream);
```

### General Description

Indicates whether the EOF flag is set for the given stream pointed to by *stream*.

The EOF flag is set when the user attempts to read past the EOF. Thus, a read of the last character in the file does not turn the flag on. A subsequent read attempt reaches the EOF.

For HFS files, a simultaneous reader cannot see the extensions automatically. Use `clearerr()` is required to reset the EOF flag.

If the file has a simultaneous writer that extends the file, the flag can be turned on by the reader before the file is extended. After the extension becomes visible to the reader, a subsequent read will get the new data and set the flag appropriately (see `fflush()`). For example, if the read does not read past the EOF, the flag is turned off. If a file does not have a simultaneous writer that is extending the file, it is not possible to read past EOF.

A successful repositioning in a file (with `fsetpos()`, `rewind()`, `fseek()`) or a call to `clearerr()` resets the EOF flag. For a terminal file, when the EOF flag is set, subsequent reads will continue to deliver no data until the EOF flag is cleared. This can be accomplished by calling `clearerr()` or `rewind()`.

The terminal can only read past the EOF after the `rewind()` function or the `clearerr()` function is called. The EOF flag is cleared by calling `rewind()`, `fsetpos()`, `fseek()`, or `clearerr()` for this stream.

### Returned Value

Returns a nonzero value if and only if the EOF flag is set for *stream*; otherwise, 0 is returned.

### Example

#### CBC3BF09

```
/* CBC3BF09
   This example scans the input stream until it reads an EOF character.
*/
#include <stdio.h>
#include <stdlib.h>

main() {
```

```
FILE *stream;
int rc;
stream = fopen("myfile.dat","r");

/* myfile.dat contains 3 characters "abc" */
while (1) {
    rc = fgetc(stream);
    if (rc == EOF) {
        if (feof(stream)) {
            printf("at EOF\n");
            break;
        }
        else {
            printf("error\n");
            break;
        }
    }
    else
        printf("read %c\n",rc);
}
```

**Output**

```
read a
read b
read c
at EOF
```

**Related Information**

- “stdio.h” on page 43
- “clearerr() — Reset Error and End-of-File” on page 187
- “fseek() — Change File Position” on page 474
- “fsetpos() — Set File Position” on page 477
- “rewind() — Set File Position to Beginning of File” on page 1139

## error() — Test for Read/Write Errors

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
int error(FILE *stream);
```

### General Description

Tests for an error in reading from or writing to the specified *stream*. If an error occurs, the error indicator for the *stream* remains set until you close the *stream*, call `rewind()`, or call `clearerr()`.

If an invalid parameter is given to an I/O function, OS/390 C/C++ does not turn the error flag on. This case differs from one where parameters are invalid in context with one another.

### Returned Value

Returns a nonzero value to indicate an error for the stream pointed to by *stream*; otherwise, it returns 0.

### Example

#### CBC3BF10

```
/* CBC3BF10
   This example puts data out to a stream and then checks that a write
   error has not occurred.
*/
#include <stdio.h>

int main(void)
{
    FILE *stream;
    char *string = "Important information";
    stream = fopen("myfile.dat","w");

    fprintf(stream, "%s\n", string);
    if (error(stream))
    {
        printf("write error\n");
        clearerr(stream);
    }
    if (fclose(stream))
        printf("fclose error\n");
}
```

**Related Information**

- “stdio.h” on page 43
- “clearerr() — Reset Error and End-of-File” on page 187
- “rewind() — Set File Position to Beginning of File” on page 1139

## fetch() — Get a Load Module

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both.	

### Format

```
#include <stdlib.h>
```

```
void (*fetch(const char *name))();
```

### General Description

Dynamically loads the load module specified by *name* into memory. The load module can then be invoked from a OS/390 C program. The name or the alias by which the fetchable load module is identified in the load module library must appear in a fetch() library function call.

To avoid infringing on the user's name-space, this non-standard function has two names. One name, the external entry point name, is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANTLRVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters \_\_fetch()), or compile with LANTLRVL(EXTENDED). When you use LANTLRVL(EXTENDED) any relevant information in the header is also exposed.

You cannot fetch a module that contains a main(). If you do, fetch() will not return a usable pointer. Use of the pointer will result in undefined behavior. To call these types of modules, use the system() library function. Alternatively, when creating the module, you can reset the entry point so that the linkage is provided by OS/390 C.

When non-reentrant modules have been fetched multiple times, you should release them in the reverse order; otherwise the load modules may not be deleted immediately.

You can fetch modules written in C and C++. For C modules, the source of the fetched module must, in general, contain #pragma linkage(..., fetchable) (the exception is described below). To fetch a C++ module, the routine must be declared extern "C", and must be declared in a #pragma linkage(..., fetchable) directive. See also "fetchep() — Share Writable Static" on page 382 for more information about the need for #pragma linkage.

If the fetched module is compiled as a DLL, it can import variables and functions from other DLL modules, but it cannot export variables or functions.

Nested fetching is supported. That is, a fetched module can also invoke the fetch() library function to dynamically load a separate fetchable module.

Multiple fetching is also supported. Fetching a module more than once will result in separate fetch pointers. If the module is marked "reentrant", multiple fetches will not reload the module into storage. Under MVS, you can place the reentrant module into the Extended Link Pack Area or the Link Pack Area (ELPA/LPA) to save time

on the load. Although multiple copies of the reentrant module are not brought into storage, each fetch returns a separate pointer. If a module is not reentrant, multiple fetches cause multiple loads into storage. Be aware that if you fetch() a non-reentrant module multiple times, the module may not get deleted by release() until all fetch instances have been released. Also, you should keep in mind that multiple loads of a non-reentrant module can be costly in terms of storage.

Writable statics are, in general, process scoped. The exception is that, when a thread calls a fetched module, the writable statics are changed for that thread only, that is, thread scoped.

Under MVS, fetchable (or dynamically loaded) modules must be link-edited and accessible using the standard system search. MVS supports fetching of non-reentrant, serially reusable, and reentrant modules. When RTLS(ON) is in effect, modules are first looked for in the RTLS logical library. If they are not found, they are loaded from the MVS standard system search.

Under POSIX, however, the fetchable, dynamically loaded modules cannot be in the HFS (Hierarchical File System). Note also, that the POSIX and XPG4 external variables are propagated. Refer to the *OS/390 C/C++ Programming Guide* for more information on external variables. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information about using POSIX support.

Unless your program is naturally reentrant, each reentrant module has a different copy of writable static. Follow these steps to allow the fetching of your reentrant module that has writable static:

1. Compile the module to be fetched with the RENT compile-time option.
2. Using the object module created in step 2, generate a fetchable member. You must specify the entry point as the function you are fetching unless you have included a `#pragma linkage(..., FETCHABLE)` directive.

See Figure 3 on page 371 for the program flow of a fetchable module. (FECB refers to a Fetch Control Block, which is a OS/390 C internal control block used by fetch().)

To dynamically fetch a set of functions with shared writable static, you can use the library function `fetchep()`. See “`fetchep()` — Share Writable Static” on page 382 for more details.

Both the module being fetched and the module invoking the `fetch()` library function can be reentrant.

You can fetch modules without specifying the directive, `#pragma linkage(..., FETCHABLE)`, in the fetched module. If you do, then using the fetch pointer will result in calling the entry point for that module. When you link the module, you must reset the entry point. In addition, you cannot have any writable static.

It follows that fetching a reentrant C module containing writable statics requires that you use the `#pragma linkage(..., FETCHABLE)` preprocessor directive in the fetched module.



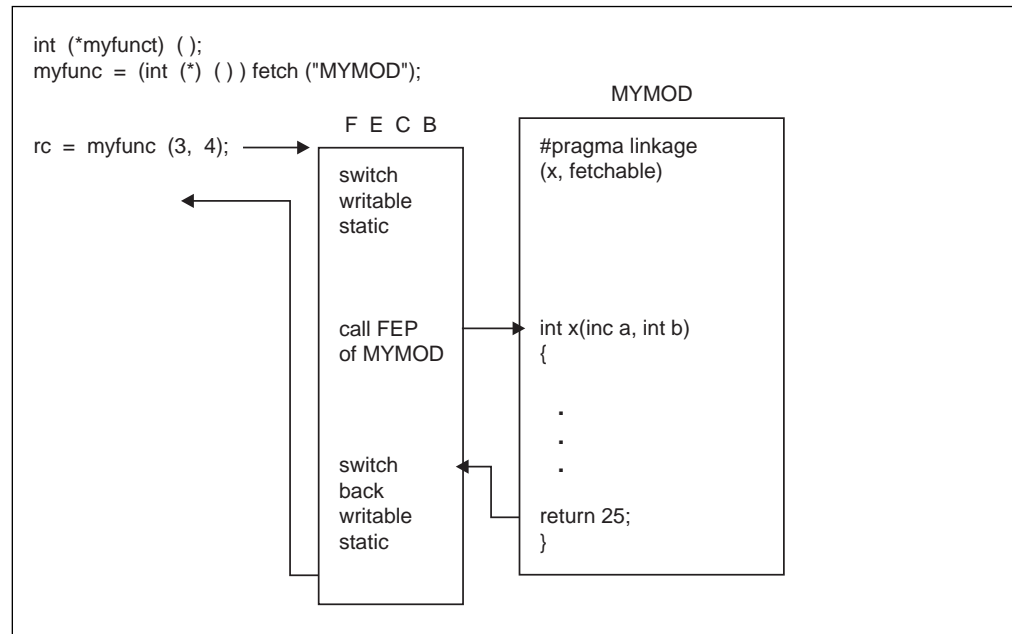


Figure 3. Program Flow of a Fetchable Module

If the entry point linkage is not a C linkage, you must use a `pragma linkage` with a function pointer defined by a *typedef*. The following sample excerpt would set up a COBOL linkage for a COBOL routine.

```

typedef int COBOL_FUNC ();
#pragma linkage (COBOL_FUNC, COBOL)
:
COBOL_FUNC * fetch_ptr;
fetch_ptr = (COBOL_FUNC *) fetch(module); /* loads fetched module */
fetch_ptr(args); /* sets up the proper linkage for the call */

```

Once the module is fetched, calling the fetched function is similar to making an interlanguage call.

`fetch()` also supports AMODE switching: when the function call is made, AMODE will be switched; upon return, the AMODE will be restored. Beware of calling fetched modules with AMODE=24 that try to access variables or the library above the line.

#### Notes:

- You cannot call functions through a function pointer that crosses load module boundaries, except through `fetchep()`. (See “`fetchep()` — Share Writable Static” on page 382 for more information.) For example, you cannot pass the address of a function to a fetched routine and invoke it from the fetched routine because the OS/390 C writable static will not be swapped.
- If you need to access code that has to run in a restricted addressing mode (such as a AMODE 24), you can package the code into a module to be fetched. The module can then be linked using the restricted addressing mode, but fetched from a program with an unrestricted addressing mode.
- A program that invokes `fetch()` many times without releasing all of the load modules may run out of memory.

## Returned Value

Returns a pointer to a stub that will call the entry point to the fetched load module. If the load fails, NULL is returned.

## Link Considerations

When linking the function to be fetched, you must link in the necessary libraries and specify the entry point as the function you are fetching unless you have included this directive: `#pragma linkage(..., FETCHABLE)`.

When linking the `main()` OS/390 C function, you must specify the necessary libraries to use the functions you are fetching. For example, if you are fetching a COBOL function, specify the COBOL library. This requirement does *not* apply to Language Environment.

When running `main()`, specify the runtime libraries you will need for `main()`, as well as the functions you will fetch. This requirement does *not* apply to Language Environment.

## Special Behavior for C++

A OS/390 C++ program cannot call `fetch()`. If you attempt to call `fetch()` from a OS/390 C++ program, the compiler will issue an error message. There are three alternatives to `fetch()` under OS/390 C++:

- You can replace `fetch()` with DLL (Dynamic Link Library) calls.
- You can provide a C DLL module to fetch modules, as shown at 375.
- A OS/390 C++ program may statically call a OS/390 C function that, in turn, fetches another module.

Refer to “Examples of Alternatives to `fetch()` Under C++” on page 375 for illustration of these points.

## Examples of Using `fetch()` with C

The following example demonstrates how to compile, link, and run a program that fetches a function in another object module that contains the directive:

`pragma linkage(..., FETCHABLE)`.

Begin with the main program.

```
#include <stdio.h>
#include <stdlib.h>

typedef int (*funcPtr)(); /* pointer to a function returning an int */

int main(void)
{
    int result;
    funcPtr add;

    printf("fetch module\n");
    add = (funcPtr) fetch("f1a");          /* load module F1A */

    if (add == NULL) {
        printf("ERROR: fetch failed\n");
    }
}
```

```

    }
    else {
        printf("execute fetched module\n");
        result = (*add)(1,2);          /* execute module F1A */

        printf("1 + 2 == %d\n", result);
    }
}

```

Then the fetched function:

```

#pragma linkage(func1, fetchable)

int func1(int a, int b)
{
    printf("in fetched module\n");

    return(a+b);
}

```

Next, JCL to compile, link, and run under MVS:

```

>
//F1A      EXEC EDCC,INFILE='userid.TEST.SOURCE(F1A)'
//          OUTFILE='userid.TEST.OBJ(F1A),DISP=SHR',
//          CPARM='NOSEQ,NOMARGIN,RENT'
//F1B      EXEC EDCPL,INFILE='userid.TEST.OBJ(F1A)'
//          OUTFILE='userid.TEST.LOAD(F1A),DISP=SHR'
//F1       EXEC EDCCLG,INFILE='userid.TEST.SOURCE(F1)'
//GO.STEPLIB DD
//          DD DSN=userid.TEST.LOAD,DISP=SHR

```

This example demonstrates the use of fetch() with COBOL and how to compile, link, and run the program.

```

/* cob1 */
#include <stdlib.h>
#include <stdio.h>

typedef void funcV();          /* function returning void      */
#pragma linkage(funcV, COBOL)  /* establish Cobol linkage */

int main(void)
{
    int var1 = 1;
    int var2 = 2;
    funcV *add;

    printf("fetch module\n");
    add = (funcV *) fetch("cob1a");          /* load module COB1A */

    if (add == NULL)
    {
        printf("ERROR: fetch failed\n");
    }
    else
    {
        printf("execute fetched module\n");
        (*add)(&var1, &var2);              /* execute module COB1A */

        printf("1 + 2 == %d\n", var1);
    }
}

```

Here is the fetched COBOL subroutine COB1A.

```

IDENTIFICATION DIVISION.
PROGRAM-ID. COB1A.
*****
* This subroutine receives 2 integer parameters. *
* The first is added to the second and the result is stored *
* back into the first. *
*****
ENVIRONMENT DIVISION.
DATA DIVISION.

WORKING-STORAGE SECTION.

LINKAGE SECTION.
01 VAR1 PIC S9(9) COMP.
01 VAR2 PIC S9(9) COMP.
*****
* PROCEDURE DIVISION *
*****

PROCEDURE DIVISION USING VAR1 VAR2.

*
* ADD VAR2 TO VAR1 PLACING THE RESULT IN VAR1.
*

COMPUTE VAR1 = VAR1 + VAR2.
GOBACK.

```

Finally, compile, link, and run under MVS:

```

//*****
//COBCL PROC CREGSIZ='2048K',
//      INFILE=,
// OUTFILE='&&GSET(GO),DISP=(MOD,PASS),UNIT=VIO,SPACE=(512,(50,20,1))'
//*
//*-----
//* COBOL Compile Step
//*-----
//COBCOMP EXEC PGM=IGYCRCTL,REGION=&CREGSIZ;
//STEPLIB DD DSNAME=IGY.V1R3M0.SIGYCOMP,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSIN DD DSNAME=&INFILE,DISP=SHR
//SYSLIN DD DSNAME=&LOADSET,UNIT=SYSDA,
//          DISP=(MOD,PASS),SPACE=(TRK,(3,3)),
//          DCB=(BLKSIZE=3200)
//SYSUT1 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT3 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT4 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT5 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT6 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT7 DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//*
//*-----
//* COBOL Link-Edit Step
//*-----
//COBLINK EXEC PGM=HEWL,COND=(8,LT,COBCOMP),REGION=1024K
//SYSLIB DD DSNAME=CEE.V1R3M0.SCEELKED,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSLIN DD DSNAME=&LOADSET,DISP=(OLD,DELETE)
//          DD DDNAME=SYSIN
//SYSLMOD DD DSNAME=&OUTFILE;

```

```
//SYSUT1 DD UNIT=SYSDA,SPACE=(TRK,(10,10))
// PEND
//*

//=====
//* Compile and Link-Edit COBOL program COB1A
//-----
//COB1A EXEC COBCL,
//      INFILE='userid.TEST.SOURCE(COB1A)',
//      OUTFILE='userid.TEST.LOAD(COB1A),DISP=SHR'
//COBLINK.SYSIN DD *
//      ENTRY COB1A
//
//
//-----
//* Compile and Link-Edit C program COB1
//-----
//COB1 EXEC EDCCLG,
//      INFILE='userid.TEST.SOURCE(COB1)',
//      CPARAM='OPT(0) NOSEQ NOMAR'
//GO.STEPLIB DD
//      DD DSNAME=userid.TEST.LOAD,DISP=SHR
```

### Examples of Alternatives to fetch() Under C++

This example shows how to use DLL as an alternative to fetch(). Here, myfunc() is the function to be dynamically loaded using DLL, and main() invokes DLL.

#### CBC3BF52

```
// CBC3BF52-part 1 of 2-other file is CBC3BF53.
// This example shows how to use DLL as an alternative to fetch().

// C++ program that invokes myfunc using DLL

#include <stdlib.h>
#include <stdio.h>
#include <dll.h>

extern "C" {
    // needed to indicate C linkage
    typedef int (*funcPtr)(); // pointer to a function returning an int
}

int main (void)
{
    dllhandle *dllh;
    funcPtr fptr;

    if ((dllh = dllload( "mydll" )) == NULL) {
        perror( "failed to load mydll" );
        exit( -1 );
    }
    if ((fptr = (funcPtr) dllqueryfn( dllh, "myfunc" )) == NULL) {
        perror( "failed to retrieve myfunc" );
        exit( -2 );
    }
    if ( fptr() != 0 ) {
        perror( "failed to execute myfunc" );
        exit( -3 );
    }
    if ( dllfree( dllh ) != 0 ) {
        perror( "failed to free mydll" );
        exit( -4 );
    }
}
```

```
    return( 0 );
}
```

### CBC3BF53

```
/* CBC3BF53-part 2 of 2-other file is CBC3BF52.
   This example shows how to use DLL as an alternative to fetch().
*/
```

```
/* C function dynamically loaded using DLL */
#include <stdio.h>
```

```
int myfunc (void)
{
    printf( "Hello world\n" );
    return( 0 );
}
```

The following example shows how a C++ program can dynamically call a function in a C DLL module, to fetch other C modules.

### CBC3BF54

```
// CBC3BF54-part 1 of 3-other files are CBC3BF55, CBC3BF56.
// This example shows how a C++ program can dynamically call a function
// in a C DLL module, to fetch other C modules
```

```
// C++ program that dynamically calls a function in a C DLL module
```

```
#include <stdio.h>
#include <stdlib.h>
#include <dll.h>
#include <iostream.h>
```

```
extern "C" {          // needed to indicate C linkage
    typedef int (*funcPtr)(); // pointer to a function returning an int
}
```

```
int main (void)
{
    dllhandle *dllh;
    funcPtr  fptr;

    if ((dllh = dllload( "mydll" )) == NULL) {
        perror( "failed to load mydll" );
        exit( -1 );
    }
    if ((fptr = (funcPtr) dllqueryfn( dllh, "fwrap" )) == NULL) {
        perror( "failed to retrieve fwrap" );
        exit( -2 );
    }
    if ( fptr() != 0 ) {
        perror( "failed to execute fwrap" );
        exit( -3 );
    }
    if ( dllfree( dllh ) != 0 ) {
        perror( "failed to free mydll" );
        exit( -4 );
    }
    return( 0 );
}
```

### CBC3BF55

```

/* CBC3BF55-part 2 of 3-other files are CBC3BF54, CBC3BF56.
   This example shows how a C++ program can dynamically call a function
   in a C DLL module, to fetch other C modules

   fwrap function used in a DLL module-it fetches mymod, which
   contains myfunc
*/
#include <stdio.h>
#include <stdlib.h>

typedef int (*funcPtr)(); /* pointer to a function returning an int */

int fwrap (void)
{
    funcPtr    fptr;

    if ((fptr = (funcPtr) fetch( "mymod" )) == NULL) {
        perror( "failed to fetch mymod" );
        return( -1 );
    }
    else
        return(fptr());
}

```

### CBC3BF56

```

/* CBC3BF56-part 3 of 3-other files are CBC3BF54, CBC3BF55.
   This example shows how a C++ program can dynamically call a function
   in a C DLL module, to fetch other C modules
*/

/*    C function to be fetched    */

#include <stdio.h>
#pragma linkage(myfunc, fetchable)

int myfunc (void)
{
    printf( "in fetched module\n" );
    return( 0 );
}

```

The following example shows how to statically call a C function that in turn fetches other functions. Here, myfunc() is the function to be fetched, fetcher() is a C function that fetches myfunc(), and main() is a function that statically calls fetcher().

### CBC3BF57

```

// CBC3BF57-part 1 of 3-other files are CBC3BF58, CBC3BF59.
// This example shows how to statically call a C function that
// fetches other functions.

// C++ statically calling a C program that uses fetch()

#include <iostream.h>

extern "C" {          // needed to indicate C linkage
    int fetcher (void);
}

int main (void)
{
    cout << "The fetcher says: ";
    fetcher();
}

```

```

    cout << "and returns";
    return( 0 );
}

```

### **CBC3BF58**

```

/* CBC3BF58-part 2 of 3-other files are CBC3BF57, CBC3BF59.
   This example shows how to statically call a C function that fetches
   other functions.
*/

/*
   C function that fetches mymod which contains myfunc
*/
#include <stdio.h>
#include <stdlib.h>

typedef int (*funcPtr)();  /* pointer to a function returning an int */

int fetcher (void)
{
    funcPtr fptr;

    if ((fptr = (funcPtr) fetch( "mymod" )) == NULL) {
        perror( "failed to fetch mymod" );
        return( -1 );
    }
    else {
        fptr();          /* invoke fetched function */
        return( 0 );
    }
}

```

### **CBC3BF59**

```

/* CBC3BF59-part 3 of 3-other files are CBC3BF57, CBC3BF58.
   This example shows how to statically call a C function that fetches
   other functions.
*/

/*   C function to be fetched   */

#include <stdio.h>
#pragma linkage(myfunc, fetchable)

int myfunc (void)
{
    printf( "Hello world " );
    return( 0 );
}

```

Although fetching and using DLL are functionally comparable, there is one subtle difference. Fresh copies of static and global variables are allocated each time a module is fetched, but not each time a DLL load of the same module is done.

The following example shows that, when a module is fetched multiple times, fresh copies of static and global variables are allocated.

### **CBC3BF60**



```

/* CBC3BF60-part 1 of 2-other file is CBC3BF61.
   This example shows how copies of variables are allocated when multiple
   fetches are done.
*/

/*
   C program fetching mymod multiple times--mymod contains myfunc.
*/
#include <stdio.h>
#include <stdlib.h>

typedef int (*funcPtr)(int); /*pointer to a function returning an int*/

int main (void)
{
    funcPtr fptr1, fptr2;

    if ((fptr1 = (funcPtr) fetch( "mymod" )) == NULL) {
        perror( "failed to fetch mymod" );
        return( -1 );
    }
    if ( fptr1(100) != 0 ) {
        perror( "failed to execute myfunc" );
        exit( -2 );
    }
    if ((fptr2 = (funcPtr) fetch( "mymod" )) == NULL) {
        perror( "failed to fetch mymod" );
        return( -3 );
    }
    if ( fptr2(100) != 0 ) {
        perror( "failed to execute myfunc" );
        exit( -4 );
    }
    return( 0 );
}

```

### CBC3BF61

```

/* CBC3BF61-part 2 of 2-other file is CBC3BF60.
   This example shows how copies of variables are allocated when multiple
   fetches are done.
*/

/*      C module mymod      */
#include <stdio.h>
#pragma linkage(myfunc, fetchable)

int globvar = 5;

int myfunc (int x)
{
    globvar += x;
    printf( "%d\n", globvar );
    return( 0 );
}

```

Running this example would produce the following results:

```

105
105

```

The following example shows that fresh copies of static and global variables are not allocated for multiple DLL loads of the same module.

**CBC3BF62**

```
// CBC3BF62-part 1 of 2-other file is CBC3BF63.
// This example shows how copies of variables are allocated when
// multiple DLL loads are done.

// C++ program doing multiple DLL loads of the same module

#include <stdlib.h>
#include <stdio.h>
#include <dll.h>

extern "C" { //needed to indicate C linkage
    typedef int (*funcPtr)(int); //pointer to a function returning an int
}

int main (void)
{
    dllhandle *dllh1, *dllh2;
    funcPtr fptr;

    if ((dllh1 = dllload( "mydll" )) == NULL) {
        perror( "failed to load mydll" );
        exit( -1 );
    }
    if ((fptr = (funcPtr) dllqueryfn( dllh1, "myfunc" )) == NULL) {
        perror( "failed to retrieve myfunc" );
        exit( -2 );
    }
    if ( fptr(100) != 0 ) {
        perror( "failed to execute myfunc" );
        exit( -3 );
    }
    if ((dllh2 = dllload( "mydll" )) == NULL) {
        perror( "failed to load mydll" );
        exit( -4 );
    }
    if ((fptr = (funcPtr) dllqueryfn( dllh2, "myfunc" )) == NULL) {
        perror( "failed to retrieve myfunc" );
        exit( -5 );
    }
    if ( fptr(100) != 0 ) {
        perror( "failed to execute myfunc" );
        exit( -6 );
    }
    if ( dllfree( dllh1 ) != 0 ) {
        perror( "failed to free mydll" );
        exit( -7 );
    }
    return( 0 );
}
```

**CBC3BF63**

```
/* CBC3BF63-part 2 of 2-other file is CBC3BF62.
   This example shows how copies of variables are allocated when multiple
   DLL loads are done.
*/

/* C function invoked using DLL */

#include <stdio.h>
#include <stdlib.h>
```

```
int globvar = 5;
int myfunc (int);

int myfunc (int x)
{
    globvar += x;
    printf( "%d\n", globvar );
    return( 0 );
}
```

Running this example would produce the following results:

```
105
205
```

### Related Information

- “Processing a Program with Multi-Threading” in the *OS/390 C/C++ Programming Guide*
- “stdlib.h” on page 45
- “fetchep() — Share Writable Static” on page 382
- “release() — Delete a Load Module” on page 1130.

## fetchep() — Share Writable Static

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	C only	

### Format

```
#include <stdlib.h>
```

```
void ( *fetchep( void ( *entry_point )()))();
```

### General Description

Dynamically fetches a set of functions with shared writable static variables.

fetchep() is used to register an entry point. It returns a pointer that may be passed across the fetch boundary and used as if it were the original entry point. Therefore, you can create more than one entry point from a fetched module. A call to the new entry point will use the same writable static as the original fetch pointer uses on each invocation.

fetchep() is called within a fetched module but not from the same level as the fetch() call. If fetchep() is called in the root program that is not a fetched module, fetchep() returns a fetch pointer that will use the root program's writable static (if any exists).

If the *entry\_point* given as input to fetchep() is a function address external to the current module or is an invalid function address, use of the resulting pointer returned from the call will result in undefined behavior.

If writable static is required, then this directive must be used:

```
#pragma linkage(entry_point, FETCHABLE)
```

In addition, the steps for fetching a reentrant module must be followed as described in “fetch() — Get a Load Module” on page 369. If writable static is *not* required, the C module using fetchep() need not contain the directive: #pragma linkage(..., FETCHABLE).

You can release the new fetch pointer without any effect on the original or any other fetch pointer created from the original fetch pointer. If the original fetched function is released, however, all the fetch pointers created using the fetchep() function will also be released. Trying to use a fetch pointer once it has been released or its origin has been released will result in undefined behavior.

To avoid infringing on the user's name-space, this non-standard function has two names. One name, the external entry point name, is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANTLRV(EXTENDED).

**Note:** The external entry point name for fetchep() is \_\_ftchep(), **NOT** \_\_fetchep().

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters \_\_ftchep()), or compile with LANTLRV(EXTENDED). When you use LANTLRV(EXTENDED) any relevant information in the header is also exposed.

## Examples

These examples and diagram demonstrate the program flow of a call to `fetch()` and subsequent calls to `fetchep()`.

```

/* The module that calls fetch() */
#include <stdlib.h>
typedef int (*FUNC_T)();

int main(void) {
    FUNC_T (*myfunc)();
    FUNC_T myfunc1;
    FUNC_T myfunc2;
    FUNC_T myfunc3;

    myfunc = (FUNC_T (*)())fetch("MYMOD");
    myfunc1 = myfunc(0);
    myfunc2 = myfunc(1);
    myfunc3 = myfunc(2);
}

/*
    The following code is the fetched module.
    Please see fetch() for information on how to compile and link the
    above.
*/

/* inside MYMOD */
#include <stdlib.h>
typedef int (*FUNC_T)();
int k; /* global variable to share within MYMOD */
#pragma linkage(x, fetchable)
FUNC_T x(int a)
{
    switch(a)
    {
        case 0:
            return (FUNC_T)fetchep((void(*)())func1);
        case 1:
            return (FUNC_T)fetchep((void(*)())func2);
        case 2:
            return (FUNC_T)fetchep((void(*)())func3);
    }
}

int func1(int a, int b)
{
    k = 6;
    :
}

int func2(int a, int b)
{
    k = 4;
    :
}

int func3(int a, int b)
{
    k = 5;
    :
}

```

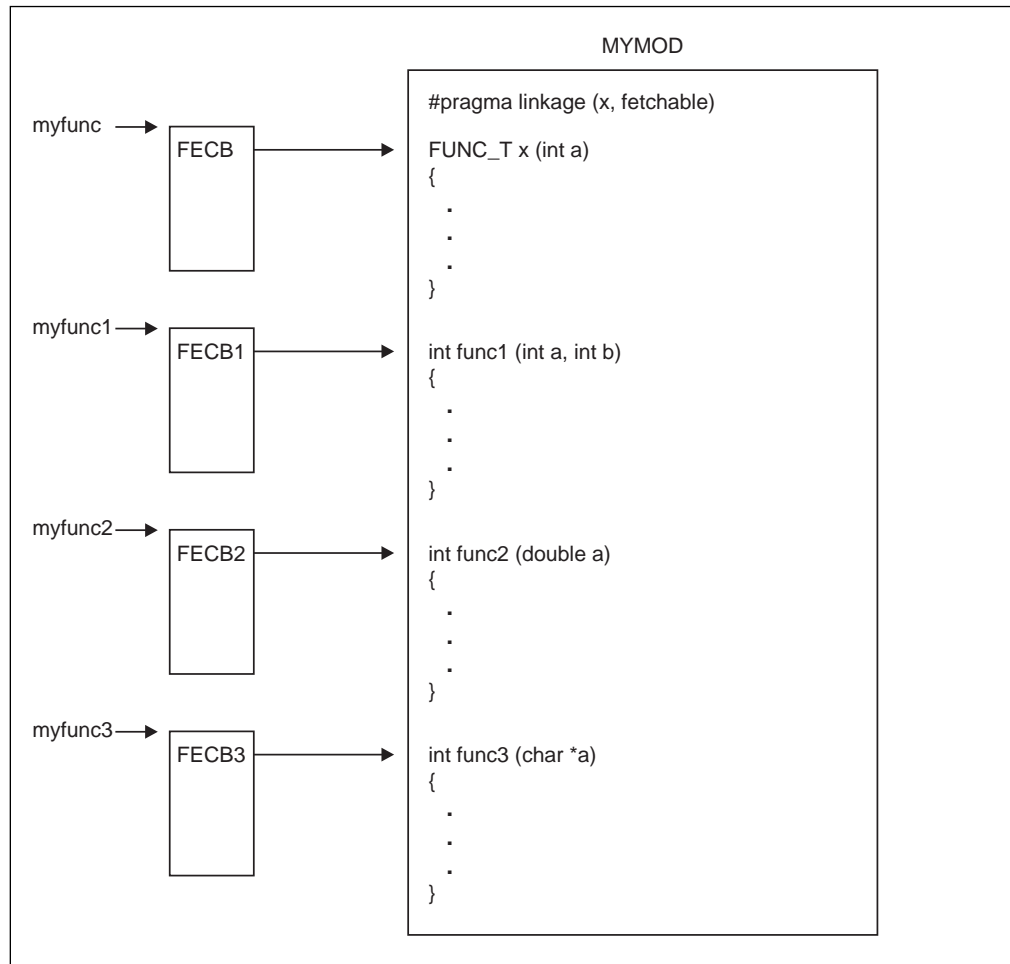


Figure 4. Program Flow of fetchep()

## Related Information

- “stdlib.h” on page 45
- “fetch() — Get a Load Module” on page 369
- “release() — Delete a Load Module” on page 1130

## fflush() — Write Buffer to File

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
int fflush(FILE *stream);
```

### General Description

Flushes the stream pointed to by *stream*. If *stream* is NULL, it flushes all open streams.

The fflush() function is affected by the ungetc() and ungetwc() functions. Calling these functions causes fflush() to back up the file position when characters are pushed back. For details, see the ungetc() and ungetwc() functions respectively. If desired, the \_EDC\_COMPAT environment variable can be set at open time such that fflush() discards any pushed back characters and leaves the file position where it was when the first ungetc() or ungetwc() function call was issued.

If fflush() is used after ungetwc() has pushed a wide char on a text stream, the position will be backed up by one wide character from the position the file was at when the ungetwc() was issued. For a wide-oriented binary stream, the position will be backed up based on the number of bytes used to represent the wide char in the ungetc buffer. For this reason, attempting to use ungetwc() on a character when the destination is a binary wide-oriented stream that was never read in the first place results in undefined behavior for fflush(). Note that the \_EDC\_COMPAT environment variable also changes the behavior of fflush() after ungetwc(), and will cause any wide char pushed back to be discarded and the position left at the point where the ungetwc() was issued. For details on the \_EDC\_COMPAT environment variable, see the “Environment Variables” in the *OS/390 C/C++ Programming Guide*.

If fflush() fails, the position is left at the point in the file where the first ungetc() or ungetwc() function call was issued. All pushed back characters are discarded.

**Note:** The system automatically flushes buffers when you close the stream or when a program ends normally without closing the stream.

The buffering mode and the file type can have an effect on when output data is flushed. For more information, see “Buffering of C Streams” in the *OS/390 C/C++ Programming Guide*.

*stream* remains open after the fflush() call. Because a read operation cannot immediately follow or precede a write operation, the fflush() function can be used to allow exchange between these two modes. The fflush() function can also be used to refresh the buffer when working with a reader and a simultaneous writer/updater.

## Returned Value

Returns the value 0 if it successfully flushes the buffer. The fflush() function returns EOF if an error occurs. When flushing all open files, a failure to flush any of the files causes EOF to be returned. However, flushing will continue on any other open files that can be flushed successfully.

## Example CBC3BF15

```
/* CBC3BF15
   This example flushes a stream buffer. It tests for the returned value of 0
   to see if the flushing was successful.
*/
#include <stdio.h>

int retval;
int main(void)
{
    FILE *stream;
    stream = fopen("myfile.dat", "w");

    retval=fflush(stream);
    printf("return value=%i",retval);
}
```

## Related Information

- “stdio.h” on page 43
- “setbuf() — Control Buffering” on page 1208
- “setvbuf() — Control Buffering” on page 1283
- “ungetc() — Push Character onto Input Stream” on page 1655
- “ungetwc() — Push a Wide Character onto a Stream” on page 1658



## ffs() — Find First Set Bit in an Integer

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <strings.h>
```

```
int ffs(int i);
```

### General Description

The `ffs()` function finds the first bit set (beginning with the least significant bit) and returns the index of that bit. Bits are numbered starting at one (the least significant bit).

### Returned Value

The `ffs()` function returns the index of the first bit set. If *i* is 0, then `ffs()` returns 0.

There are no `errno` values defined for `ffs()`

### Related Information

- “strings.h” on page 46

## fgetc() — Read a Character

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
int fgetc(FILE *stream);
```

### General Description

Reads a single-byte unsigned character from the input stream pointed to by *stream* at the current position, and increases the associated file pointer so that it points to the next character.

The fgetc() function is not supported for files opened with type=record.

fgetc() has the same restriction as any read operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

### Returned Value

Returns the character read as an integer. An EOF returned value indicates an error or an EOF condition. Use feof() or ferror() to determine whether the EOF value indicates an error or the end of the file. Note that EOF is only reached when an attempt is made to read “past” the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

### Example

#### CBC3BF16

```
/* CBC3BF16
   This example gathers a line of input from a stream.
   It tests to see if the file can be opened.
   If the file cannot be opened perror() is called.
*/
#include <stdio.h>
#define MAX_LEN 80

int main(void)
{
    FILE *stream;
    char buffer[MAX_LEN + 1];
    int i, ch;

    if ((stream = fopen("myfile.dat", "r")) != NULL) {
        for (i = 0; (i < (sizeof(buffer)-1)) &&
            ((ch = fgetc(stream)) != EOF) && (ch != '\n')); i++)
            printf("character is %d\n", ch);
        buffer[i] = ch;
    }
```

```
    buffer[i] = '\0';

    if (fclose(stream))
        perror("fclose error");
}
else
    perror("fopen error");
}
```

### Related Information

- “stdio.h” on page 43
- “feof() — Test End-of-File Indicator” on page 365
- “ferror() — Test for Read/Write Errors” on page 367
- “fgetwc() — Get Next Wide Character” on page 394
- “fputc() — Write a Character” on page 448
- “getc() - getchar() — Read a Character” on page 499

## fgetpos() — Get File Position

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
int fgetpos(FILE *stream, fpos_t *pos);
```

### General Description

Stores the current value of the file pointer associated with *stream* into the object pointed to by *pos*. The value pointed to by *pos* can be used later in a call to *fsetpos()* to reposition the stream pointed to by *stream*.

Both *fgetpos()* and *fsetpos()* functions also save state information for wide-oriented files. The value stored in *pos* is unspecified, and it is usable only by *fsetpos()*.

The position returned by *fgetpos()* is affected by the *ungetc()* and *ungetwc()* functions. Each call to these functions causes the file position indicator to be backed up from the position where the *ungetc()* or *ungetwc()* was issued. For details on how *ungetc()* affects *fgetpos()* behavior, see “*ungetc()* — Push Character onto Input Stream” on page 1655. For details on how *ungetwc()* affects *fgetpos()* behavior for a wide-oriented stream, see “*ungetwc()* — Push a Wide Character onto a Stream” on page 1658. Note that the `_EDC_COMPAT` environment variable can be set at open time such that *fgetpos()* will ignore any pushed back characters. For further details on `_EDC_COMPAT`, see the “Environment Variables” in the *OS/390 C/C++ Programming Guide*.

### Returned Value

If successful, the *fgetpos()* function returns 0. On error, *fgetpos()* returns a nonzero value and sets *errno* to a nonzero value.

### Special Behavior for XPG4.2

The *fgetpos()* function returns -1 and sets *errno* to *ESPIPE* if the underlying file type for the stream is a PIPE or a socket.

### Example CBC3BF17

```
/* CBC3BF17
   This example opens the file myfile.dat for reading.
   The current file pointer position is stored into the variable pos.
*/
#include <stdio.h>

int main(void)
{
    FILE *stream;
    int retcode;
    fpos_t pos;
```

```

stream = fopen("myfile.dat", "rb");

/* The value returned by fgetpos can be used by fsetpos()
   to set the file pointer if 'retcode' is 0 */

if ((retcode = fgetpos(stream, &pos)) == 0)
    printf("Current position of file pointer found\n");
fclose(stream);
}

```

### Related Information

- “stdio.h” on page 43
- “fseek() — Change File Position” on page 474
- “fsetpos() — Set File Position” on page 477
- “ftell() — Get Current File Position” on page 485
- “ungetc() — Push Character onto Input Stream” on page 1655
- “ungetwc() — Push a Wide Character onto a Stream” on page 1658

## fgets() — Read a String from a Stream

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
char *fgets(char *string, int n, FILE *stream);
```

### General Description

Reads bytes from a stream pointed to by *stream* into an array pointed to by *string*, starting at the position indicated by the file position indicator. Reading continues until the number of characters read is equal to  $n-1$ , or until a new-line character ( $\backslash n$ ), or until the end of the stream, whichever comes first. The `fgets()` function stores the result in *string* and adds a null character ( $\backslash 0$ ) to the end of the string. The *string* includes the new-line character, if read.

The `fgets()` function is not supported for files opened with `type=record`.

`fgets()` has the same restriction as any read operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

### Returned Value

Returns a pointer to the *string* buffer if successful. Otherwise it returns `NULL` to indicate failure.

If  $n$  is less than or equal to 0, it indicates a domain error; `errno` is set to `EDOM` to indicate the cause of the failure.

When  $n$  equals 1, it indicates a valid result. It means that the string buffer has only room for the null terminator; nothing is physically read from the file. (Such an operation is still considered a read operation, so it cannot immediately follow a write operation unless there is an intervening flush or reposition operation first.)

If  $n$  is greater than 1, `fgets()` will only fail if an I/O error occurs or if EOF is reached, and no data is read from the file.

The `ferror()` and `feof()` functions are used to distinguish between a read error and an EOF. Note that EOF is only reached when an attempt is made to read “past” the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

If EOF is reached after data has already been read into the string buffer, `fgets()` returns a pointer to the string buffer to indicate success. A subsequent call would

result in NULL being returned since EOF would be reached without any data being read.

### Example CBC3BF18

```
/* CBC3BF18
   This example gets a line of input from a data stream.
   It reads no more than MAX_LEN - 1 characters, or up to a new-line
   character, from the stream.
*/
#include <stdio.h>
#define MAX_LEN 100

int main(void)
{
    FILE *stream;
    char line[MAX_LEN], *result;

    stream = fopen("myfile.dat","r");

    if ((result = fgets(line,MAX_LEN,stream)) != NULL)
        printf("The string is %s\n", result);

    if (fclose(stream))
        printf("fclose error\n");
}
```

### Related Information

- “stdio.h” on page 43
- “feof() — Test End-of-File Indicator” on page 365
- “ferror() — Test for Read/Write Errors” on page 367
- “fgetc() — Read a Character” on page 388
- “fgetws() — Get a Wide-Character String” on page 396
- “fputs() — Write a String” on page 450
- “gets() — Read a String” on page 595
- “puts() — Write a String” on page 1059

## fgetwc() — Get Next Wide Character

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h>
```

```
wint_t fgetwc(FILE *stream);
```

### General Description

Obtains the next multibyte character from the input stream pointed to by *stream*, converts it to a wide character, and advances the associated file position indicator for the stream (if defined).

The behavior of this wide-character function is affected by the LC\_CTYPE category of the current locale. If you change the category, undefined results can occur.

Using non-wide-character functions with fgetwc() results in undefined behavior. This happens because fgetwc() processes a whole multibyte character and does not expect to be “within” such a character. In addition, fgetwc() expects state information to be set already. Because functions like fgetc() and fputc() do not obey such rules, their results fail to meet the assumptions made by fgetwc().

fgetwc() has the same restriction as any read operation for read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

### Returned Value

Returns the next wide character that corresponds to the multibyte character from the input stream pointed to by *stream*. If the stream is at EOF, the EOF indicator for the stream is set and fgetwc() returns WEOF. If a *read* error occurs, the error indicator for the stream is set and fgetwc() returns WEOF. If an *encoding* error occurs (an error converting the multibyte character into a wide character), the value of the macro EILSEQ (illegal sequence) is stored in errno and WEOF is returned.

The ferror() and feof() functions are used to distinguish between a read error and an EOF. Note that EOF is only reached when an attempt is made to read “past” the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

### Example

#### CBC3BF19

```
/* CBC3BF19 */
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>
```



```

int main(void)
{
    FILE    *stream;
    wint_t   wc;

    if ((stream = fopen("myfile.dat", "r")) == NULL) {
        printf("Unable to open file\n");
        exit(1);
    }

    errno = 0;
    while ((wc = fgetwc(stream)) != WEOF)
        printf("wc=0x%X\n", wc);

    if (errno == EILSEQ) {
        printf("An invalid wide character was encountered.\n");
        exit(1);
    }

    fclose(stream);
}

```

### Related Information

- “stdio.h” on page 43
- “wchar.h” on page 54
- “fgetc() — Read a Character” on page 388
- “fgetws() — Get a Wide-Character String” on page 396
- “fputwc() — Output a Wide-Character” on page 452

## fgetws() — Get a Wide-Character String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h>
```

```
wchar_t *fgetws(wchar_t *wcs, int n, FILE *stream);
```

### General Description

Reads at most one less than the number of the wide characters specified by *n*, from the stream pointed to by *stream*, into the array pointed to by *wcs*. No additional wide characters are read after a new-line wide character (which is retained) or after the EOF. A null wide character is written immediately after the last wide character read into the array.

The `fgetws()` function advances the file position unless there is an error, when the file position is undefined.

The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you change the category, undefined results can occur.

Using non-wide-character functions with `fgetws()` results in undefined behavior. This happens because `fgetws()` processes a whole multibyte character and does not expect to be “within” such a character. In addition, `fgetws()` expects state information to be set already. Because functions like `fgetc()` and `fputc()` do not obey such rules, their results fail to meet the assumptions made by `fgetws()`.

`fgetws()` has the same restriction as any read operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

### Returned Value

Returns the new value of *wcs* if successful. If EOF is encountered and no wide characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error or an encoding error occurs during the operation, the array contents are indeterminate and a null pointer is returned. An *encoding* error is one that occurs when a wide character is converted to a multibyte character. If it occurs, `errno` is set to `EILSEQ` and `NULL` is returned.

If *n* is less than or equal to 0, it indicates a domain error; `errno` is set to `EDOM` to indicate the cause of the failure.

When *n* equals 1, it indicates a valid result. It means that the string buffer has only room for the null terminator; nothing is physically read from the file. (Such an oper-

ation is still considered a read operation, so it cannot immediately follow a write operation unless there is an intervening flush or reposition operation first.)

If *n* is greater than 1, fgets() will only fail if an I/O error occurs or if EOF is reached, and no data is read from the file. To find out which error has occurred, use either the feof() or the ferror() function. If EOF is reached after data has already been read into the string buffer, fgetws() returns a pointer to the string buffer to indicate success. A subsequent call would result in NULL being returned because EOF would be reached without any data being read.

The ferror() and feof() functions are used to distinguish between a read error and an EOF. Note that EOF is only reached when an attempt is made to read “past” the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

### Example CBC3BF20

```
/* CBC3BF20 */
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
    FILE    *stream;
    wchar_t  wcs[100];
    wchar_t  *ptr;

    if ((stream = fopen("myfile.dat", "r")) == NULL) {
        printf("Unable to open file\n");
        exit(1);
    }

    errno = 0;
    ptr = fgetws(wcs, 100, stream);

    if (ptr == NULL) {
        if (errno == EILSEQ) {
            printf("An invalid wide character was encountered.\n");
            exit(1);
        }
        else if (feof(stream))
            printf("end of file reached\n");
        else
            perror("read error");
    }

    printf("wcs=\"%ls\"\n", wcs);

    fclose(stream);
}
```

### **Related Information**

- “stdio.h” on page 43
- “wchar.h” on page 54
- “fgets() — Read a String from a Stream” on page 392
- “fgetwc() — Get Next Wide Character” on page 394
- “fputws() — Output a Wide-Character String” on page 454.

## fileno() — Get the File Descriptor from an Open Stream

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a XPG4 XPG4.2	both	

### Format

```
#define _POSIX_SOURCE
#include <stdio.h>
int fileno(const FILE *stream);
```

### General Description

Returns the file descriptor number associated with a specified OS/390 C/C++ I/O stream. The argument *stream* points to a FILE structure controlling a OS/390 C/C++ I/O stream.

The unistd.h header file defines the following macros, which are constants that map to the file descriptors of the standard streams:

```
STDIN_FILENO
    Standard input, stdin (value 0)

STDOUT_FILENO
    Standard output, stdout (value 1)

STDERR_FILENO
    Standard error, stderr (value 2)
```

Note that *stdin*, *stdout*, and *stderr* are macros, not constants.

### Returned Value

Returns the file descriptor number associated with an open HFS stream (that is, one opened with *fopen()* or *freopen()*). MVS datasets are not supported, so *fileno()* of an MVS data set returns *-1*.

If unsuccessful, *fileno()* returns *-1*, and sets *errno* to *EBADF*, which indicates *one of the following*:

- *stream* points to a closed stream
- *stream* is an incorrect *stream* pointer
- *stream* points to a stream associated with an MVS data set.

### Example

#### CBC3BF21

```
/* CBC3BF21
   This example illustrates one use of fileno().
*/
#define _POSIX_SOURCE
#include <errno.h>
#include <stdio.h>
```

```
main() {
    FILE *stream;
    char hfs_file[] = "./hfs_file", mvs_ds[] = "//mvs.ds";

    printf("fileno(stdin) = %d\n", fileno(stdin));

    if ((stream = fopen(hfs_file, "w")) == NULL)
        perror("fopen() error for HFS file");
    else {
        printf("fileno() of the HFS file is %d\n", fileno(stream));
        fclose(stream);
        remove(hfs_file);
    }

    if ((stream = fopen(mvs_ds, "w")) == NULL)
        perror("fopen() error for MVS data set");
    else {
        errno = 0;
        printf("fileno() returned %d for the MVS data set,\n", fileno(stream));
        printf("errno=%s\n", strerror(errno));
        fclose(stream);
        remove(mvs_ds);
    }
}
```

## Output

```
fileno(stdin) = 0
fileno() of the HFS file is 3
fileno() returned -1 for the MVS data set,
errno=Bad file descriptor
```

## Related Information

- The “Standard Streams” chapter in the *OS/390 C/C++ Programming Guide*
- “stdio.h” on page 43
- “fdopen() — Associate a Stream with an Open File Descriptor” on page 363
- “fopen() — Open a File” on page 417
- “freopen() — Redirect an Open File” on page 460
- “open() — Open a File” on page 872

## finite() — Determine the Infinity Classification of a Floating-Point Number

### Standards

Standards / Extensions	C or C++	Dependencies
	both	OS/390 V2R6

### Format

```
#include <math.h>
```

```
int finite(x)
```

```
double x;
```

### General Description

The `finite()` function determines the infinity classification of floating-point number `x`.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

The `finite()` function returns a nonzero value if the `x` parameter is a finite number, i.e. if `x` is not `+-`, `INF`, `NaNQ`, or `NaNS`.

The `finite()` function does not return errors or set bits in the floating-point exception status, even if a parameter is a `NaNS`.

### Related Information

- IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standards 754-1985 and 854-1987).

## fldata() — Retrieve File Information

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <stdio.h>
```

```
int fldata(FILE *file, char *filename, fldata_t *info);
```

### General Description

Retrieves information about an open stream pointed to by *file*. It returns the file name in *filename* and other information in the structure *info*. The file name returned in *filename* is the name specified in `fopen()` or `freopen()`. If the file is opened with a *ddname* (for example, `fopen("DD:A", "w")`), then the *filename* field will contain the *ddname* used to open the file, prefixed with `dd:.` If the file is a DASD data set or a memory file, the field `__dsname` contains the *dsname*. If the file is an HFS file, the field `__dsname` contains the *pathname*. For all other files, it is `NULL`.

After a failure, the contents of the information structure are indeterminate.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use `LANGLVL(EXTENDED)`.

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with `LANGLVL(EXTENDED)`. When you use `LANGLVL(EXTENDED)` any relevant information in the header is also exposed.

For full details about *filename* considerations, see one of the “Opening Files” sections in the *OS/390 C/C++ Programming Guide*.

If *fldata* is the first reference to a standard stream, a call to the `fldata()` function opens the stream.

See Table 19 on page 403.

### Special Behavior for POSIX C

Under OS/390 UNIX services, if there had been an `exec` to an application that invokes `fldata()`, the standard streams are opened at the time of the `exec`. Thus `fldata()` does not attempt to open them again. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information about using POSIX support.



## Returned Value

If successful, fldata() returns 0; otherwise, it returns a nonzero value.

Table 19 (Page 1 of 2). Elements Returned in fldata\_t Data Structure

Element	Data Type	General Description
__recfmF:1	unsigned int	Indicates whether it has fixed-length records.
__recfmV:1	unsigned int	Indicates whether it has variable-length records.
__recfmU:1	unsigned int	Indicates whether it has undefined-length records.
__recfmS:1	unsigned int	Indicates whether it has either standard (if fixed-length) or spanned (if variable-length) records.
__recfmBlk:1	unsigned int	Indicates whether it has blocked records.
__recfmASA:1	unsigned int	Indicates whether it has ASA print-control characters.
__recfmM:1	unsigned int	Indicates whether it has machine print-control codes.
__dsorgPO:1	unsigned int	Indicates whether it is a partitioned data set.
__dsorgPDSmem:1	unsigned int	Indicates whether a file is a member.
__dsorgPDSdir:1	unsigned int	Indicates whether a file is a PDS or PDSE directory.
__dsorgPS:1	unsigned int	Indicates whether it is a sequential data set.
__dsorgConcat:1	unsigned int	Indicates whether it is a sequentially concatenated file.
__dsorgMem:1	unsigned int	Indicates whether it is a memory file.
__dsorgHiper:1	unsigned int	Indicates whether it is a memory file in hiperspace.
__dsorgTemp:1	unsigned int	Indicates whether it is a temporary file created by tmpfile().
__dsorgVSAM:1	unsigned int	Indicates whether it is a VSAM file.
__dsorgHFS:	unsigned int	Indicates whether it is an HFS file.
__openmode:2	unsigned int	Values are __TEXT, __BINARY, __RECORD.
__modeflag:4	unsigned int	Values are __APPEND, __READ, __UPDATE, __WRITE. These macros can be added together to determine the value; for example, a file opened with mode a+ will have the value __APPEND + __UPDATE.
__dsorgPDSE:1	unsigned int	Indicates whether a file is a PDSE.
__vsamRLS:3	unsigned int	Returned values are __NORLS and __RLS.
__reserve2:5	unsigned int	Reserved bits.
__device	char	Returned values are __DISK, __TERMINAL, __PRINTER, __TAPE, __TDQ, __DUMMY, __OTHER, __MEMORY, __MSGFILE, __HFS, __HIPERSPACE.
__blksize	unsigned long	Total block size of the file, including all control information needed in the block.
__maxreclen	unsigned long	Maximum length of the data in the record, including ASA control characters, if present.
__vsamtype	unsigned short	Returned values are __NOTVSAM, __ESDS, __KSDS, __RRDS, __ESDS_PATH, __KSDS_PATH.
__vsamkeylen	unsigned long	Length of VSAM key (if any).
__vsamRKP	unsigned long	Key position.

Table 19 (Page 2 of 2). Elements Returned in `fldata_t` Data Structure

Element	Data Type	General Description
<code>__dsname</code>	<code>char *</code>	<p>The contents of this field is determined by the following:</p> <ul style="list-style-type: none"> <li>• If the file is a DASD data set, memory file, or an HFS file, then <code>__dsname</code> contains the real file name of file opened by <code>ddname</code></li> <li>• If you open by <code>ddname</code>, and the <code>ddname</code> is a concatenation of PDS or PDSE data sets, then <code>__dsname</code> contains the data set name of the first PDS or PDSE. This is because you are only opening the directory of the first PDS or PDSE.</li> <li>• If you open by <code>ddname(member)</code> and the <code>ddname</code> is a concatenation of PDS or PDSE data sets, then <code>__dsname</code> contains the data set name of the first PDS or PDSE containing the member.</li> </ul> <p>Otherwise this field is NULL.</p> <p>The <code>char *__dsname</code> field is allocated internally by the library functions and must be saved before the next call to the <code>fldata()</code> function.</p>
<code>__reserve4</code>	unsigned long	Reserved.

### Example

```
#include <stdio.h>

int main(void)
{
    FILE *stream;
    char filename[100];
    fldata_t fileinfo;
    int rc;

    stream = fopen("myfile.dat","rb+");
    :
    rc = fldata(stream, filename, &fileinfo);
    if (rc != 0)
        printf("fldata failed\n");
    else
        printf("filename is %s\n",filename);
}
```

### Related Information

- “stdio.h” on page 43
- “fopen() — Open a File” on page 417
- “freopen() — Redirect an Open File” on page 460

flocate() — Locate a VSAM Record

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdio.h>

int flocate(FILE *stream, const void *key, size_t key_len, int
options);
```

General Description

Moves the VSAM file position indicator associated with the stream pointed to by *stream*, according to the rest of the arguments specified.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

*key* points to a key used for positioning.

*key\_len* specifies the length of the search key. The *key\_len* argument value must always be nonzero, except for \_\_KEY\_FIRST and \_\_KEY\_LAST.

KSDS, KSDS PATH, and ESDS PATH

The *key* can point to a field of any storage type except register. Typically it points to a character string whose length is *key\_len*. The *key\_len* must be less than or equal to the key length of the data set. If *key\_len* is the same as the file's key length, a full key search is automatically used; otherwise, a generic search is used. A generic key search is one in which the search key is a leading portion of the key field. The record positioned to is the first of the records having the same generic key.

- ESDS** The *key* points to a relative byte address stored as an unsigned long int. *key\_len* is always 4.
- RRDS** The *key* points to a relative record number stored as an unsigned long int. *key\_len* is always 4.

*options* specifies the position options described in Table 20.

Table 20 (Page 1 of 2). Position Options Parameter for flocate()

__KEY_FIRST	Positions to the first record in the file. Subsequent reads are in the forward direction. <i>key</i> and <i>key_len</i> are ignored.
-------------	--

Table 20 (Page 2 of 2). Position Options Parameter for flocate()

__KEY_LAST	Positions to the last record in the file. Subsequent reads are in backward order. <i>key</i> and <i>key_len</i> are ignored.  Only applies to VSAM files opened in record mode.
__KEY_EQ	Positions to the first record with the specified key. Subsequent reads are in the forward direction.
__KEY_EQ_BWD	Positions to the first record with the specified key. Subsequent reads are in backward order.  Only applies to VSAM files opened in record mode.
__KEY_GE	Positions to the first record with a key greater than or equal to the specified key.
__RBA_EQ	Positions to the record with the specified RBA. Subsequent reads are in the forward direction.  You cannot use __RBA_EQ with an alternative index path.  Using this option with RRDS is <i>not</i> recommended. The underlying VSAM utilities do not support seeking to an RBA in an RRDS file. The flocate() function attempts to convert the RBA to a Relative Record Number by dividing the value by the LRECL of the file and using the equivalent __KEY_EQ.  Using this option with KSDS is <i>not</i> recommended because the RBA of a given record may change over time, because of inserts, deletions, or updates of other records.
__RBA_EQ_BWD	Positions to the record with the specified RBA. Subsequent reads are in backward order.  You cannot use __RBA_EQ_BWD with an alternative index path.  Using this option with RRDS is not recommended. The underlying VSAM utilities do not support seeking to an RBA in an RRDS file. The flocate() function attempts to convert the RBA to a Relative Record Number by dividing the value by the LRECL of the file and using the equivalent __KEY_EQ_BWD.  Using this option with KSDS is <i>not</i> recommended because the RBA of a given record may change over time, because of inserts, deletions, or updates of other records.  Only applies to VSAM files opened in record mode.

**Notes:**

- When you are trying to use flocate() in a path to a nonunique key, the resulting position will be at the first physical record of the duplicate key set.
- flocate() releases all record locking.
- Writes to VSAM data sets are not affected by preceding calls to flocate().
- If a record was not found, you must successfully relocate to another position before reading or writing (using the flocate() function). The exception to this is that a write that follows a failed flocate() will succeed if the file was opened for initial loading, but no records have been written to it yet.

## Returned Value

Returns 0 if it was successful. Otherwise, if a record was not found or the position is beyond the EOF, then an EOF is returned.

## Example

```
#include <stdio.h>

int main(void)
{
    FILE *stream;
    int    vsam_rc;
    char *key = "RECORD 27";

    stream = fopen("DD:MYCLUS", "rb+", type=record);
    vsam_rc = flocate(stream, key, 9, __KEY_EQ);
    :
}
```

## Related Information

- “Performing VSAM I/O Operations” in the *OS/390 C/C++ Programming Guide*
- “stdio.h” on page 43
- “fdelrec() — Delete a VSAM Record” on page 359
- “fgetpos() — Get File Position” on page 390
- “fseek() — Change File Position” on page 474
- “fsetpos() — Set File Position” on page 477
- “ftell() — Get Current File Position” on page 485
- “fupdate() — Update a VSAM Record” on page 493

## floor() — Round Down to Integral Value

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double floor(double x);
```

### General Description

Calculates the largest integer that is less than or equal to *x*.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

Returns the calculated integral value expressed as a double value. The result cannot have a range error.

### Example

#### CBC3BF24

```
/* CBC3BF24
   This example assigns y the value of the largest integer that is less
   than or equal to 2.8, and it assigns z the value of the largest integer
   that is less than or equal to -2.8.
*/
#include <math.h>
#include <stdio.h>

int main(void)
{
    double y, z;

    y = floor(2.8);
    z = floor(-2.8);

    printf("floor( 2.8 ) = %f\n", y);
    printf("floor( -2.8 ) = %f\n", z);
}
```

### Output

```
floor( 2.8 ) = 2.000000
floor( -2.8 ) = -3.000000
```

**Related Information**

- “math.h” on page 35
- “ceil() — Round Up to Integral Value” on page 153
- “fmod() — Calculate Floating-Point Remainder” on page 410

## fmod() — Calculate Floating-Point Remainder

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double fmod(double x, double y);
```

### General Description

Calculates the floating-point remainder of  $x/y$ . The absolute value of the result is always less than the absolute value of  $y$ . The result will have the same sign as  $x$ .

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

If  $y$  is 0, or the result would overflow, then `fmod()` returns 0. `Errno` remains unchanged.

### Example

#### CBC3BF25

```
/* CBC3BF25
   This example computes z as the remainder of x/y; here x/y is -3 with a
   remainder of -1.
*/
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, y, z;

    x = -10.0;
    y = 3.0;
    z = fmod(x,y);      /* z = -1.0 */

    printf("fmod( %f, %f) = %lf\n", x, y, z);
}
```

### Output

```
fmod( -10.000000, 3.000000) = -1.000000
```



**Related Information**

- “math.h” on page 35

## fmtmsg() — Display a Message in the Specified Format

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <fmtmsg.h>
```

```
int fmtmsg(long classification, const char *label, int severity,
           const char *text, const char *action, const char *tag);
```

### General Description

The `fmtmsg()` function can be used to display messages in a specified format instead of the traditional `printf()` function.

Based on a message's classification component, `fmtmsg()` writes a formatted message either to standard error, to the console, or to both.

A formatted message consists of up to five components as defined below. The component classification is not part of a message displayed to the user, but defines the source of the message and directs the display of the formatted message.

*classification* Contains identifiers from the following groups of major classifications and subclassifications. Any one identifier from a subclass may be used in combination with a single identifier from a different subclass. Two or more identifiers from the same subclass should not be used together, with the exception of identifiers from the display subclass. (Both display subclass identifiers may be used so that messages can be displayed to both standard error and the system console).

#### Major Classifications

Identifies the source of the condition. Identifiers are: **MM\_HARD** (hardware), **MM\_SOFT** (software), and **MM\_FIRM** (firmware).

#### Message Source Subclassifications

Identifies the type of software in which the problem is detected. Identifiers are: **MM\_APPL** (application), **MM\_UTIL** (utility), and **MM\_OPSYS** (operating system).

#### Display Subclassifications

Indicates where the message is to be displayed. Identifiers are: **MM\_PRINT** to display the message on the standard error stream, **MM\_CONSOLE** to display the message on the system console. One or both identifiers may be used.

#### Status Subclassifications

Indicates whether the application will recover from the condition. Identifiers are: **MM\_RECOVER** (recoverable) and **MM\_NRECOV** (non-recoverable).

	An additional identifier, <b>MM_NULLMC</b> , indicates that no classification component is supplied for the message.
<i>label</i>	Identifies the source of the message. The format is two fields separated by a colon. The first field is up to 10 bytes, the second is up to 14 bytes. The constant <b>__MM_MXLABELLN</b> defines the maximum length of <i>label</i> .
<i>severity</i>	<p>Indicates the seriousness of the condition. Identifiers for the levels of severity are:</p> <p><b>MM_HALT</b> indicates that the application has encountered a severe fault and is halting. Produces the string "HALT".</p> <p><b>MM_ERROR</b> indicates that the application has detected a fault. Produces the string "ERROR".</p> <p><b>MM_WARNING</b> indicates a condition that is out of the ordinary, that might be a problem, and should be watched. Produces the string "WARNING".</p> <p><b>MM_INFO</b> provides information about a condition that is not in error. Produces the string "INFO".</p> <p><b>MM_NOSEV</b> indicates that no severity level is supplied for the message.</p> <p>Other provides an unknown severity. Produce the string "SV=n", where n is the <i>severity</i> value specified.</p>
<i>text</i>	Describes the error condition that produced the message. The character string is not limited to a specific size. If the character string is empty, then the text produced is unspecified.
<i>action</i>	Describes the first step to be taken in the error-recovery process. The <code>fmtmsg()</code> function precedes the action string with the prefix: "TO FIX:". The action string is not limited to a specific size.
<i>tag</i>	An identifier that references on-line documentation for the message. Suggested usage is that tag includes the label and a unique identifying number. A sample tag is "XSI:cat:146". The constant <b>__MM_MXTAGLN</b> defines the maximum length of <i>tag</i> .

The MSGVERB environment variable (for message verbosity) tells `fmtmsg()` which message components it is to select when writing messages to standard error. The value of MSGVERB is a colon-separated list of optional keywords. Valid keywords are: label, severity, text, action, and tag. If MSGVERB contains a keyword for a component and the component's value is not the component's null value, `fmtmsg()` includes that component in the message when writing the message to standard error. If MSGVERB does not include a keyword for a message component, that component is not included in the display of the message. The keywords may appear in any order. If MSGVERB is not defined, if its value is the null string, if its value is not of the correct format, or if it contains keywords other than the valid ones listed above, `fmtmsg()` selects all components.

MSGVERB affects only which components are selected for display to standard error. All message components are included in console messages.

### Returned Value

The `fmtmsg()` function returns one of the following values:

**MM\_OK**      The function succeeded.

**MM\_NOTOK**  
The function failed completely.

**MM\_NOMSG**  
The function was unable to generate a message on standard error, but otherwise succeeded.

**MM\_NOCON**  
The function was unable to generate a console message, but otherwise succeeded.

### Examples

The following is an example of `fmtmsg()`:

```
fmtmsg(MM_PRINT, "XSI:cat", MM_ERROR, "illegal option",  
"refer to cat in user's reference manual", "XSI:cat:001")
```

produces a complete message in the specified message format:

```
XSI:cat: ERROR: illegal option  
TO FIX: refer to cat in user's reference manual XSI:cat:001
```

The following is another example when the environment variable `MSGVERB` is set.

```
export MSGVERB=severity:text:action
```

```
fmtmsg(MM_PRINT, "XSI:cat", MM_ERROR, "illegal option",  
"refer to cat in user's reference manual", "XSI:cat:001")
```

produces a complete message in the specified message format:

```
ERROR: illegal option  
TO FIX: refer to cat in user's reference manual
```

### Related Information

- “`fprintf()` - `printf()` - `sprintf()` — Format and Write Data” on page 436

## fnmatch() — Match Filename or Pathname

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	XPG4.2	both

### Format

```
#define _XOPEN_SOURCE
#include <fnmatch.h>
```

```
int fnmatch(const char *pattern, const char *string,
            int flags);
```

### General Description

The `fnmatch()` function matches patterns as described in the *OS/390 C/C++ IBM Open Class Library User's Guide* Section 2.13.1, **Patterns Matching a Single Character**, and Section 2.13.2, **Patterns Matching Multiple Characters**. It checks the string specified by the *string* argument to see if it matches the pattern specified by the *pattern* argument.

The *flags* argument modifies the interpretation of *pattern* and *string*. It is the bitwise inclusive OR of zero or more of the flags defined in the header `<fnmatch.h>`. If the `FNМ_PATHNAME` flag is set in *flags*, then a slash character in *string* will be explicitly matched by a slash in *pattern*; it will not be matched by either the asterisk or question-mark special characters, nor by a bracket expression. If `FNМ_PATHNAME` is set and either of these characters would match a slash, the function returns `FNМ_ESLASH`. If the `FNМ_PATHNAME` flag is not set, the slash character is treated as an ordinary character.

If `FNМ_NOESCAPE` is not set in *flags*, a backslash character (`\`) in *pattern* followed by any other character will match that second character in *string*. In particular, `\\` will match a backslash in *string*. If `FNМ_NOESCAPE` is set, a backslash character will be treated as an ordinary character.

If `FNМ_PERIOD` is set in *flags*, then a leading period in *string* will match a period in *pattern*; as described by rule 2 in the XCU specification, Section 2.13.3, Patterns Used for Filename Expansion where the location of “leading” is indicated by the value of `FNМ_PATHNAME`:

- If `FNМ_PATHNAME` is set, a period is “leading” if it is the first character in *string* or if it immediately follows a slash.
- If `FNМ_PATHNAME` is not set, a period is “leading” only if it is the first character of *string*.

If `FNМ_PERIOD` is not set, then no special restrictions are placed on matching a period. If `FNМ_PERIOD` is set, and a pattern wildcard would match a leading period as defined by the above rules, then the function returns `FNМ_EPERIOD`.

**Returned Value**

If *string* matches the pattern specified by *pattern*, then `fnmatch()` returns 0. If there is no match, `fnmatch()` returns `FNM_NOMATCH`, which is defined in the header `<fnmatch.h>`. If an error occurs, `fnmatch()` returns another nonzero value. See the discussion above for the various possible nonzero returns.

**Related Information**

- “`fnmatch.h`” on page 29
- “`glob()` — Generate Pathnames Matching a Pattern” on page 636
- “`wordexp()` — Perform Shell Word Expansions” on page 1774

fopen() — Open a File

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

Format

```
#include <stdio.h>
```

```
FILE *fopen(const char *filename, const char *mode);
```

General Description

Opens the file specified by *filename* and associates a stream with it. The *mode* variable is a character string specifying the type of access requested for the file. The *mode* variable contains one positional parameter followed by optional keyword parameters. The positional parameters are described in Table 21 and Table 22 on page 419.

The positional parameters must be passed as lowercase characters.

The keyword parameters can be passed in mixed case. They must be separated by commas. Only one instance of a keyword can be specified.

The file name passed to `fopen()` often determines the type of file that is opened. A set of file-naming rules exist, which allow you to create an application that references both MVS and HFS files specifically. For details on how `fopen()` determines the type of file from the *filename* and *mode* strings, see one of the “Opening Files” sections in the *OS/390 C/C++ Programming Guide*.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

File mode

Table 21 (Page 1 of 2). Values for the Positional Parameter

File Mode	General Description
r	Open a text file for reading. (The file must exist.)
w	Open a text file for writing. If the w mode is specified for a ddname that has DISP=MOD, the behavior is the same as if a had been specified. Otherwise, if the file already exists, its contents are destroyed.
a	Open a text file in append mode for writing at the end of the file. <code>fopen()</code> creates the file if it does not exist.
r+	Open a text file for both reading and writing. (The file must exist.)
w+	Open a text file for both reading and writing. If the w+ mode is specified for a ddname that has DISP=MOD, the behavior is the same as if a+ had been specified. Otherwise, if the file already exists, its contents are destroyed.

Table 21 (Page 2 of 2). Values for the Positional Parameter

File Mode	General Description
a+	Open a text file in append mode for reading or updating at the end of the file. <code>fopen()</code> creates the file if it does not exist.
rb	Open a binary file for reading. (The file must exist.)
wb	Open an empty binary file for writing. If the <code>wb</code> mode is specified for a <code>ddname</code> that has <code>DISP=MOD</code> , the behavior is the same as if <code>ab</code> had been specified. Otherwise, if the file already exists, its contents are destroyed.
ab	Open a binary file in append mode for writing at the end of the file. <code>fopen()</code> creates the file if it does not exist.
rt	Open a text file for reading. (The file must exist.)
wt	Open a text file for writing. If the file already exists, its contents are destroyed.
at	Open a text file in append mode for writing at the end of the file. <code>fopen()</code> creates the file if it does not exist.
r+b or rb+	Open a binary file for both reading and writing. (The file must exist.)
w+b or wb+	Open an empty binary file for both reading and writing. If the <code>w+b</code> (or <code>wb+</code> ) mode is specified for a <code>ddname</code> that has <code>DISP=MOD</code> , the behavior is the same as if <code>ab+</code> had been specified. Otherwise, if the file already exists, its contents are destroyed.
a+b or ab+	Open a binary file in append mode for writing at the end of the file. <code>fopen()</code> creates the file if it does not exist.
r+t or rt+	Open a text file for both reading and writing. (The file must exist.)
w+t or wt+	Open a text file for both reading and writing. If the file already exists, its contents are destroyed.
a+t or at+	Open a text file in append mode for writing at the end of the file. <code>fopen()</code> creates the file if it does not exist.

**Attention:** Use the `w`, `w+`, `wb`, `w+b`, and `wb+` parameters with care; data in existing files of the same name will be lost.

*Text files* contain printable characters and control characters organized into lines. Each line ends with a new-line character. The system may insert or convert control characters in an output text stream. For example, `\r` written to an MVS DASD text file will be treated as if `\n` (new-line) was written.

**Note:** When compared, data output to a text stream may not be equal to data input from the same text stream.

*Binary files* contain a series of characters. For binary files, the system does not translate control characters on input or output. Under OS/390 C/C++, some types of files are always treated as binary files, even when opened in text mode.

In such cases, a control character is written to the file as binary data. On input, the control character will be read back as it was written. See “The Byte Stream Model” in the *OS/390 C/C++ Programming Guide* for more information.

OS/390 C/C++ has a *Record I/O file* extension. These files are binary in nature—no data interpretation—and require the additional qualifier: `type=record`. See “Writing to Record I/O Files” in the *OS/390 C/C++ Programming Guide* for more information.



When you open a file with a, a+, ab, a+b, or ab+ mode, all write operations take place at the end of the file. Although you can reposition the file pointer using fseek(), fsetpos(), or rewind(), the write functions move the file pointer back to the end of the file before they carry out any output operation. This action prevents you from overwriting existing data.

When you specify the update mode (using + in the second or third position), you can both read from and write to the file. However, when switching between reading and writing, you must include an intervening positioning function such as fseek(), fsetpos(), rewind(), or fflush(). Output may immediately follow input if the EOF was detected.

Table 22 (Page 1 of 2). Keyword Parameters for File Mode

Parameter	Description
acc=value	Indicator of the direction of the access of the VSAM data set. Value can be fwd or bwd.
acc=bwd	Sets the file position indicator to the last record. The access direction may be changed by a call to flocate().
blksize=value	Specifies the maximum length, in bytes, of a physical block of records. To check whether your blksize parameter is valid and is within its limits, see the appropriate section in the <i>OS/390 C/C++ Programming Guide</i> for the type of file you are opening.
bytesseek	Indicator to allow byte seeks for a binary file. For more information, see the ftell() and fseek() functions.
lrec1=value	Specifies the length, in bytes, for fixed-length records and the maximum length for variable-length records. To check whether your lrec1 parameter is valid and is within its limits, see the appropriate section in the <i>OS/390 C/C++ Programming Guide</i> for the type of file you are opening.
recfm=A	ASA print-control characters
recfm=F	Fixed-length, unblocked
recfm=FA	Fixed-length, ASA print-control characters
recfm=FB	Fixed-length, blocked
recfm=FM	Fixed-length, machine print-control codes
recfm=FS	Fixed-length, unblocked, standard
recfm=FBA	Fixed-length, blocked, ASA print-control characters
recfm=FBM	Fixed-length, blocked, machine print-control codes
recfm=FBS	Fixed-length, unblocked, standard ASA print-control characters
recfm=FSA	Fixed-length, unblocked, standard, ASA print-control characters
recfm=FSM	Fixed-length, unblocked, standard, machine print-control codes
recfm=FBSA	Fixed-length, blocked, standard, ASA print-control characters
recfm=FBSM	Fixed-length, blocked, standard, machine print-control codes
recfm=U	Undefined-length
recfm=UA	Undefined-length, ASA print control characters
recfm=UM	Undefined-length, machine print control codes
recfm=V	Variable, unblocked
recfm=VA	Variable, ASA print-control characters

Table 22 (Page 2 of 2). Keyword Parameters for File Mode

Parameter	Description
recfm=VB	Variable, blocked
recfm=VM	Variable, machine print-control codes
recfm=VS	Variable, unblocked, spanned
recfm=VBA	Variable, blocked, ASA print-control characters
recfm=VBM	Variable, blocked, machine print-control codes
recfm=VBS	Variable, blocked, spanned
recfm=VSA	Variable, unblocked, spanned, ASA print-control characters
recfm=VSM	Variable, unblocked, spanned, machine print-control codes
recfm=VBSA	Variable, blocked, spanned, ASA print-control characters
recfm=VBSM	Variable, blocked, spanned, machine print-control codes
recfm=*	Existing file attributes are used if file is opened in write mode. <b>Note:</b> Using recfm=* is only valid for existing DASD data sets. It is ignored in all other cases.
rls=value	Indicates the VSAM RLS (Record Level Sharing) access mode in which a VSAM file is to be opened. This keyword is ignored for non-VSAM files. The following values are valid: <ul style="list-style-type: none"> <li>nri - No Read Integrity</li> <li>cr - Consistent Read</li> </ul> Refer to the <i>OS/390 C/C++ Programming Guide</i> and <i>DFSMS/MVS Using Data Sets</i> for a description of these VSAM RLS access modes.
space	Space attributes for MVS data sets. Within the parameter, you cannot have any imbedded blanks.
type=memory	This parameter identifies this file as a memory file that is accessible only from C programs.
type=memory(hiperspace)	If you are using MVS/ESA, you can specify the HIPERSPACE suboption to open a hiperspace memory file.
type=record	This parameter specifies that the file is to be opened for sequential record I/O. The file must be opened as a binary file; otherwise, fopen() fails. Read and write operations are done with the fread() and fwrite() functions. This is the default fopen() mode for accessing VSAM clusters.
asis	Indicates that the file name is not to be converted to uppercase but used as is. This option is the default under POSIX. It is also the default for HFS file names (see <i>OS/390 C/C++ Programming Guide</i> for details).
password=xxxxxxx	Specifies the password for a VSAM data set.
noseek	Indicates that the stream may not use any of the reposition functions. This may improve performance.

## Returned Value

Returns a pointer to the object controlling the associated stream. A NULL pointer returned value indicates an error.

The `fopen()` function generally fails if parameters are mismatched.

## Example CBC3BF26

```
/* CBC3BF26
   This example attempts to open two files for reading, myfile.dat and
   myfile2.dat.
*/
#include <stdio.h>

int main(void)
{
    FILE *stream;

    /* The following call opens a text file for reading */

    if ((stream = fopen("myfile.dat", "r")) == NULL)
        printf("Could not open data file for reading\n");

    /* The following call opens: the file myfile2.dat,
       a binary file for reading and writing, whose record length is 80 bytes,
       and maximum length of a physical block is 240 bytes,
       fixed-length, blocked record format for sequential record I/O. */

    if ( (stream = fopen("myfile2.dat", "rb+", lrecl=80,\
        blksize=240, recfm=fb, type=record)) == NULL )
        printf("Could not open data file for read update\n");
}
```

## Related Information

- Various chapters dealing with I/O, in the *OS/390 C/C++ Programming Guide*.
- “stdio.h” on page 43
- “fclose() — Close File” on page 348
- “fldata() — Retrieve File Information” on page 402
- “freopen() — Redirect an Open File” on page 460

## fork() — Create a New Process

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <sys/types.h>
#include <unistd.h>
```

```
pid_t fork(void);
```

**Note:** Although POSIX.1 does not require that the `<sys/types.h>` include file be included, XPG4 has it as an optional header. Therefore, it is recommended that you include it for portability.

### General Description

Creates a new process. The new process (the *child process*) is an exact duplicate of the process that calls `fork()` (the *parent process*), except for the following:

- The child process has a unique process ID (PID) that does not match any active process group ID.
- The child has a different parent process ID, that is, the process ID of the process that called `fork()`.
- The child has its own copy of the parent's file descriptors. Each file descriptor in the child refers to the same open file description as the corresponding file descriptor in the parent.
- The child has its own copy of the parent's open directory streams. Each child's open directory stream can share directory stream positioning with the corresponding parent's directory stream.
- The following elements in the `tms` structure are set to 0 in the child:

```
tms_utime
tms_stime
tms_cutime
tms_cstime
```

For more information about these elements, see “`times()` — Get Process and Child Process Times” on page 1572.

- The child does not inherit any file locks previously set by the parent.
- The child process has no alarms set (similar to the results of a call to `alarm()` with an argument value of 0).
- The child has no pending signals.

In all other respects, the child is identical to the parent. Because the child is a duplicate, it contains the same call to `fork()` that was in the parent. Execution begins with this `fork()` call, which returns a value of 0; the child then proceeds with normal execution.

The child address space inherits the following address space attributes of the parent address space:

- Region size
- Time limit

For more information on `fork()` from an MVS perspective, refer to *OS/390 UNIX System Services Programming: Assembler Callable Services Reference*.

You can use MVS memory files from an OS/390 UNIX program. However, use of the `fork()` function from the program removes access from a hiperspace memory file for the child process. Use of an `exec` function from the program clears a memory file when the process address space is cleared.

### Interoperability Restriction

For POSIX resources, `fork()` behaves as just described. But in general, MVS resources that existed in the parent do *not* exist in the child. This is true for open streams in MVS data sets and assembler-accessed MVS facilities, such as STIMERS. In addition, MVS allocations (through JCL, SVC99, or ALLOCATE) are not passed to the child process.

### Special Behavior for OS/390 UNIX Services

#### Notes:

1. A prior loaded copy of an HFS program in the same address space is reused under the same circumstances that apply to the reuse of a prior loaded MVS unauthorized program from an unauthorized library by the MVS XCTL service with the following exceptions:
  - If the calling process is in Ptrace debug mode, a prior loaded copy is not reused.
  - If the calling process is not in Ptrace debug mode, but the only prior loaded usable copy found of the HFS program is in storage modifiable by the caller, the prior copy is not reused.
2. If the specified file name represents an external link or a sticky bit file, the program is loaded from the caller's MVS load library search order. For an external link, the external name is only used if the name is eight characters or less, otherwise the caller receives an error from the loadhfs service. For a sticky bit program, the file name is used if it is eight characters or less. Otherwise, the program is loaded from the HFS.
3. If the calling task is in a WLM enclave, the resulting task in the new process image is joined to the same WLM enclave. This allows WLM to manage the old and new process images as one 'business unit of work' entity for system accounting and management purposes.

### Returned Value

If successful, `fork()` returns 0 to the child process and the process ID of the newly created child to the parent process. If unsuccessful, `fork()` fails to create a child process and returns -1 to the parent. `fork()` then sets `errno` to one of the following:

- EAGAIN**     There are insufficient resources to create another process, or the process has already reached the maximum number of processes you can run.
- ELEMSGERR**  
              LE message file not available.
- ELENOFORK**  
              Application contains a language that does not support fork().
- ENOMEM**     The process requires more space than is available.

### Example

#### CBC3BF27

```

/* CBC3BF27
   This example creates a new child process.
*/
#define _POSIX_SOURCE
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

main() {
    pid_t pid;
    int status;

    if ((pid = fork()) < 0)
        perror("fork() error");
    else if (pid == 0) {
        puts("This is the child.");
        printf("Child's pid is %d and my parent's is %d\n",
            (int) getpid(), (int) getppid());
        exit(42);
    }
    else {
        puts("This is the parent.");
        printf("Parent's pid is %d and my child's is %d\n",
            (int) getpid(), (int) pid);
        puts("I'm waiting for my child to complete.");
        if (wait(&status) == -1)
            perror("wait() error");
        else if (WIFEXITED(status))
            printf("The child exited with status of %d\n",
                WEXITSTATUS(status));
        else
            puts("The child did not exit successfully");
    }
}

```

### Output

```

This is the parent.
This is the child.
Child's pid is 1114120 and my parent's is 2293766
Parent's pid is 2293766 and my child's is 1114120
I'm waiting for my child to complete.
The child exited with status of 42

```

## Related Information

- “sys/types.h” on page 49
- “alarm() — Set an Alarm” on page 102
- “exec Functions” on page 322
- “fcntl() — Control Open File Descriptors” on page 350
- “getrlimit() — Control Maximum Resource Consumption” on page 591
- “kill() — Send a Signal to a Process” on page 728
- “nice() — Change Priority of a Process” on page 864
- “putenv() — Change or Add an Environment Variable” on page 1054
- “semop() — Semaphore Operations” on page 1175
- “shmat() — Shared Memory Attach Operation” on page 1285
- “sysconf() — Determine System Configuration Options” on page 1483
- “times() — Get Process and Child Process Times” on page 1572
- “ulimit() — Get/Set Process File Size Limits” on page 1645
- “wait() — Wait for a Child Process to End” on page 1687

## fortrc() — Return FORTRAN Return Code

### Standards

Standards / Extensions	C or C++	Dependencies
C Library	both	

### Format

```
#include <stdlib.h>
```

```
int fortrc(void);
```

### External Entry Point

```
@@FORTRC, __fortrc
```

### General Description

The `fortrc()` function returns the value specified on the Fortran RETURN statement issued by the last Fortran routine called from the C program.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use `LANGVLV(EXTENDED)`.

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with `LANGVLV(EXTENDED)`. When you use `LANGVLV(EXTENDED)` any relevant information in the header is also exposed.

The Fortran routine called must be identified to C as a Fortran routine using the following preprocessor directive:

```
#pragma
linkage(identifier,FORTRAN,RETURNCODE).
```

The function `fortrc()` should be called immediately after a call to the FORTRAN routine *identifier* or else results are unpredictable.

If you do not include `stdlib.h` in your source code or you use the compile-time option `LANGVLV(ANSI)`, then you must use `__fortrc` to call the function.

### Related Information

- “`stdlib.h`” on page 45



## fpathconf() — Determine Configurable Path Name Variables

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
long fpathconf(int fildes, int varcode);
```

### General Description

Determines the value of a configuration variable (*varcode*) associated with a particular file descriptor (*fildes*).

fpathconf() works exactly like pathconf(), except that it takes a file descriptor as an argument rather than taking a path name.

The *varcode* argument can be any one of a set of symbols defined in the unistd.h header file. Each symbol stands for a configuration variable. These are the possible symbols:

#### **\_PC\_LINK\_MAX**

Represents LINK\_MAX, the maximum number of links the file can have. If *pathname* is a directory, fpathconf() returns the maximum number of links that can be established to the directory itself.

#### **\_PC\_MAX\_CANON**

Represents MAX\_CANON, the maximum number of bytes in a terminal canonical input line. *pathname* must refer to a character special file for a terminal.

#### **\_PC\_MAX\_INPUT**

Represents MAX\_INPUT, the minimum number of bytes for which space will be available in a terminal input queue. This input space is the maximum number of bytes that a portable application will allow an end user to enter before the application actually reads the input. *pathname* must refer to a character special file for a terminal.

#### **\_PC\_NAME\_MAX**

Represents NAME\_MAX, the maximum number of characters in a file name (not including any terminating null character if the file name is stored as a string). This limit refers only to the file name itself, that is, the last component of the file's path name. fpathconf() returns the maximum length of file names.

#### **\_PC\_PATH\_MAX**

Represents PATH\_MAX, the maximum number of characters in a complete path name (not including any terminating null if the path name is stored as a string). fpathconf() returns the maximum length of a relative path name.

**\_PC\_PIPE\_BUF**

Represents PIPE\_BUF, the maximum number of bytes that can be written to a pipe as one unit. If more than this number of bytes is written to a pipe, the operation can take more than one physical write operation and can require more than one physical read operation to read the data on the other end of the pipe. If *pathname* is a FIFO special file, fpathconf() returns the value for the file itself. If *pathname* is a directory, fpathconf() returns the value for any FIFOs that exist or can be created under the directory. If *pathname* is any other kind of file, an errno of EINVAL (see description below) will be returned.

**\_PC\_CHOWN\_RESTRICTED**

Represents \_POSIX\_CHOWN\_RESTRICTED defined in the unistd.h header file. This symbol indicates that the use of chown() is restricted; see the callable service chown() for more details. If *pathname* is a directory, fpathconf() returns the value for any kind of file under the directory, but not for subdirectories of the directory.

**\_PC\_NO\_TRUNC**

Represents \_POSIX\_NO\_TRUNC defined in the unistd.h header file. This symbol indicates that an error should be generated if a file name is longer than NAME\_MAX. If *pathname* refers to a directory, the value returned by fpathconf() applies to all files under that directory.

**\_PC\_VDISABLE**

Represents \_POSIX\_VDISABLE defined in the unistd.h header file. This symbol indicates that terminal special characters can be disabled using this character value, if it is defined. See the callable service tcsetattr() for details. *pathname* must refer to a character special file for a terminal.

**Returned Value**

If a particular variable has no limit, fpathconf() returns -1 but does not change errno.

If successful, fpathconf() returns the value of the variable requested in *varcode*. If unsuccessful, it returns -1 and sets errno to one of the following:

- |        |   |
|--------|---|
| EBADF  | <i>fildev</i> is not a valid open file descriptor.  |
| EINVAL | <p><i>varcode</i> is not a valid variable code, or the given variable cannot be associated with the specified file.</p> <ul style="list-style-type: none"> <li>• If <i>varcode</i> refers to MAX_CANON, MAX_INPUT, or _POSIX_VDISABLE, and <i>pathname</i> does not refer to a character special file, fpathconf() returns -1 and sets errno to EINVAL.</li> <li>• If <i>varcode</i> refers to NAME_MAX, PATH_MAX, or POSIX_NO_TRUNC, and <i>pathname</i> does not refer to a directory, fpathconf() returns the requested information.</li> <li>• If <i>varcode</i> refers to PC_PIPE_BUF and <i>pathname</i> refers to a pipe or a FIFO, the value returned applies to the referenced object itself. If <i>pathname</i> refers to a directory, the value returned applies to any FIFOs that exist or can be created within the directory. If</li> </ul> |

*pathname* refers to any other type of file, the function sets `errno` to `EINVAL`.

### Example CBC3BF29

```
/* CBC3BF29
   This example uses fpathconf() with __PC_NAME_MAX to determine the value
   of the NAME_MAX configuration variable.
*/
#define _POSIX_SOURCE
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

main() {
    long result;
    char fn[]="temp.file";
    int fd;

    if ((fd = creat(fn, S_IRUSR)) < 0)
        perror("creat() error");
    else {
        errno = 0;
        puts("examining NAME_MAX limit for current working directory's");
        puts("file system:");
        if ((result = fpathconf(fd, _PC_NAME_MAX)) == -1)
            if (errno == 0)
                puts("There is no limit to NAME_MAX.");
            else perror("fpathconf() error");
        else
            printf("NAME_MAX is %ld\n", result);
        close(fd);
        unlink(fn);
    }
}
```

### Output

```
examining NAME_MAX limit for current working directory's
file system:
NAME_MAX is 255
```

### Related Information

- “unistd.h” on page 53
- “open() — Open a File” on page 872
- “pathconf() — Determine Configurable Path Name Variables” on page 896

## fp\_clr\_flag() — Reset Floating-Point Exception Status Flag

### Standards

Standards / Extensions	C or C++	Dependencies
	both	OS/390 V2R6

### Format

```
#include <float.h>
#include <fpxcp.h>
```

```
void fp_clr_flag(mask)
fp_flag_t mask;
```

### General Description

The `fp_clr_flag()` function resets the exception status flags defined by the *mask* parameter to 0 (false). The remaining flags in the exception status remain unchanged.

The **fpxcp.h** file defines the following names for the flags indicating floating-point exception status:

<i>FP_INVALID</i>	Invalid operation summary
<i>FP_OVERFLOW</i>	Overflow
<i>FP_UNDERFLOW</i>	Underflow
<i>FP_DIV_BY_ZERO</i>	Division by 0
<i>FP_INEXACT</i>	Inexact result

Users can reset multiple exception flags using the `fp_clr_flag()` function by ORing the names of individual flags. For example, the following resets both the overflow and inexact flags.

```
fp_clr_flag(FP_OVERFLOW | FP_INEXACT)
```

### Returned Value

None

### Related Information

- “`fp_raise_xcp()` — Raise a Floating-Point Exception” on page 431
- “`fp_read_flag()` — Return the Current Floating-Point Exception Status” on page 433
- “`__isBFP()` — Determine Application Floating-Point Format” on page 705

## fp\_raise\_xcp() — Raise a Floating-Point Exception

### Standards

Standards / Extensions	C or C++	Dependencies
	both	OS/390 V2R6

### Format

```
#include <fpxcp.h>
```

```
int fp_raise_xcp(mask)
fpflag_t mask;
```

### General Description

The `fp_raise_xcp()` function causes floating-point exceptions defined by the *mask* parameter to be raised immediately.

If the exceptions defined by the *mask* parameter are enabled and the program is running in serial mode, the signal for floating-point exceptions, SIGFPE, is raised.

The **fpxcp.h** file defines the following names for the flags indicating floating-point exception status:

<i>FP_INVALID</i>	Invalid operation summary
<i>FP_OVERFLOW</i>	Overflow
<i>FP_UNDERFLOW</i>	Underflow
<i>FP_DIV_BY_ZERO</i>	Division by 0
<i>FP_INEXACT</i>	Inexact result

Users can cause multiple exceptions using `fp_raise_xcp()` by ORing the names of individual flags. For example, the following causes both overflow and division by 0 exceptions to occur.

```
fp_raise_xcp(FP_OVERFLOW | FP_DIV_BY_ZERO)
```

If more than one exception is included in the mask variable, the exceptions are raised in the following order:

1. Invalid operation
2. Division by zero
3. Underflow
4. Overflow
5. Inexact result

Thus, if the user exception handler does not disable further exceptions, one call to the `fp_raise_xcp()` function can cause the exception handler to be entered many times.

### **Returned Value**

The `fp_raise_xcp()` function returns 0 for normal completion and returns a nonzero value if an error occurs.

### **Related Information**

- “`fp_clr_flag()` — Reset Floating-Point Exception Status Flag” on page 430
- “`fp_read_flag()` — Return the Current Floating-Point Exception Status” on page 433
- “`__isBFP()` — Determine Application Floating-Point Format” on page 705

## fp\_read\_flag() — Return the Current Floating-Point Exception Status

### Standards

Standards / Extensions	C or C++	Dependencies
	both	OS/390 V2R6

### Format

```
#include <float.h>
#include <fpxcp.h>
```

```
fp_flag_t fp_read_flag()
```

### General Description

The `fp_read_flag()` function returns the current floating-point exception status.

These functions aid in determining both when an exception has occurred and the exception type. These functions can be called explicitly around blocks of code that may cause a floating-point exception.

According to the IEEE Standard for Binary Floating-Point Arithmetic, the following types of floating-point operations must be signaled when detected in a floating-point operation:

- Invalid operation
- Division by zero
- Overflow
- Underflow
- Inexact

An invalid operation occurs when the result cannot be represented (for example, a square root operation on a number less than 0).

The IEEE Standard for Binary Floating-Point Arithmetic states: “For each type of exception, the implementation shall provide a status flag that shall be set on any occurrence of the corresponding exception when no corresponding trap occurs. It shall be reset only at the user's request. The user shall be able to test and to alter the status flags individually, and should further be able to save and restore all five at one time.”

Floating-point operations can set flags in the floating-point exception status but cannot clear them. Users can clear a flag in the floating-point exception status using an explicit software action such as the `fp_clr_flag(0)` subroutine.

The **fpxcp.h** file defines the following names for the flags indicating floating-point exception status:

<i>FP_INVALID</i>	Invalid operation summary
<i>FP_OVERFLOW</i>	Overflow
<i>FP_UNDERFLOW</i>	Underflow
<i>FP_DIV_BY_ZERO</i>	Division by 0
<i>FP_INEXACT</i>	Inexact result

**Returned Value**

The `fp_read_flag()` function returns the current floating-point exception status. The flags in the returned exception status can be tested using the flag definitions above. You can test individual flags or sets of flags.

**Related Information**

- IEEE Standard for Binary Floating-Point Arithmetic (ANSI/IEEE Standards 754-1985 and 854-1987).
- “`fp_clr_flag()` — Reset Floating-Point Exception Status Flag” on page 430
- “`fp_raise_xcp()` — Raise a Floating-Point Exception” on page 431
- “`__isBFP()` — Determine Application Floating-Point Format” on page 705



## fp\_read\_rnd() — Determine Rounding Mode

### Standards

Standards / Extensions	C or C++	Dependencies
	both	OS/390 V2R6

### Format

```
#define _AIX_COMPATIBILITY 1
#include <float.h>
```

```
fprnd_t fp_read_rnd;
```

### General Description

For an application running in IEEE binary floating-point (IEEE floating-point) mode, the `fp_read_rnd()` function returns the current rounding mode indicated by the rounding mode field of the floating-point control (FPC) register. For an application running in S/390 hexadecimal floating-point (hexadecimal floating-point) mode, `fp_read_rnd()` returns 0.

### Returned Value

For an application running in IEEE floating-point mode, `fp_read_rnd` returns the following:

Value	Rounding Mode
<code>_FP_RND_RZ</code>	Round toward 0
<code>_FP_RND_RN</code>	Round to nearest
<code>_FP_RND_RP</code>	Round toward +infinity
<code>_FP_RND_RM</code>	Round toward -infinity

For an application running in hexadecimal floating-point mode, `fp_read_rnd` returns 0.

### Related Information

- “`fp_swap_rnd()` — Swap Rounding Mode” on page 446
- “`__isBFP()` — Determine Application Floating-Point Format” on page 705

## fprintf() - printf() - sprintf() — Format and Write Data

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
int fprintf(FILE *stream, const char *format-string, ...);
int printf(const char *format-string, ...);
int sprintf(char *buffer, const char *format-string, ...);
```

### General Description

These three related functions are referred to as the *fprintf family*.

The fprintf() function formats and writes output to a *stream*. It converts each entry in the *argument list*, if any, and writes to the stream according to the corresponding format specification in the *format-string*. The fprintf() function cannot be used with a file that is opened using type=record.

The printf() function formats and writes output to the standard output stream stdout. printf() cannot be used if stdout has been reopened using type=record.

The sprintf() function formats and stores a series of characters and values in the array pointed to by *buffer*. Any *argument-list* is converted and put out according to the corresponding format specification in the *format-string*. If the strings pointed to by *buffer* and *format* overlap, behavior is undefined.

To provide an ASCII input/output format for applications using the sprintf() function, define feature test macro \_\_LIBASCII as described on page 22.

fprintf() and printf() have the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

The *format-string* consists of ordinary characters, escape sequences, and conversion specifications. The ordinary characters are copied in order of their appearance. Conversion specifications, beginning with a percent sign (%) or the sequence (%n\$) where n is a decimal integer in the range [1,NL\_ARGMAX], determine the output format for any *argument-list* following the *format-string*. The *format-string* can contain multibyte characters beginning and ending in the initial shift state.

### Special Behavior for XPG4

- If the %n\$ conversion specification is found, the value of the nth *argument* after the *format-string* is converted and output according to the conversion specification. Numbered arguments in the argument list can be referenced from *format-string* as many times as required.

- The *format-string* can contain either form of the conversion specification, that is, % or %n\$ but the two forms cannot be mixed within a single *format-string* except that %% can be mixed with the %n\$ form. When numbered conversion specifications are used, specifying the 'nth' argument requires that the first to (n-1)th arguments are specified in the *format-string*.

The *format-string* is read from left to right. When the first format specification is found, the value of the first *argument* after the *format-string* is converted and output according to the format specification. The second format specification causes the second *argument* after the *format-string* to be converted and output, and so on through the end of the *format-string*. If there are more arguments than there are format specifications, the extra arguments are evaluated and ignored. The results are undefined if there are not enough arguments for all the format specifications. The format specification is illustrated below.

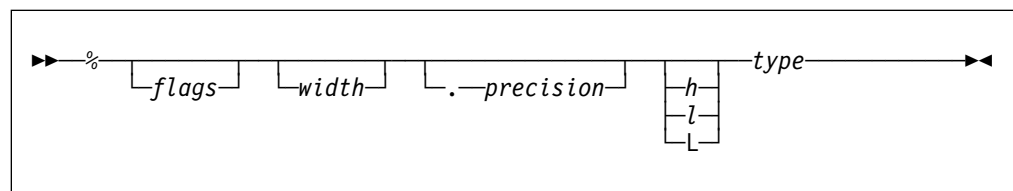


Figure 5. Format Specification for fprintf(), printf(), and sprintf()

Each field of the format specification is a single character or number signifying a particular format option. The *type* character, which appears after the last optional format field, determines whether the associated argument is interpreted as a character, a string, a number, or pointer. The simplest format specification contains only the percent sign and a *type* character (for example, %s).

### The percent sign

If a percent sign (%) is followed by a character that has no meaning as a format field, the character is simply copied to stdout. For example, to print a percent sign character, use %%.

### The flag characters

The *flag* characters in Table 23 are used for the justification of output and printing of thousands' grouping characters, signs, blanks, decimal points, octal, and hexadecimal prefixes, and the semantics for wchar\_t precision unit. Notice that more than one *flag* can appear in a format specification. This is an optional field.

Table 23 (Page 1 of 2). Flag Characters for fprintf() Family

Flag	Meaning	Default
'	<b>Added for XPG4:</b> The integer portion of the result of a decimal conversion(%i,%d,%u, %f,%g or %G) will be formatted with the thousands' grouping characters.	No grouping.
-	Left-justify the result within the field width.	Right-justify.
+	Prefix the output value with a sign (+ or -) if the output value is of a signed type.	Sign appears only for negative signed values (-).

Table 23 (Page 2 of 2). Flag Characters for fprintf() Family

Flag	Meaning	Default
<i>blank</i> (' ')	Prefix the output value with a blank if the output value is signed and positive. The + flag overrides the <i>blank</i> flag if both appear, and a positive signed value will be output with a sign.	No blank.
#	When used with the o, x, or X formats, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively.	No prefix.
	When used with the f, e, or E formats, the # flag forces the output value to contain a decimal point in all cases.	Decimal point appears only if digits follow it.
	The decimal point is sensitive to the LC_NUMERIC category of the same current locale.	
	When used with the g or G formats, the # flag forces the output value to contain a decimal point in all cases and prevents the truncation of trailing zeros.	Decimal point appears only if digits follow it; trailing zeros are truncated.
	When used with the l or L format, the # flag causes precision to be measured in wide characters.	Precision indicates the maximum number of bytes to be output.
0	When used with the d, i, o, u, x, X, e, E, f, g, or G formats, the 0 flag causes leading 0's to pad the output to the field width. The 0 flag is ignored if precision is specified for an integer or if the - flag is specified.	Space padding.

The code point for the # character varies between the EBCDIC encoded character sets. The definition of the # character is based on the current LC\_SYNTAX category. The default C locale expects the # character to use the code point for encoded character set IBM-1047.

When the LC\_SYNTAX category is set using setlocale(), the format strings passed to the printf() functions must use the same encoded character set as is specified for the LC\_SYNTAX category.

The # flag should not be used with c, lc, C, d, i, u, s, or p types.

### The Width of the Output

*Width* is a non-negative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the specified *width*, blanks are added on the left or the right (depending on whether the - flag is specified) until the minimum width is reached.

*Width* never causes a value to be truncated; if the number of characters in the output value is greater than the specified *width*, or *width* is not given, all characters of the value are output (subject to the *precision* specification).

The *width* specification can be an asterisk (\*); if it is, an argument from the argument list supplies the value. The *width* argument must precede the value being formatted in the argument list. This is an optional field.

If *format-string* contains the %n\$ form of conversion specification, *width* can be indicated by the sequence \*m\$, where m is a decimal integer in the range [1,NL\_ARGMAX] giving the position of an integer argument in the argument list containing the field width.

### The Precision of the Output

*precision* is a non-negative decimal integer preceded by a period. It specifies the number of characters to be output, or the number of decimal places. Unlike the *width* specification, the *precision* can cause truncation of the output value or rounding of a floating-point value.

The *precision* specification can be an asterisk (\*); if it is, an argument from the argument list supplies the value. The *precision* argument must precede the value being formatted in the argument list. The *precision* field is optional.

If *format-string* contains the %n\$ form of conversion specification, *precision* can be indicated by the sequence \*m\$, where m is a decimal integer in the range [1,NL\_ARGMAX] giving the position of an integer argument in the argument list containing the field precision.

The interpretation of the *precision* value and the default when the *precision* is omitted depend upon the *type*, as shown in Table 24 on page 440.

Table 24. Precision Argument in fprintf() family

Type	Meaning	Default
i d u o x X	<i>Precision</i> specifies the minimum number of digits to be output. If the number of digits in the argument is less than <i>precision</i> , the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds <i>precision</i> .	If <i>precision</i> is 0 or omitted entirely, or if the period (.) appears without a number following it, the <i>precision</i> is set to 1.
f e E	<i>Precision</i> specifies the number of digits to be output after the decimal point. The last digit output is rounded.  The decimal point is sensitive to the LC_NUMERIC category of the current locale.	Default <i>precision</i> is 6. If <i>precision</i> is 0 or the period appears without a number following it, no decimal point is output.
g G	<i>Precision</i> specifies the maximum number of significant digits output.	All significant digits are output.
c	No effect.	The character is output.
C lc	No effect.	The wide character is output.
s	<i>Precision</i> specifies the maximum number of characters to be output. Characters in excess of <i>precision</i> are not output.	Characters are output until a null character is encountered.
S ls	<i>Precision</i> specifies the maximum number of bytes to be output. Bytes in excess of <i>precision</i> are not output; however, multi-byte integrity is always preserved.	wchar_t characters are output until a null character is encountered.

**Optional prefix**

Used to indicate the size of the argument expected:

h	A prefix with the integer types d, i, o, u, x, X, and n that specifies that the argument is short int or unsigned short int.
l	A prefix with d, i, o, u, x, X, and n types that specifies that the argument is a long int or unsigned long int.  The l prefix with the c type conversion specifier indicates that the argument is a wint_t. The l prefix with the s type conversion specifier indicates that the argument is a pointer to a wchar_t.
L	A prefix with e, E, f, g, or G types that specifies that the argument is long double.

**Note:** If you pass a long double value and do not use the L qualifier or if you pass a double value only and use the L qualifier, errors occur.

Table 25 on page 441 below shows the meaning of the type characters used in the precision argument.

Table 25. Type Characters and their Meanings

Type	Argument	Output Format
d, i	Integer	Signed decimal integer.
u	Integer	Unsigned decimal integer.
o	Integer	Unsigned octal integer.
x	Integer	Unsigned hexadecimal integer, using abcdef.
X	Integer	Unsigned hexadecimal integer, using ABCDEF.
f	Double	<p>Signed value having the form <code>[-]dddd.dddd</code>, where <code>dddd</code> is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number. The number of digits after the decimal point is equal to the requested precision.</p> <p>The decimal point is sensitive to the <code>LC_NUMERIC</code> category of the current locale.</p>
e	Double	Signed value having the form <code>[-]d.ddde[ <i>sig n</i>]ddd</code> , where <code>d</code> is a single-decimal digit, <code>dddd</code> is one or more decimal digits, <code>ddd</code> is 2 or more decimal digits, and <i>sign</i> is <code>+</code> or <code>-</code> .
E	Double	Identical to the <code>e</code> format, except that <code>E</code> introduces the exponent, not <code>e</code> .
g	Double	Signed value output in <code>f</code> or <code>e</code> format. The <code>e</code> format is used only when the exponent of the value is less than <code>-4</code> or greater than or equal to the <i>precision</i> . Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it.
G	Double	Identical to the <code>g</code> format, except that <code>E</code> introduces the exponent (where appropriate), not <code>e</code> .
D(n,p)	Decimal type argument.	Fixed-point value consisting of a series of one or more decimal digits possibly containing a decimal point.
c	Character	Single character.
C or lc	Wide Character	The argument of <code>wchar_t</code> type is converted to an array of bytes representing a multibyte character as if by call to <code>wctomb()</code> .
s	String	Characters output up to the first null character ( <code>\0</code> ) or until <i>precision</i> is reached.
S or ls	Wide String	<p>The argument is a pointer to an array of <code>wchar_t</code> type. Wide characters from the array are converted to multibyte characters up to and including a terminating null wide character. Conversion takes place as if by a call to <code>wcstombs()</code>, with the conversion state described by the <code>mbstate_t</code> object initialized to 0. The result written out will not include the terminating null character.</p> <p>If no precision is specified, the array contains a null wide character. If a precision is specified, it sets the maximum number of characters written, including shift sequences. A partial multibyte character cannot be written.</p>
n	Pointer to integer	Number of characters successfully output so far to the <i>stream</i> or buffer; this value is stored in the integer whose address is given as the argument.
p	Pointer	Pointer to void converted to a sequence of printable characters. Refer to the individual system reference guides for the specific format.

### fprintf Family of Formatted Output Functions

*fprintf* family functions match `e`, `E`, `f`, `g` or `G` conversion specifiers to floating-point arguments for which they produce floating-point number substrings in the output stream. *fprintf* family functions have been extended to determine the floating-point

format, hexadecimal floating-point or IEEE floating-point, of types e, E, f, g or G by using `__isBFP()`.

*fprintf* family functions convert IEEE floating-point infinity and NaN argument values to special infinity and NaN floating-point number output sequences.

- The special output sequence for infinity values is a plus or minus sign, then the character sequence INF followed by a white-space character (space, tab, or new line), a null character (`\0`) or EOF.
- The special output sequence for NaN values is a plus or minus sign, then the character sequence NANS for a signalling NaN or NANQ for a quiet NaN, then a NaN ordinal sequence, and then a white-space character (space, tab, or new line), a null character (`\0`) or EOF.

A NaN ordinal sequence is a left-parenthesis character, “(”, followed by a digit sequence representing an integer *n*, where  $1 \leq n \leq \text{INT\_MAX}-1$ , followed by a right-parenthesis character, “)”. The integer value, *n*, is determined by the fraction bits of the NaN argument value as follows:

1. For a signalling NaN value, NaN fraction bits are reversed (left to right) to produce bits (right to left) of an even integer value,  $2*n$ . Then formatted output functions produce a (signalling) NaN ordinal sequence corresponding to the integer value *n*.
2. For a quiet NaN value, NaN fraction bits are reversed (left to right) to produce bits (right to left) of an odd integer value,  $2*n-1$ . Then formatted output functions produce a (quiet) NaN ordinal sequence corresponding to the integer value *n*.

Some compatibility with NaN sequences output by AIX formatted output functions can be achieved by setting a new environment variable, `_AIX_NAN_COMPATIBILITY`, which OS/390 formatted output functions recognize, to one of the following (string) values:

Value	Output Function
1	Formatted output functions which produce special NaN output sequences omit the NaN ordinal output sequence (1). This results in output NaN sequences of plus or minus sign followed by NANS or NANQ instead of plus or minus sign followed by NANS(1) or NANQ(1). All other NaN ordinal sequences are explicitly output.
ALL	Formatted output functions which produce special NaN output sequences omit the NaN ordinal output sequence for all NaN values. This results in output NaN sequences of plus or minus sign followed by NANS or NANQ instead of plus or minus sign followed by NANS( <i>n</i> ) or NANQ( <i>n</i> ) for all NaN values.

The `sprintf()` function is available to C applications in a stand-alone Systems Programming Environment.

### Returned Value

The `fprintf()`, `printf()`, and `sprintf()` functions return the number of characters output, or a negative value if an output error occurs. The ending null character is not counted.



## Example

### CBC3BF30

```

/* CBC3BF30
   This example prints data using printf() in a variety of formats.
*/
#include <stdio.h>

int main(void)
{
    char ch = 'h', *string = "computer";
    int count = 234, hex = 0x10, oct = 010, dec = 10;
    double fp = 251.7366;
    unsigned int a = 12;
    float b = 123.45;
    int c;
    void *d = "a";

    printf("the unsigned int is %u\n\n",a);

    printf("the float number is %g, and %G\n\n",b,b);

    printf("RAY%n\n\n",&c);

    printf("last line prints %d characters\n\n",c);

    printf("Address of d is %p\n\n",d);

    printf("%d  %d  %06d  %X  %x  %o\n\n",
        count, count, count, count, count, count);

    printf("12345678901234567890123456789\n\n", &count);

    printf("Value of count should be 13; count = %d\n\n", count);

    printf("%10c%5c\n\n", ch, ch);

    printf("%25s\n%25.4s\n\n", string, string);

    printf("%f  %.2f  %e  %E\n\n", fp, fp, fp, fp);

    printf("%i  %i  %i\n\n", hex, oct, dec);
}

```

## Output

```

the unsigned int is 12

the float number is 123.45 and 123.45

RAY

last line prints 3 characters

Address of d is DD72F9

234  +234  000234  EA  ea  352

12345678901234567890123456789

Value of count should be 13; count = 13

    h    h

```

```

                                computer
                                comp
251.736600    251.74    2.517366e+02    2.517366E+02

16    8    10

```

### **CBC3BF31**

```

/* CBC3BF31
   The following example illustrates the use of printf() to print
   fixed-point decimal data types.
   This example works under C only, not C++.
*/
#include <stdio.h>
#include <decimal.h>

decimal(10,2) pd01 = -12.34d;
decimal(12,4) pd02 = 12345678.9876d;
decimal(31,10) pd03 = 123456789013579246801.9876543210d;

int main(void) {
    printf("pd01 %D(10,2)      = %D(10,2)\n", pd01);
    printf("pd02 %D( 12 , 4 ) = %D( 12 , 4 )\n", pd02);

    printf("pd01 %010.2D(10,2) = %010.2D(10,2)\n", pd01);
    printf("pd02 %020.2D(12,4)  = %020.2D(12,4)\n", pd02);
    printf("\n Give strange result if the specified size is wrong!\n");
    printf("pd03 %D(15,3)      = %D(15,3)\n\n", pd03);
}

```

### **Output**

```

pd01 %D(10,2)      = -12.34
pd02 %D( 12 , 4 ) = 12345678.9876
pd01 %010.2D(10,2) = -000012.34
pd02 %020.2D(12,4) =          12345678.98

```

```

Give strange result if the specified size is wrong!
pd03 %D(15,3)      = -123456789013.579

```

### **CBC3BF32**

```

/* CBC3BF32
   This example illustrates the use of sprintf() to format and print
   various data.
*/
#include <stdio.h>

char buffer[200];
int i, j;
double fp;
char *s = "baltimore";
char c;

int main(void)
{
    c = 'l';
    i = 35;
    fp = 1.7320508;

    /* Format and print various data */
    j = sprintf(buffer, "%s\n", s);
    j += sprintf(buffer+j, "%c\n", c);
}

```

```

j += sprintf(buffer+j, "%d\n", i);
j += sprintf(buffer+j, "%f\n", fp);
printf("string:\n%s\ncharacter count = %d\n", buffer, j);
}

```

### Output

```

string:
Baltimore
1
35
1.732051

```

```

character count = 24

```

### Related Information

- “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*
- “System Programming Facilities” in the *OS/390 C/C++ Programming Guide*
- “locale.h” on page 33
- “stdio.h” on page 43
- “wchar.h” on page 54
- “fscanf() – scanf() – sscanf() — Read and Format Data” on page 464
- “\_\_isBFP() — Determine Application Floating-Point Format” on page 705
- “localeconv() — Query Numeric Conventions” on page 756
- “setlocale() — Set Locale” on page 1241
- “wctomb() — Convert a Wide Character to a Multibyte Character” on page 1697.

## fp\_swap\_rnd() — Swap Rounding Mode

### Standards

Standards / Extensions	C or C++	Dependencies
	both	OS/390 V2R6

### Format

```
#define _AIX_COMPATIBILITY 1
#include <float.h>
```

```
fprnd_t fp_swap_rnd(RoundMode)
fprnd_t RoundMode
```

### General Description

For an application running in IEEE floating-point mode, the `fp_swap_rnd()` function returns the current rounding mode specified by the rounding mode field of the floating-point control (FPC) register and sets the rounding mode field in the FPC register based on the value of *RoundMode* as follows:

Value	Rounding Mode
_FP_RND_RZ	Round toward 0
_FP_RND_RN	Round to nearest
_FP_RND_RP	Round toward +infinity
_FP_RND_RM	Round toward -infinity

**Note:** When processing IEEE floating-point values, the C/C++ run-time library math functions require IEEE rounding mode of round to nearest. The C/C++ run-time library takes care of setting round to nearest rounding mode while executing math functions and restoring application rounding mode before returning to the caller.

### Returned Value

For an application running in hexadecimal floating-point mode, `fp_swap_rnd()` returns 0.

For an application running in IEEE floating-point mode, `fp_swap_rnd()` returns the previous (changed from) rounding mode as follows:

Value	Rounding Mode
_FP_RND_RZ	Round toward 0
_FP_RND_RN	Round to nearest
_FP_RND_RP	Round toward +infinity
_FP_RND_RM	Round toward -infinity

**Related Information**

- “fp\_read\_rnd() — Determine Rounding Mode” on page 435
- “\_\_isBFP() — Determine Application Floating-Point Format” on page 705

## fputc() — Write a Character

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
int fputc(int c, FILE *stream);
```

### General Description

Converts *c* to an unsigned char and then writes *c* to the output stream pointed to by *stream* at the current position and advances the file position appropriately. The fputc() function is identical to *putc* but is always a function, because it is not available as a macro.

If the stream is opened with one of the append modes, the character is appended to the end of the stream regardless of the current file position.

The fputc() function is not supported for files opened with type=record.

fputc() has the same restriction as any write operation for a read immediately following a write, or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

### Returned Value

Returns the character written. A returned value of EOF indicates an error.

### Example

#### CBC3BF34

```
/* CBC3BF34
   This example writes the contents of buffer to a file called myfile.dat.
   Because the output occurs as a side effect within the second expression
   of the for statement, the statement body is null.
*/
#include <stdio.h>
#define NUM_ALPHA 26

int main(void)
{
    FILE * stream;
    int i;
    int ch;

    char buffer[NUM_ALPHA + 1] = "abcdefghijklmnopqrstuvwxyz";

    if (( stream = fopen("myfile.dat", "w")) != NULL )
    {
        /* Put buffer into file */
    }
```

```
    for ( i = 0; ( i < sizeof(buffer) ) &&  
        ((ch = fputc( buffer[i], stream)) != EOF ); ++i );  
    fclose( stream );  
}  
else  
    printf( "Error opening myfile.dat\n" );  
}
```

**Related Information**

- “stdio.h” on page 43
- “fgetc() — Read a Character” on page 388
- “putc() - putchar() — Write a Character” on page 1052

## fputs() — Write a String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
int fputs(const char *string, FILE *stream);
```

### General Description

Writes the string pointed to by *string* to the output stream pointed to by *stream*. It does not write the terminating `\0` at the end of the string.

For a text file, truncation may occur if the record is too long. *Truncation* means that excess characters are discarded after the record is full, up to a control character that ends the line (`\n`). Characters after the `\n` start at the next record. For more information, see the section on “Truncation” in the *OS/390 C/C++ Programming Guide*.

`fputs()` is not supported for files opened with `type=record`.

`fputs()` has the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

### Returned Value

If an error occurs, the `fputs()` function returns EOF; otherwise, it returns the number of bytes written.

### Example

#### CBC3BF35

```
/* CBC3BF35
   This example writes a string to a stream.
*/
#include <stdio.h>
#define NUM_ALPHA 26

int main(void)
{
    FILE * stream;
    int num;

    /* Do not forget that the '/' char occupies one character */
    static char buffer[NUM_ALPHA + 1] = "abcdefghijklmnopqrstuvwxyz";

    if ((stream = fopen("myfile.dat", "w")) != NULL )
    {
        /* Put buffer into file */
    }
```



```
if ( (num = fputs( buffer, stream )) != EOF )
{
    /* Note that fputs() does not copy the /0 character */
    printf( "Total number of characters written to file = %i\n", num );
    fclose( stream );
}
else /* fputs failed */
    printf( "fputs failed" );
}
else
    printf( "Error opening myfile.dat" );
}
```

**Related Information**

- “stdio.h” on page 43
- “fgets() — Read a String from a Stream” on page 392
- “gets() — Read a String” on page 595
- “puts() — Write a String” on page 1059

## fputwc() — Output a Wide-Character

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

#### Non-XPG4

```
#include <stdio.h>
#include <wchar.h>
```

```
wint_t fputwc(wchar_t wc, FILE *stream);
```

#### XPG4

```
#define _XOPEN_SOURCE
#include <stdio.h>
#include <wchar.h>
```

```
wint_t fputwc(wint_t wc, FILE *stream);
```

#### XPG4 and MSE

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <stdio.h>
#include <wchar.h>
```

```
wint_t fputwc(wchar_t wc, FILE *stream);
```

### General Description

Converts the wide character specified by *wc* to a multibyte character and writes it to the output stream pointed to by *stream*, at the position indicated by the associated file position indicator for the stream (if defined), and advances the indicator appropriately. If the file cannot support positioning requests or if the stream was opened with append mode, the character is appended to the output stream.

The behavior of this wide-character function is affected by the LC\_CTYPE category of the current locale. Using non-wide-character functions with *fputwc()* results in undefined behavior.

*fputwc()* has the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

### Special Behavior for XPG4

If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the *wchar* header, then the compiler assumes that your program is using the XPG4 variety of the *fputwc()* function, unless you

also define the `_MSE_PROTOS` feature test macro. Please see Table 3 on page 18 for a list of XPG4 and other feature test macros.

The prototype for the XPG4 variety of the `fputwc()` function is:

```
wint_t fputwc(wint_t wc, FILE *stream);
```

The difference between this variety and the MSE variety of the `fputwc()` function is that the first parameter has type `wint_t` rather than type `wchar_t`.

## Returned Value

Returns the wide character written. If a write error occurs, the error indicator for the stream is set and `WEOF` is returned. If an encoding error occurs during conversion from wide character to a multibyte character, the value of the macro `EILSEQ` is stored in `errno` and `WEOF` is returned.

## Example CBC3BF36

```
/* CBC3BF36 */
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
    FILE    *stream;
    wchar_t *wcs = L"This test string should not cause a WEOF condition";
    int      i;
    int      rc;

    if ((stream = fopen("myfile.dat", "w")) == NULL) {
        printf("Unable to open file.\n");
        exit(1);
    }

    for (i=0; wcs[i] != L'\0'; i++) {
        errno = 0;
        if ((rc = fputwc(wcs[i], stream)) == WEOF) {
            printf("Unable to fputwc() the wide character.\n");
            printf("wcs[%d] = 0x%x\n", i, wcs[i]);
            if (errno == EILSEQ)
                printf("An invalid wide character was encountered.\n");
            exit(1);
        }
    }

    fclose(stream);
}
```

## Related Information

- “`stdio.h`” on page 43
- “`wchar.h`” on page 54

## fputws() — Output a Wide-Character String

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#include <wchar.h>
```

```
int fputws(const wchar_t *wcs, FILE *stream);
```

### General Description

Converts the wide character string pointed to by *wcs* to a multibyte character string and writes it to the stream pointed to by *stream*, as a multibyte character string. The terminating null byte is not written.

The behavior of this wide-character function is affected by the LC\_CTYPE category of the current locale. Using non-wide-character functions with `fputws()` results in undefined behavior.

`fputws()` has the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

### Returned Value

Returns a non-negative value if successful.

If a *stream* error occurs, the error indicator for the stream is set and `-1` is returned. If an *encoding* error occurs, the value of the macro `EILSEQ` is stored in `errno` and `-1` is returned. An encoding error is one that occurs when converting a wide character to a multibyte character.

### Example

#### CBC3BF37

```
/* CBC3BF37 */
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
    FILE    *stream;
    wchar_t *wcs = L"This test string should not return -1";
    int      rc;

    if ((stream = fopen("myfile.dat", "w")) == NULL) {
        printf("Unable to open file.\n");
        exit(1);
    }

    errno = 0;
```

```
rc = fputws(wcs, stream);

if (rc == EOF) {
    printf("Unable to complete fputws() function.\n");
    if (errno == EILSEQ)
        printf("An invalid wide character was encountered.\n");
    exit(1);
}

fclose(stream);
}
```

**Related Information**

- “stdio.h” on page 43
- “wchar.h” on page 54

## fread() — Read Items

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
size_t fread(void *buffer, size_t size, size_t count, FILE
*stream);
```

### General Description

Reads up to *count* items of *size* length from the input stream pointed to by *stream* and stores them in the given *buffer*. The file position indicator advances by the number of bytes read.

If there is an error during the read operation, the file position indicator is undefined. If a partial element is read, the element's value is undefined.

When you are using `fread()` for record I/O, set *size* to 1 and *count* to the maximum expected length of the record, to obtain the number of bytes. Only one record is read, regardless of *count*, when using record I/O.

`fread()` has the same restriction as any read operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

### Returned Value

Returns the number of complete items successfully read. If *size* or *count* is 0, `fread()` returns 0, and the contents of the array and the state of the stream remain unchanged. For record I/O, it is possible that the number of complete items can be less than *count*. However, this result does not necessarily indicate that an error has occurred.

The `ferror()` and `feof()` functions are used to distinguish between a read error and an EOF. Note that EOF is only reached when an attempt is made to read “past” the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

### Example

#### CBC3BF38

```
/* CBC3BF38
   This example attempts to read NUM_ALPHA characters from the file myfile.dat.
   If there are any errors with either fread() or fopen(), a message is printed.
*/
#include <stdio.h>
#define NUM_ALPHA 26
```

```

int main(void)
{
    FILE * stream;
    int num;          /* number of characters read from stream */

    /* Do not forget that the '\0' char occupies one character too! */
    char buffer[NUM_ALPHA + 1];
    buffer[NUM_ALPHA] = '\0';

    if (( stream = fopen("myfile.dat", "r")) != NULL )
    {
        num = fread( buffer, sizeof( char ), NUM_ALPHA, stream );
        if (num == NUM_ALPHA) { /* fread success */
            printf( "Number of characters read = %i\n", num );
            printf( "buffer = %s\n", buffer );
            fclose( stream );
        }
        else { /* fread() failed */
            if ( ferror(stream) ) /* possibility 1 */
                printf( "Error reading myfile.dat" );
            else if ( feof(stream)) { /* possibility 2 */
                printf( "EOF found\n" );
                printf( "Number of characters read %d\n", num );
                printf( "buffer = %.*s\n", num, buffer);
            }
        }
    }
    else
        printf( "Error opening myfile.dat" );
}

```

## Related Information

- “stdio.h” on page 43
- “fopen() — Open a File” on page 417
- “freopen() — Redirect an Open File” on page 460
- “fwrite() — Write Items” on page 495

## free() — Free a Block of Storage

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
void free(void *ptr);
```

### General Description

Frees a block of storage pointed to by *ptr*. The *ptr* variable points to a block previously reserved with a call to `calloc()`, `malloc()`, or `realloc()`. The number of bytes freed is the number of bytes specified when you reserved (or reallocated, in the case of `realloc()`), the block of storage. If *ptr* is `NULL`, `free()` simply returns and does not set the pointer. `free()` will not set *ptr* to `NULL`.

This function is also available to C applications in a stand alone Systems Programming Environment.

**Note:** Attempting to free a block of storage not allocated with `calloc()`, `malloc()`, or `realloc()`, or previously freed storage, can affect the subsequent reserving of storage and lead to an abend.

### Special Behavior for C++

Under C++, you cannot use `free()` with an item that was allocated using the C++ `new` keyword.

### Returned Value

There is no returned value.

### Example

```
/* This example illustrates the use of calloc() to allocate storage for x
   array elements and then calls free() to free them.
   */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    long * array;    /* start of the array */
    long * index;    /* index variable */
    int i;           /* index variable */
    int num;         /* number of entries of the array */

    printf( "Enter the size of the array\n" );
    scanf( "%i", &num );

    /* allocate num entries */
    if ( (index = array = (long *)calloc( num, sizeof( long ))) != NULL )
    {
```



```

:
    /* do something with the array */
    free( array );                /* deallocates array */
}
else
{ /* Out of storage */
    printf( "Error: out of storage\n" );
    abort();
}
}

```

## Related Information

- “Using the System Programming C Facilities” in the *OS/390 C/C++ Programming Guide*
- “stdlib.h” on page 45
- “calloc() — Reserve and Initialize Storage” on page 141
- “malloc() — Reserve Storage Block” on page 786
- “realloc() — Change Reserved Storage Block Size” on page 1096
- “spc.h” on page 42

## freopen() — Redirect an Open File

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

### General Description

Closes the file currently associated with *stream* and pointed to by *stream*, opens the file specified by the *filename*, and then associates the stream with it.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

The `freopen()` function opens the new file with the type of access requested by the *mode* argument. The *mode* argument is used as in the `fopen()` function. See “`fopen()` — Open a File” on page 417 for a description of the *mode* parameter.

You can also use the `freopen()` function to redirect the standard stream files `stdin`, `stdout`, and `stderr` to files that you specify. The file pointer input to the `freopen()` function must point to a valid open file. If the file pointer has been closed, the behavior is undefined.

You could use the following `freopen()` call to redirect `stdout` to a memory file `a.b`:

```
freopen("a.b", "wb, type=memory", stdout);
```

If *filename* is an empty string, `freopen()` closes the file and reuses the original file name. For details on how the file name and open mode is interpreted, see the *OS/390 C/C++ Programming Guide*.

A standard stream can be opened by default to a type of file not available to a general `fopen()`. This is true for standard streams under CICS, and also true for the default `stderr`, when running a non-POSIX Language Environment application.

The following statement uses `freopen()` to have `stdin` use binary mode instead of text mode:

```
fp = freopen("", "rb", stdin);
```

You can use the same empty string method to change the mode from binary back to text. This method is not allowed for:

- The default CICS data queues used by the standard streams under CICS
- The LE MSGFILE, which is the default for `stderr`
- HFS files.

**Note:** Using the empty string method is included in the SAA C definition, but not in the ANSI C standard.

### Returned Value

If successful, the `freopen()` function returns the value of *stream*, the same value that was passed to it, and clears both the error and EOF indicators associated with the stream.

A failed attempt to close the original file is ignored.

If an error occurs in reopening the requested file, `freopen()` closes the original file, and returns a NULL pointer value.

### Example

This example illustrates the OS/390 C extension that allows you to change characteristics of a file by reopening it.

```
#include <stdio.h>

int main(void)
{
    FILE *stream, *stream2;

    stream = fopen("myfile.dat", "r");
    stream2 = freopen("", "w+", stream);
}
```

This example closes the stream data stream and reassigns its stream pointer:

```
#include <stdio.h>

int main(void)
{
    FILE *stream, *stream2;

    stream = fopen("myfile.dat", "r");
    stream2 = freopen("myfile2.dat", "w+", stream);
}
```

**Note:** `stream` and `stream2` will have the same value.

### Related Information

- “`stdio.h`” on page 43
- “`fclose()` — Close File” on page 348
- “`fopen()` — Open a File” on page 417

## frexp() — Extract Mantissa and Exponent of the Floating-Point Value

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double frexp(double x, int *expptr);
```

### General Description

Breaks down the floating-point value  $x$  into a component  $m$  for the normalized fraction component and another term  $n$  for the exponent, such that the absolute value of  $m$  is greater than or equal to 0.5 and less than 1.0 or equal to 0, and  $x = m * 2^n$ . The `frexp()` function stores the integer exponent  $n$  at the location to which `exp_ptr` points.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

Returns the normalized fraction  $m$ . If  $x$  is 0, the function returns 0 for both the fraction and exponent. The fraction has the same sign as the argument  $x$ . The result of the `frexp()` function cannot have a range error.

### Example

#### CBC3BF41

```
/* CBC3BF41
   This example decomposes the floating-point value of x, 16.4, into its
   normalized fraction 0.5125, and its exponent 5.
   It stores the mantissa in y and the exponent in n.
*/
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, m;
    int n;

    x = 16.4;
    m = frexp(x, &n);

    printf("The fraction is %lf and the exponent is %d\n", m, n);
}
```

### Output

The mantissa is 0.512500 and the exponent is 5

**Related Information**

- “math.h” on page 35
- “ldexp() — Multiply by a Power of Two” on page 738
- “modf() — Extract Fractional and Integral Parts of Floating-Point Value” on page 837

## fscanf() – scanf() – sscanf() — Read and Format Data

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
int fscanf (FILE *stream, const char *format-string, ...);
int scanf(const char *format-string, ...);
int sscanf(const char *buffer, const char *format, ... );
```

### General Description

These three related functions are referred to as the `fscanf` family.

Reads data from the current position of the specified *stream* into the locations given by the entries in the argument list, if any. The argument list, if it exists, follows the format string. The `fscanf()` function cannot be used for a file opened with `type=record`.

To provide an ASCII input/output format for applications using the `scanf()` or `sscanf()` functions, define feature test macro `__LIBASCII` as described on page 22.

The `scanf()` function reads data from the standard input stream `stdin` into the locations given by each entry in the argument list. The argument list, if it exists, follows the format string. `scanf()` *cannot* be used if `stdin` has been reopened as a `type=record` file.

The `sscanf()` function reads data from *buffer* into the locations given by *argument-list*. Reaching the end of the string pointed to by *buffer* is equivalent to `fscanf()` reaching EOF. If the strings pointed to by *buffer* and *format* overlap, behavior is undefined.

`fscanf()` and `scanf()` have the same restriction as any read operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

For all three functions, each entry in the argument list must be a pointer to a variable of a type that matches the corresponding conversion specification in *format-string*. If the types do not match, the results are undefined.

For all three functions, the *format-string* controls the interpretation of the argument list. The *format-string* can contain multibyte characters beginning and ending in the initial shift state.

The format string pointed to by *format-string* can contain one or more of the following:

- White-space characters, as specified by `isspace()`, such as blanks and new-line characters. A white-space character causes `fscanf()`, `scanf()`, and `sscanf()` to read, but not to store, all consecutive white-space characters in the input up to the next character that is not white space. One white-space character in *format-string* matches any combination of white-space characters in the input.
- Characters that are not white space, except for the percent sign character (%). A non-white-space character causes `fscanf()`, `scanf()`, and `sscanf()` to read, but not to store, a matching non-white-space character. If the next character in the input stream does not match, the function ends.
- Conversion specifications which are introduced by the percent sign (%) or the sequence (%n\$) where n is a decimal integer in the range [1,NL\_ARGMAX]. A conversion specification causes `fscanf()`, `scanf()`, and `sscanf()` to read and convert characters in the input into values of a conversion specifier. The value is assigned to an argument in the argument list.

All three functions read *format-string* from left to right. Characters outside of conversion specifications are expected to match the sequence of characters in the input stream; the matched characters in the input stream are scanned but not stored. If a character in the input stream conflicts with *format-string*, the function ends, terminating with a “matching” failure. The conflicting character is left in the input stream as if it had not been read.

When the first conversion specification is found, the value of the first *input field* is converted according to the conversion specification and stored in the location specified by the first entry in the argument list. The second conversion specification converts the second input field and stores it in the second entry in the argument list, and so on through the end of *format-string*.

#### Special Behavior for XPG4.2

- When the %n\$ conversion specification is found, the value of the *input field* is converted according to the conversion specification and stored in the location specified by the nth argument in the argument list. Numbered arguments in the argument list can only be referenced once from *format-string*.
- The *format-string* can contain either form of the conversion specification, that is, % or %n\$ but the two forms cannot be mixed within a single *format-string* except that %% or %\* can be mixed with the %n\$ form.

An *input field* is defined as:

- All characters until a white-space character (space, tab, or new line) is encountered
- All characters until a character is encountered that cannot be converted according to the conversion specification
- All characters until the field *width* is reached.

If there are too many arguments for the conversion specifications, the extra arguments are evaluated but otherwise ignored. The results are undefined if there are not enough arguments for the conversion specifications.

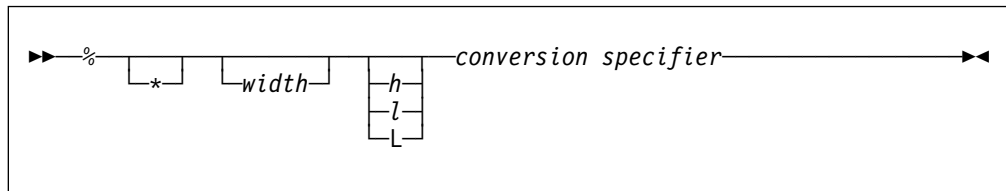


Figure 6. Syntax of Conversion Specification for *fscanf()*, *scanf()*, and *sscanf()*

Each field of the conversion specification is a single character or a number signifying a particular format option. The *conversion specifier*, which appears after the last optional format field, determines whether the input field is interpreted as a character, a string, or a number. The simplest conversion specification contains only the percent sign and a *conversion specifier* (for example, %s).

Each field of the format specification is discussed in detail below.

Other than conversion specifiers, you should avoid using the percent sign (%), except to specify the percent sign: %%. Currently, the percent sign is treated as the start of a conversion specifier. Any unrecognized specifier is treated as an ordinary sequence of characters. If, in the future, OS/390 C/C++ permits a new conversion specifier, it could match a section of your format string, be interpreted incorrectly, and result in undefined behavior. See Table 26 for a list of conversion specifiers.

An asterisk (\*) following the percent sign suppresses assignment of the next input field, which is interpreted as a field of the specified *conversion specifier*. The field is scanned but not stored.

*width* is a positive decimal integer controlling the maximum number of characters to be read. No more than *width* characters are converted and stored at the corresponding *argument*.

Fewer than *width* characters are read if a white-space character (space, tab, or new line), or a character that cannot be converted according to the given format occurs before *width* is reached.

The optional prefix l shows that you use the long version of the following *conversion specifier*, while the prefix h indicates that the short version is to be used. The corresponding *argument* should point to a long or double object (for the l character), a long double object (for the L character), or a short object (with the h character). The l and h modifiers can be used with the d, i, o, x, and u *conversion specifiers*. The l modifier can also be used with the e, f, and g *conversion specifiers*. The L modifier can be used with the e, f and g *conversion specifiers*. Note that the l modifier is also used with the c and s conversion specifiers to indicate a multibyte character or string. The l and h modifiers are ignored if specified for any other *conversion specifier*.

The *type* characters and their meanings are in Table 26.

Table 26 (Page 1 of 4). Conversion Specifiers in *fscanf()* and *scanf()*

Conversion Specifier	Type of Input Expected	Type of Argument
d	Decimal integer	Pointer to int



Table 26 (Page 2 of 4). Conversion Specifiers in fscanf() and scanf()

Conversion Specifier	Type of Input Expected	Type of Argument
o	Octal integer	Pointer to unsigned int
x X	Hexadecimal integer	Pointer to unsigned int
i	Decimal, hexadecimal, or octal integer	Pointer to int
u	Unsigned decimal integer	Pointer to unsigned int
e f g E G	Floating-point value consisting of an optional sign (+ or -); a series of one or more decimal digits possibly containing a decimal point; and an optional exponent (e or E) followed by a possibly signed integer value	Pointer to float

### fscanf Family of Formatted Input Functions

*fscanf* family functions match e, E, f, g or G conversion specifiers to floating-point number substrings in the input stream. *fscanf* family functions convert each input substring matched by an e, E, f, g or G conversion specifier to a float, double or long double value depending on a size modifier preceeding the e, E, f, g or G conversion specifier.

The floating-point value produced is hexadecimal floating-point or IEEE floating-point format depending on the floating-point mode of the thread invoking the *fscanf* family function. The *fscanf* family functions use `__isBFP()` to determine the floating-point mode of invoking threads.

Many OS/390 (C/C++) formatted input functions, including the *fscanf* family, recognize special infinity and NaN floating-point number input sequences when the invoking thread is in IEEE floating-point mode as determined by `__isBFP()`.

- The special sequence for infinity input is an optional plus or minus sign, then the character sequence INF, where the individual characters may be upper or lower case, and then a white-space character (space, tab, or new line), a null character (`\0`) or EOF.
- The special sequence for NaN input is an optional plus or minus sign, then the character sequence NANS for a signalling NaN or NANQ for a quiet NaN, where the individual characters may be upper or lower case, then an optional NaN ordinal sequence, and then a white-space character (space, tab, or new line), a null character (`\0`) or EOF.

A NaN ordinal sequence is a left-parenthesis character, "(", followed by a digit sequence representing an integer *n*, where  $1 \leq n \leq \text{INT\_MAX}-1$ , followed by a right-parenthesis character, ")". If the NaN ordinal sequence is omitted, NaN ordinal sequence (1) is assumed. The integer value, *n*, corresponding to a NaN ordinal sequence determines what IEEE floating-point NaN fraction bits are produced by formatted input functions.

For a signalling NaN, these functions produce NaN fraction bits (left to right) by reversing the bits (right to left) of the even integer value  $2*n$ .

For a quiet NaN they produce NaN fraction bits (left to right) by reversing the bits (right to left) of the odd integer value  $2*n-1$ .

D(n,p)	Fixed-point value consisting of an optional sign (+ or -); a series of one or more decimal digits possibly containing a decimal point.	Pointer to decimal
--------	--	--------------------

Table 26 (Page 3 of 4). Conversion Specifiers in fscanf() and scanf()

Conversion Specifier	Type of Input Expected	Type of Argument
c	(Can be used with the "l" modifier as lc). Character; white-space characters that are ordinarily skipped are read when c is specified.	Pointer to char large enough for input field.
C or lc	The input matches the number of multibyte characters specified by the field width. Each multibyte character in the sequence is converted to a wide character as if by a call to the mbstowcs() function. The conversion state described by mbstate_t object is initialized to zero before the first multibyte character is converted. The number of wide characters matched is specified by the field width (1 if no field width is present in the directive). The corresponding argument is a pointer to the initial element of an array of wchar_t large enough to accept the resulting sequence of wide characters. No null wide character is added.  C or lc is the multibyte character constant.	C or lc uses a pointer to wchar_t.
s	(Can be used with the "l" modifier as ls). String, which matches a sequence of multibyte characters that begins and ends in the initial shift state. None of the multibyte characters in the sequence are also single-byte white-space characters (as specified by the isspace() function). Each multibyte character in the sequence is converted to a wide character as if by a call to the mbrtowc() function, with the conversion state described by mbstate_t object initialized to zero before the first multibyte character is converted.	Pointer to character array large enough for input field, plus a terminating null character (\0) that is automatically appended.  S or ls uses a pointer to wchar_t string.
S or lS	The corresponding argument is a pointer to the initial array of wchar_t large enough to accept the sequence and the terminating null wide character, which is added automatically.  S or lS expects a multibyte string constant.	
n	No input read from <i>stream</i> or buffer.	Pointer to int, into which is stored the number of characters successfully read from the <i>stream</i> or buffer up to that point in the call to either fscanf() or to scanf().
p	Pointer to void converted to series of characters. For the specific format of the input, see the individual system reference guides.	Pointer to void.

Table 26 (Page 4 of 4). Conversion Specifiers in fscanf() and scanf()

Conversion Specifier	Type of Input Expected	Type of Argument
[ ]	<p>A non-empty sequence of bytes from a set of expected bytes (the <i>scanset</i>), which form the conversion specification. The conversion continues reading bytes until a failure to match or until an input failure.</p> <p>Consider the following situations:</p> <p>[^bytes]. In this case, the scanset contains all bytes that do not appear between the circumflex and the right square bracket.</p> <p>[abc] or [^]abc.] In both these cases the right square bracket is included in the scanset (in the first case: ]abc and in the second case, <i>not</i> ]abc)</p> <p>[a–z] The – is in the scanset; the characters b through y are <i>not</i> in the scanset.</p> <p>The code point for the square brackets ([ and ]) and the caret (^) vary among the EBCDIC encoded character sets. The default C locale expects these characters to use the code points for encoded character set Latin-1 / Open Systems 1047. Conversion proceeds one byte at a time: there is no conversion to wide characters.</p>	<p>Pointer to the initial byte of an array of char, signed char, or unsigned char large enough to accept the sequence and a terminating byte, which will be added automatically.</p>

When the LC\_SYNTAX category is set using setlocale(), the format strings passed to the fscanf(), scanf(), or sscanf() functions must use the same encoded character set as is specified for the LC\_SYNTAX category.

To read strings not delimited by space characters, substitute a set of characters in square brackets ([ ]) for the s (string) conversion specifier. The corresponding input field is read up to the first character that does not appear in the bracketed character set. If the first character in the set is a logical not (–), the effect is reversed: the input field is read up to the first character that does appear in the rest of the character set.

To store a string without storing an ending null character (\0), use the specification %ac, where *a* is a decimal integer. In this instance, the c conversion specifier means that the argument is a pointer to a character array. The next *a* characters are read from the input stream into the specified location, and no null character is added.

The input for a %x conversion specifier is interpreted as a hexadecimal number.

All three functions, fscanf(), scanf(), and sscanf() scan each input field character by character. It might stop reading a particular input field either before it reaches a space character, when the specified *width* is reached, or when the next character cannot be converted as specified. When a conflict occurs between the specification and the input character, the next input field begins at the first unread character. The conflicting character, if there is one, is considered unread and is the first character of the next input field or the first character in subsequent read operations on the input stream.

## Returned Value

All three functions, `fscanf()`, `scanf()`, and `sscanf()` return the number of input items that were successfully matched and assigned. The returned value does not include conversions that were performed but not assigned (for example, suppressed assignments). The functions return `EOF` if there is an input failure before any conversion, or if `EOF` is reached before any conversion. Thus a returned value of 0 means that no fields were assigned: there was a matching failure before any conversion. Also, if there is an input failure, then the file error indicator is set, which is not the case for a matching failure.

The `ferror()` and `feof()` functions are used to distinguish between a read error and an `EOF`. Note that `EOF` is only reached when an attempt is made to read “past” the last byte of data. Reading up to and including the last byte of data does *not* turn on the `EOF` indicator.

## Examples

### CBC3BF42

```
/* This example scans various types of data */
#include <stdio.h>

int main(void)
{
    int i;
    float fp;
    char c, s[81];

    printf("Enter an integer, a real number, a character "
           "and a string : \n");
    if (scanf("%d %f %c %s", &i, &fp, &c, s) != 4)
        printf("Not all of the fields were assigned\n");
    else
    {
        printf("integer = %d\n", i);
        printf("real number = %f\n", fp);
        printf("character = %c\n", c);
        printf("string = %s\n", s);
    }
}
```

## Output

If input is: 12 2.5 a yes, then output would be:

```
Enter an integer, a real number, a character and a string:
integer = 12
real number = 2.500000
character = a
string = yes
```

### CBC3BF43

```
/* CBC3BF43
   This example converts a hexadecimal integer to a decimal integer.
   The while loop ends if the input value is not a hexadecimal integer.
*/
#include <stdio.h>

int main(void)
{
    int number;
```

```

printf("Enter a hexadecimal number or anything else to quit:\n");
while (scanf("%x",&number))
{
    printf("Hexadecimal Number = %x\n",number);
    printf("Decimal Number      = %d\n",number);
}
}

```

### Output

If input is: 0x231 0xf5e 0x1 q, then output would be:

```

Enter a hexadecimal number or anything else to quit:
Hexadecimal Number = 231
Decimal Number      = 561
Hexadecimal Number = f5e
Decimal Number      = 3934
Hexadecimal Number = 1
Decimal Number      = 1

```

### CBC3BF44

```

/* CBC3BF44
   The next example illustrates the use of scanf() to input fixed-point
   decimal data types. This example works under C only, not C++.
*/
#include <stdio.h>
#include <decimal.h>

decimal(15,4) pd01;
decimal(10,2) pd02;
decimal(5,5) pd03;

int main(void) {
    printf("\nFirst time :-----\n");
    printf("Enter three fixed-point decimal number\n");
    printf("  (15,4) (10,2) (5,5)\n");
    if (scanf("%D(15,4) %D(10,2) %D(5,5)", &pd01, &pd02, &pd03) != 3) {
        printf("Error found in scanf\n");
    } else {
        printf("pd01 = %D(15,4)\n", pd01);
        printf("pd02 = %D(10,2)\n", pd02);
        printf("pd03 = %D(5,5)\n", pd03);
    }
    printf("\nSecond time :-----\n");
    printf("Enter three fixed-point decimal number\n");
    printf("  (15,4) (10,2) (5,5)\n");
    if (scanf("%D(15,4) %D(10,2) %D(5,5)", &pd01, &pd02, &pd03) != 3) {
        printf("Error found in scanf\n");
    } else {
        printf("pd01 = %D(15,4)\n", pd01);
        printf("pd02 = %D(10,2)\n", pd02);
        printf("pd03 = %D(5,5)\n", pd03);
    }
    return(0);
}

```

### Output

```

First time :-----
Enter three fixed-point decimal number
(15,4) (10,2) (5,5)
12345678901.2345 -987.6 .24680
pd01 = 12345678901.2345
pd02 = -987.60
pd03 = 0.24680

```

```

Second time :-----
Enter three fixed-point decimal number
(15,4) (10,2) (5,5)
123456789013579.24680 123.4567890 987
pd01 = 12345678901.3579
pd02 = 123.45
pd03 = 0.98700

```

**CBC3BF46**

```

/* CBC3BF46
   The next example opens the file myfile.dat for reading and then scans
   this file for a string, a long integer value, a character, and a
   floating-point value.
*/
#include <stdio.h>
#define MAX_LEN 80

int main(void)
{
    FILE *stream;
    long l;
    float fp;
    char s[MAX_LEN + 1];
    char c;

    stream = fopen("myfile.dat", "r");

    /* Put in various data. */
    fscanf(stream, "%s", &s[0]);
    fscanf(stream, "%ld", &l);
    fscanf(stream, "%c", &c);
    fscanf(stream, "%f", &fp);

    printf("string = %s\n", s);
    printf("long double = %ld\n", l);
    printf("char = %c\n", c);
    printf("float = %f\n", fp);
}

```

**Output**

If myfile.dat contains abcdefghijklmnopqrstuvwxyz 343.2, then the expected output is:

```

string = abcdefghijklmnopqrstuvwxyz
long double = 343
char = .
float = 2.000000

```

**CBC3BS32**

```

/* CBC3BS32
   This example uses sscanf() to read various data from the string
   tokenstring, and then displays the data.
*/
#include <stdio.h>
#define SIZE 81

int main(void)
{
    char *tokenstring = "15 12 14";
    int i;
    float fp;
    char s[SIZE];
    char c;

    /* Input various data */
    printf("No. of conversions=%d\n",
           sscanf(tokenstring, "%s %c%d%f", s, &c, &i, &fp));

    /* If there were no space between %s and %c,
       /* sscanf would read the first character following
       /* the string, which is a blank space.

    /* Display the data */
    printf("string = %s\n",s);
    printf("character = %c\n",c);
    printf("integer = %d\n",i);
    printf("floating-point number = %f\n",fp);
}

```

## Output

You would see this output from example CBC3BS32.

```

No. of conversions = 4
string = 15
character = 1
integer = 2
floating-point number = 14.000000

```

## Related Information

- “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*
- “locale.h” on page 33
- “stdio.h” on page 43
- “fprintf() - printf() - sprintf() — Format and Write Data” on page 436
- “\_\_isBFP() — Determine Application Floating-Point Format” on page 705
- “localtime() — Convert Time and Correct for Local Time” on page 758
- “setlocale() — Set Locale” on page 1241

## fseek() — Change File Position

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
int fseek(FILE *stream, long int offset, int origin);
```

### General Description

Changes the current file position associated with *stream* to a new location within the file. The next operation on the *stream* takes place at the new location. On a *stream* open for update, the next operation can be either a reading or a writing operation.

The *origin* must be one of the following constants defined in `stdio.h`:

#### Origin      Definition

SEEK\_SET    Beginning of file

SEEK\_CUR    Current position of file pointer

SEEK\_END    End of file

**Note:** If you specify SEEK\_CUR, any characters pushed back by `ungetc()` or `ungetwc()` will have backed up the current position of the file pointer—which is the starting point of the seek. The seek will discard any pushed back characters before repositioning, but the starting point will still be affected. For more information about calling `fseek()` after an `ungetc()` or `ungetwc()` see “`ungetc()` — Push Character onto Input Stream” on page 1655 and “`ungetwc()` — Push a Wide Character onto a Stream” on page 1658.

The `_EDC_COMPAT` environment variable causes `fseek()` to ignore the effects of `ungetc()` or `ungetwc()`. For more details, see “Environment Variables” in the *OS/390 C/C++ Programming Guide*.

### Binary Streams

ANSI states that binary streams use relative byte offsets for both `ftell()` and `fseek()`. Under OS/390 C/C++, this is true except for record-oriented files that have variable length records. For these types of files, the default behavior is to use encoded offsets for `ftell()` and `fseek()`, using an origin of SEEK\_SET.

Encoded offsets restrict you to seeking only to those positions that are recorded by a previous `ftell()` or to position 0. If you want to use relative-byte offsets for these types of files, you can either open with the `BYTESEEK` `fopen()` option or set the `_EDC_BYTESEEK` environment variable prior to opening. For details about `BYTESEEK` or `_EDC_BYTESEEK`, see the *OS/390 C/C++ Programming Guide*.



With relative-byte offsets, you are free to calculate your own offsets. If the offset exceeds the EOF, your file is extended with nulls, except for HFS files, for which the file is only extended with nulls if you subsequently write new data. This is true also under POSIX, using HFS files, where the file is only extended with nulls if you subsequently write new data.

Attempting to reposition to before the start of the file causes `fseek()` to fail.

Regardless of whether encoded or relative offsets are returned by `ftell()`, you can specify relative offsets when using `SEEK_CUR` and `SEEK_END`.

If the new position is before the start of the file, `fseek()` fails. If the relative offset is positioned beyond the EOF, the file is padded with nulls, except in the case of POSIX, using HFS files, where padding does not occur until a subsequent write of new data.

### Text Streams

For text streams, `ftell()` returns an encoded offset. When seeking with an origin of `SEEK_SET`, you are restricted to seeking only to 0 or to positions returned by a previous `ftell()`.

Attempting to calculate your own position is not supported, and may result in an invalid position and the failure of `fseek()`.

When you are using `SEEK_CUR` or `SEEK_END`, the offset is a relative byte offset. Attempting to seek to before the start of the file or past the EOF results in failure.

### Record I/O

For files opened as `type=record`, `ftell()` returns the relative record number. For the origins of `SEEK_SET`, `SEEK_CUR`, and `SEEK_END`, the offset is a relative record number.

Attempting to seek to before the first record or past the EOF results in failure.

For wide-oriented streams, all the above restrictions apply.

**Attention:** Repositioning within a wide-oriented file and performing updates is strongly discouraged because it is not possible to predict if your update will overwrite part of a multibyte string or character, thereby invalidating subsequent data. For example, you could inadvertently add data that overwrites a shift-out. The following data expects the shift-out to be there, so is not valid if it is treated as if in the initial shift state. Repositioning to the end of the file and adding new data is safe.

For details about wide-oriented streams, see the *OS/390 C/C++ Programming Guide*.

If successful, the `fseek()` function clears the EOF indicator, even when *origin* is `SEEK_END`, and cancels the effect of any preceding `ungetc()` or `ungetwc()` function on the same stream.

If the call to the `fseek()` function or the `fsetpos()` function is invalid, the call is treated as a flush and the `ungetc` characters are discarded.

## Returned Value

Returns the value 0 if it successfully moves the pointer. A nonzero returned value indicates an error. On devices that cannot seek, such as terminals and printers, the returned value is nonzero.

## Special Behavior for XPG4.2

The `fseek()` function returns -1 and sets `errno` to `ESPIPE` if the underlying file type for the stream is a PIPE or a socket.

## Example

```
/* This example opens a file myfile.dat for reading.
   After performing input operations (not shown), it moves the file
   pointer to the beginning of the file.
*/
#include <stdio.h>

int main(void)
{
    FILE *stream;
    int result;

    if (stream = fopen("myfile.dat", "r"))
    { /* successful */

        if (fseek(stream, 0L, SEEK_SET)); /* moves pointer to */
                                           /* the beginning of the file */
        { /* if not equal to 0
            then error ... */
        }
        else {
            /* fseek() successful */
        }
    }
}
```

## Related Information

- “stdio.h” on page 43
- “ftell() — Get Current File Position” on page 485
- “ungetc() — Push Character onto Input Stream” on page 1655
- “ungetwc() — Push a Wide Character onto a Stream” on page 1658

## fsetpos() — Set File Position

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

### General Description

Moves the file position associated with *stream* to a new location within the file according to the value of the object pointed to by *pos*. The value of *pos* must be obtained by a call to the `fgetpos()` library function. If successful, the `fsetpos()` function clears the EOF indicator, and cancels the effect of any previous `ungetc()` or `ungetwc()` function on the same stream.

If the call to the `fsetpos()` function is invalid, the call is treated as a flush, and the `ungetc` characters are discarded.

The `fsetpos()` function handles double-byte character set (DBCS) state information for wide-oriented files. An `fsetpos()` call to a position that no longer exists results in an error.

For text streams, the DBCS shift state is recalculated from the start of the record, which has a performance implication. The `fsetpos()` function repositions to the start of a multibyte character.

For binary streams, the DBCS shift state is set to the state saved by the `fsetpos()` function. If the record has been updated in the meantime, the shift state may be incorrect.

After the `fsetpos()` call, the next operation on a stream in update mode may be input or output.

**Note:** Repositioning within a wide-oriented file and performing updates is strongly discouraged because it is not possible to predict if your update will overwrite part of a multibyte string or character, thereby invalidating subsequent data. For example, you could inadvertently add data that overwrites a shift-out. The following data expects the shift-out to be there, so is not valid if it is treated as if in the initial shift state. Repositioning to the end of the file and adding new data is safe.

### Returned Value

If `fsetpos()` successfully changes the current position of the file, it returns the value 0. If there is an error, `errno` is set and a nonzero value is returned.

### Special Behavior for XPG4.2

The `fsetpos()` function returns -1 and sets `errno` to `ESPIPE` if the underlying file type for the stream is a PIPE or a socket.

### Example

```
/* This example opens a file called myfile.dat for reading.
   After performing input operations (not shown), it moves the file
   pointer to the beginning of the file and rereads the first byte.
*/
#include <stdio.h>

int main(void)
{
    FILE *stream;
    int retcode;
    fpos_t pos, pos1, pos2, pos3;
    char ptr[20]; /* existing file 'myfile.dat' has 20 byte records */

    /* Open file, get position of file pointer, and read first record */

    stream = fopen("myfile.dat", "rb");
    fgetpos(stream, &pos);
    pos1 = pos;
    if (!fread(ptr, sizeof(ptr), 1, stream))
        printf("fread error\n");

    /* Perform a number of read operations. The value of 'pos'
       changes if 'pos' is passed to fgetpos() */
    :
    /* Re-set pointer to start of file and re-read first record */

    fsetpos(stream, &pos1);
    if (!fread(ptr, sizeof(ptr), 1, stream))
        printf("fread error\n");

    fclose(stream);
}
```

### Related Information

- “stdio.h” on page 43
- “fgetpos() — Get File Position” on page 390
- “ftell() — Get Current File Position” on page 485
- “rewind() — Set File Position to Beginning of File” on page 1139
- “ungetc() — Push Character onto Input Stream” on page 1655
- “ungetwc() — Push a Wide Character onto a Stream” on page 1658

## fstat() — Get Status Information about a File

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <sys/stat.h>
```

```
int fstat(int fildev, struct stat *info);
```

### General Description

Gets status information about the file specified by the open file descriptor *fildev* and stores it in the area of memory indicated by the *info* argument. The status information is returned in a `stat` structure, as defined in the `sys/stat.h` header file. The elements of this structure are described in “stat() — Get File Information” on page 1404.

### Returned Value

If successful, `fstat()` returns 0. If unsuccessful, it returns `-1` and sets `errno` to one of the following:

`EBADF`      *fildev* is not a valid open file descriptor.

`EINVAL`     *info* contains a null.

### Special Behavior for XPG4.2

The following new `errno`s:

`EIO`            An I/O error occurred while reading from the file system.

`EOVERFLOW`    The file size exceeded the storage reserved for `st_size` in the `stat` structure.

### Example

#### CBC3BF47

```
/* CBC3BF47
   This example gets status information for the file called temp.file.
*/
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

main() {
    char fn[]="temp.file";
    struct stat info;
    int fd;
```

```

if ((fd = creat(fn, S_IWUSR)) < 0)
    perror("creat() error");
else {
    if (fstat(fd, &info) != 0)
        perror("fstat() error");
    else {
        puts("fstat() returned:");
        printf("  inode:   %d\n", (int) info.st_ino);
        printf(" dev id:  %d\n", (int) info.st_dev);
        printf("   mode:  %08x\n", info.st_mode);
        printf("  links:  %d\n", info.st_nlink);
        printf("   uid:   %d\n", (int) info.st_uid);
        printf("   gid:   %d\n", (int) info.st_gid);
        printf("created:  %s", ctime(&info.st_createtime));
    }
    close(fd);
    unlink(fn);
}
}

```

**Output**

```

fstat() returned:
  inode:   3057
 dev id:   1
   mode:  03000080
  links:   1
   uid:    25
   gid:    500
created:   Fri Jun 16 16:03:16 1995

```

**Related Information**

- “sys/stat.h” on page 48
- “sys/types.h” on page 49
- “fcntl() — Control Open File Descriptors” on page 350
- “lstat() — Get Status of File or Symbolic Link” on page 778
- “open() — Open a File” on page 872
- “stat() — Get File Information” on page 1404

## fstatvfs() — Get File System Information

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/statvfs.h>
int fstatvfs(int fildev, struct statvfs *fsinfo);
```

### General Description

The `fstatvfs()` function obtains information about the file system containing the file referenced by *fildev* and stores it in the area of memory pointed to by the *fsinfo* argument.

The information is returned in a `statvfs` structure, as defined in the `sys/statvfs.h` header file. The elements of this structure are described in “`statvfs()` — Get File System Information” on page 1408. If `fstatvfs()` successfully determines this information, it stores it in the area indicated by the *fsinfo* argument. The size of the buffer determines how much information is stored; data that exceeds the size of the buffer is truncated.

### Returned Value

If successful, `fstatvfs()` returns zero. If unsuccessful, `fstatvfs()` returns `-1`, and returns the error value in `errno`. The following are the possible values of `errno`:

`EBADF`      *fildev* is not a valid open file descriptor.

`EIO`        An I/O error has occurred while reading the file system.

`EINTR`      A signal was caught during the execution of the function.

### Example

```
#include <sys/statvfs.h>
#include <stdio.h>

main()
{
    char fn[]="temp.file";
    int fd;
    struct statvfs buf;

    if ((fd = creat(fn,S_IWUSR)) < 0)
        perror("creat() error");
    else {
        if (fstatvfs(fd, &buf) == -1)
            perror("fstatvfs() error");
        else {
            printf("each block is %d bytes big\n", buf.f_bsize);
            printf("there are %d blocks available\n", buf.f_bavail);
            printf("out of a total of %d in bytes,\n", buf.f_blocks);
            printf("that's %.0f bytes free out of a total of %.0f\n",
                ((double)buf.f_bavail * buf.f_bsize),
                ((double)buf.f_blocks * buf.f_bsize));
        }
    }
}
```

```

        close(fd);
        unlink(fn);
    }
}

```

### Output

each block is 4096 bytes big  
 there are 2089 blocks available  
 out of a total of 2400 in bytes,  
 that's 8556544 bytes free out of a total of 9830400

### Related Information

- “sys/statvfs.h” on page 49
- “chmod() — Change the Mode of a File or Directory” on page 174
- “chown() — Change the Owner or Group of a File or Directory” on page 177
- “creat() — Create a New File or Rewrite an Existing One” on page 233
- “dup() — Duplicate an Open File Descriptor” on page 288
- “fcntl() — Control Open File Descriptors” on page 350
- “link() — Create a Link to a File” on page 749
- “mknod() — Make a Directory or File” on page 823
- “open() — Open a File” on page 872
- “pipe() — Create an Unnamed Pipe” on page 907
- “read() — Read From a File or Socket” on page 1080
- “rexec() — Execute Commands One at a Time on a Remote Host” on page 1143
- “time() — Determine Current Time” on page 1570
- “unlink() — Remove a Directory Entry” on page 1660
- “utime() — Set File Access and Modification Times” on page 1664
- “write() — Write Data on a File or Socket” on page 1780



## fsync() — Write Changes to Direct-Access Storage

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX1_SOURCE 2
#include <unistd.h>
```

```
int fsync(int filides);
```

### General Description

Transfers all data for the file indicated by the open file descriptor *filides* to the storage device associated with *filides*. *fsync()* does not return until the transfer has completed, or until an error is detected.

### Returned Value

If successful, *fsync()* returns 0. If unsuccessful, it returns *-1* and sets *errno* to one of the following:

**EBADF**        *filides* is not a valid open file descriptor.

**EINVAL**      The file is not a regular file.

### Example

#### CBC3BF48

```
/* CBC3BF48 */
#define _POSIX1_SOURCE 2
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

#define mega_string_len 250000

main() {
    char *mega_string;
    int fd, ret;
    char fn[]="fsync.file";

    if ((mega_string = (char*) malloc(mega_string_len)) == NULL)
        perror("malloc() error");
    else if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        memset(mega_string, 's', mega_string_len);
        if ((ret = write(fd, mega_string, mega_string_len)) == -1)
            perror("write() error");
        else {
            printf("write() wrote %d bytes\n", ret);
            if (fsync(fd) != 0)
                perror("fsync() error");
            else if ((ret = write(fd, mega_string, mega_string_len)) == -1)
```

```
        perror("write() error");
    else
        printf("write() wrote %d bytes\n", ret);
    }
    close(fd);
    unlink(fn);
}
}
```

**Output**

```
write() wrote 250000 bytes
write() wrote 250000 bytes
```

**Related Information**

- “unistd.h” on page 53
- “open() — Open a File” on page 872
- “write() — Write Data on a File or Socket” on page 1780

## ftell() — Get Current File Position

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
long int ftell(FILE *stream);
```

### General Description

Obtains the current value of the file position indicator for the stream pointed to by *stream*.

### Binary Streams

ANSI states that `ftell()` returns relative byte offsets from the beginning of the file for binary files. Under OS/390 C/C++, this is true except for record-oriented files that have variable length records. For these types of files, `ftell()` returns an encoded offset.

If you want to use relative-byte offsets for these types of files, you can either open your files with the `BYTESEEK` `fopen()` option, or set the `_EDC_BYTESEEK` environment variable prior to opening them. For details about `BYTESEEK` or `_EDC_BYTESEEK` see the *OS/390 C/C++ Programming Guide*.

### Text Streams

`ftell()` returns an encoded offset for text streams.

### Record I/O

For files opened for record I/O using the `type=record` open mode parameter, `ftell()` returns the relative *record* offset of the current file position from the beginning of the file. All offset values are given in terms of records. For more information about calling `ftell()` after an `ungetc()` or `ungetwc()` see “`ungetc()` — Push Character onto Input Stream” on page 1655 and “`ungetwc()` — Push a Wide Character onto a Stream” on page 1658.

### Returned Value

Returns the calculated value if successful. On error, `ftell()` returns `-1` and sets `errno` to a positive value.

### Special Behavior for XPG4.2

The `ftell()` function returns `-1` and sets `errno` to `ESPIPE` if the underlying file type for the stream is a PIPE or a socket.

**Example**

```

/* This example opens the file myfile.dat for reading.
   The current file pointer position is stored in the variable pos.
*/
#include <stdio.h>

int main(void)
{
    FILE *stream;
    long int pos;

    stream = fopen("myfile.dat", "rb");

    /* The value returned by ftell can be used by fseek()
       to set the file pointer if 'pos' is not -1          */

    if ((pos = ftell(stream)) != EOF)
        printf("Current position of file pointer found\n");
    fclose(stream);
}

```

**Related Information**

- “stdio.h” on page 43
- “fgetpos() — Get File Position” on page 390
- “fopen() — Open a File” on page 417
- “fseek() — Change File Position” on page 474
- “fsetpos() — Set File Position” on page 477

## ftime() — Get the Date and Time

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/timeb.h>
```

```
int ftime(struct timeb *tp);
```

### General Description

The `ftime()` function sets the `time` and `millitm` members of the `timeb` structure pointed to by `tp` to contain seconds and milliseconds, respectively, of the current time in seconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970.

### Returned Value

`ftime()` returns 0 unless overflow occurs. It returns -1 if overflow occurs.<sup>1</sup>

### Related Information

- “limits.h” on page 32
- “sys/timeb.h” on page 49
- “ctime() — Convert Time to a Character String” on page 250
- “time() — Determine Current Time” on page 1570

<sup>1</sup> Overflow occurs when the current time in seconds since 00:00:00 UTC, January 1, 1970 exceeds the capacity of the `time` member of the `timeb` structure pointed to by `tp`. The `time` member is type `time_t`.

## ftok() — Generate an Interprocess Communication (IPC) key

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/ipc.h>
```

```
key_t ftok(const char *path, int id);
```

### General Description

The `ftok()` function returns a key based on *path* and *id* that is usable in subsequent calls to `msgget()`, `semget()`, and `shmget()`. The *path* argument must be the pathname of an existing file that the process is able to `stat()`.

The `ftok()` function returns the same key value for all paths that name the same file, when called with the same *id* value. If a different *id* value is given, or a different file is given, a different key is returned. Only the low order 8-bits of *id* are significant, and must be nonzero.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

If successful, `ftok()` return a key.

If unsuccessful, `ftok()` returns `-1`, and returns the error value in `errno`. The following are the possible values of `errno`:

**EACCES** Search permission is denied for a component of the path prefix.

**EINVAL** The low order 8-bits of *id* are zero.

**ELOOP** Too many symbolic links were encountered in resolving *path*

**ENAMETOOLONG**

The length of the *path* argument exceeds **PATH\_MAX** or a pathname component is longer than **NAME\_MAX**. Or, the pathname resolution of a symbolic link produced an intermediate result whose length exceeds **PATH\_MAX**.

**ENOENT** A component of *path* does not name an existing file or *path* is an empty string.

**ENOTDIR** A component of the path prefix is not a directory.

### Related Information

- “`sys/ipc.h`” on page 47
- “`msgget()` — Get Message Queue” on page 847
- “`semget()` — Get a Set of Semaphores” on page 1172
- “`shmget()` — Get a Shared Memory Segment” on page 1290
- “`stat()` — Get File Information” on page 1404

## ftruncate() — Truncate a File

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX1_SOURCE 2
#include <unistd.h>
```

```
int ftruncate(int fil-des, off_t length);
```

### General Description

Truncates the file indicated by the open file descriptor *fil-des* to the indicated *length*. *fil-des* must be a regular file that is open for writing. If the file size exceeds *length*, any extra data is discarded. If the file size is smaller than *length*, bytes between the old and new lengths are read as zeros. A change to the size of the file has no impact on the file offset.

### Special Behavior for XPG4.2

If `ftruncate()` would cause the file size to exceed the soft file size limit for the process, `ftruncate()` will fail and a SIGXFSZ signal will be generated for the process.

When `ftruncate()` is completed successfully, it marks the `st_ctime` and `st_mtime` fields of the file. If `ftruncate()` is unsuccessful, the file is unchanged.

### Returned Value

If successful, `ftruncate()` returns 0. If unsuccessful, it returns `-1` and sets `errno` to one of the following:

`EBADF`      *fil-des* is not a valid open file descriptor.

`EINVAL`      *fil-des* does not refer to a regular file, it is opened read-only, or the *length* specified is incorrect.

`EROFS`      The file resides on a read-only file system.

### Special Behavior for XPG4.2

Adds the following:

`EINTR`      A signal was caught during execution.

`EIO`        An I/O error occurred while reading from or writing to a file system.

### Example

#### CBC3BF49

```
/* CBC3BF49 */
#define _POSIX1_SOURCE 2
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
```

```

#include <unistd.h>
#include <stdio.h>

#define string_len 1000

main() {
    char *mega_string;
    int fd, ret;
    char fn[]="write.file";
    struct stat st;

    if ((mega_string = (char*) malloc(string_len)) == NULL)
        perror("malloc() error");
    else if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        memset(mega_string, '0', string_len);
        if ((ret = write(fd, mega_string, string_len)) == -1)
            perror("write() error");
        else {
            printf("write() wrote %d bytes\n", ret);
            fstat(fd, &st);
            printf("the file has %ld bytes\n", (long) st.st_size);
            if (ftruncate(fd, 1) != 0)
                perror("ftruncate() error");
            else {
                fstat(fd, &st);
                printf("the file has %ld bytes\n", (long) st.st_size);
            }
        }
        close(fd);
        unlink(fn);
    }
}

```

**Output**

```

write() wrote 1000 bytes
the file has 1000 bytes
the file has 1 bytes

```

**Related Information**

- “unistd.h” on page 53
- “open() — Open a File” on page 872



# ftw() — Traverse a File Tree

## Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 4.3

## Format

```
#define _XOPEN_SOURCE
#include <ftw.h>

int ftw(const char *path,
        int (*fn)(const char *, const struct stat *, int),
        int ndirs);
```

## General Description

The `ftw()` function recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, `ftw()` calls the function pointed to by *fn*, passing it a pointer to a null-terminated string containing the name of the object, a pointer to a *stat* structure containing information about the object, and an integer. Possible values of the integer, defined in the `<ftw.h>` header, are:

- FTW\_D      for a directory
- FTW\_DNR    for a directory that cannot be read
- FTW\_F      for a file
- FTW\_SL     for a symbolic link
- FTW\_NS     for an object other than a symbolic link on which `stat()` could not be successfully executed. If the object is a symbolic link, and `stat()` failed, it is unspecified whether `ftw()` passes FTW\_SL or FTW\_NS to the user-supplied function.

If the integer is FTW\_DNR, descendants of that directory will not be processed. If the integer is FTW\_NS, the *stat* structure will contain undefined values. An example of an object that would cause FTW\_NS to be passed to the function pointed to by *fn* would be a file in a directory with read but without execute (search) permission.

The `ftw()` function visits a directory before visiting any of its descendants.

The `ftw()` function uses at most one file descriptor for each level in the tree.

The argument *ndirs* should be in the range of 1 to {OPEN\_MAX}.

The tree traversal continues until the tree is exhausted, an invocation of *fn* returns a nonzero value, or some other error, other than [EACCES], is detected within `ftw()`.

The *ndirs* argument specifies the maximum number of directory streams or file descriptors or both available for use by `ftw()` while traversing the tree. When `ftw()` returns it closes any directory streams and file descriptors it uses not counting any opened by the application-supplied *fn* function.

## Returned Value

If the tree is exhausted, `ftw()` returns 0. If the function pointed to by `fn` returns a nonzero value, `ftw()` stops its tree traversal and returns whatever value was returned by the function pointed to by `fn()`. If `ftw()` detects an error, it returns `-1`, and sets `errno` to indicate the error. The following are the possible values of `errno`: All other `errno`'s returned by `ftw()` are unchanged.

EACCES	Search permission is denied for any component of <i>path</i> or read permission is denied for <i>path</i> .
ELOOP	Too many symbolic links were encountered.
ENOENT	A component of <i>path</i> does not name an existing file or <i>path</i> is an empty string.
ENOTDIR	A component of <i>path</i> is not a directory.
EINVAL	The value of the <i>ndirs</i> argument is invalid.
ENAMETOOLONG	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.
ENAMETOOLONG	The length of <i>path</i> exceeds {PATH_MAX}, or a pathname component is longer than {PATH_MAX}.

## Related Information

- “ftw.h” on page 29
- “longjmp() — Restore Stack Environment” on page 768
- “lstat() — Get Status of File or Symbolic Link” on page 778
- “malloc() — Reserve Storage Block” on page 786
- “opendir() — Open a Directory” on page 877
- “siglongjmp() — Restore the Stack Environment and Signal Mask” on page 1327
- “stat() — Get File Information” on page 1404

## fupdate() — Update a VSAM Record

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <stdio.h>
```

```
size_t fupdate(const void *buffer, size_t size, FILE
*stream);
```

### General Description

Replaces the last record read from the VSAM cluster pointed to by *stream*, with the contents of *buffer* for a length of *size*. See “Performing VSAM I/O Operations” in the *OS/390 C/C++ Programming Guide* for details.

The fupdate() function can be used *only* with a VSAM data set opened in update mode (rb+/r+b, ab+/a+b, or wb+/w+b) with the type=record option.

The fupdate() function can only be used after an fread() call has been performed and before any other operation on that file pointer. For example, if you need to acquire the file position using ftell() or fgetpos(), you can do it either before the fread() or after the fupdate(). An fread() after an fupdate() retrieves the next updated record.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

### KSDS or KSDS PATH

The size of the record can be changed by a call to fupdate(). If the size is greater than the existing record size but less than or equal to the maximum record length of the file, a call to fupdate() will lengthen the record up to the maximum record length of the file. If the size is greater than the maximum record length of the file, the record is truncated and errno is set. If the size is less than or equal to the existing record length, all size bytes of the record are written, and no padding or overlaying occurs. The records will be shortened and not partially updated.

### ESDS, ESDS PATH, or RRDS

The size of a record cannot be changed by a call to fupdate(). If you call fupdate() with *size* smaller than the size of the existing record, *size* bytes of the record are updated; the remaining bytes are unchanged, and the record length remains unchanged.

The key of reference (the prime key if opened as a cluster, the alternative index key if opened as a path) cannot be changed by an update. If a data set is opened as a path, the prime key cannot be changed by an update. For RRDS files, the buffer must be an RRDS record structure, which includes an `rrds_key`.

### Returned Value

Returns the size of the updated record if the operation is successful. If the update operation is not successful, `fupdate()` returns zero.

### Example

#### CBC3BF50

```
/* CBC3BF50 */
#include <stdio.h>

int main(void)
{
    FILE *stream;
    struct record { char name[20];
                   char address[40];
                   int age;
    } buffer;
    int vsam_rc, numread;

    stream = fopen("DD:MYCLUS", "rb+", type=record");
    numread = fread(&buffer, 1, sizeof(buffer), stream);
    /* ... Update fields in the record ... */
    vsam_rc = fupdate(&buffer, sizeof(buffer), stream);
}
```

### Related Information

- “Performing VSAM I/O Operations” in the *OS/390 C/C++ Programming Guide*
- “stdio.h” on page 43
- “fdelrec() — Delete a VSAM Record” on page 359
- “flocate() — Locate a VSAM Record” on page 405

## fwrite() — Write Items

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
size_t fwrite(const void *buffer, size_t size, size_t count, FILE
*stream);
```

### General Description

Writes up to *count* items of size *size* from the location pointed to by *buffer* to the stream pointed to by *stream*.

When you are using `fwrite()` for record I/O output, set *size* to 1 and *count* to the length of the record to obtain the number of bytes written. You can only write one record at a time when you are using record I/O. Any string longer than the record length is cut off at the record length. A flush or reposition is required before a subsequent read.

Because `fwrite()` may buffer output before writing it out to the stream, data from prior `fwrite()` calls may be lost where a subsequent call to `fwrite()` causes a failure when the buffer is written to the stream.

`fwrite()` has the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

### Returned Value

Returns the number of items that were successfully written. This number can be smaller than *count* only if a write error occurs.

### Example

#### CBC3BF51

```
/* CBC3BF51
   This example writes NUM long integers to a stream in binary format.
   It checks that the fopen() function is successful and that 100 items
   are written to the stream.
*/
#include <stdio.h>
#define NUM 100

int main(void)
{
    FILE *stream;
    long list[NUM];
    int numwritten, number;
```

```
if((stream = fopen("myfile.dat", "w+b")) != NULL )
{
    for (number = 0; number < NUM; ++number)
        list[number] = number;
    numwritten = fwrite(list, sizeof(long), NUM, stream);
    printf("number of long characters written is %d\n", numwritten);
}
else
    printf("fopen error\n");
}
```

**Related Information**

- “stdio.h” on page 43
- “fopen() — Open a File” on page 417
- “freopen() — Redirect an Open File” on page 460
- “fread() — Read Items” on page 456

## gamma() — Calculate Gamma Function

### Standards

Standards / Extensions	C or C++	Dependencies
SAA XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double gamma(double x);
```

### Compiler Option

LANGVL(SAA), LANGVL(SAAL2), or LANGVL(EXTENDED)

### General Description

gamma() provides the same function as lgamma(), including the use of *signgam*. Use of lgamma() instead of gamma() is suggested by XPG4.2.

## gcvt() — Convert Double to String

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>
```

```
char *gcvt(double x, int ndigit,
           char *buf);
```

### General Description

The `gcvt()` function converts double floating-point argument values to floating-point output strings. The `gcvt()` function has been extended to determine the floating-point format (hexadecimal floating-point or IEEE floating-point) of double argument values by using `__isBFP()`.

OS/390 (C/C++) formatted output functions, including the `gcvt()` function, convert IEEE floating-point infinity and NaN argument values to special infinity and NaN floating-point number output sequences. See “*fprintf* Family of Formatted Output Functions” on page 441 for a description of the special infinity and Nan output sequences.

The `gcvt()` function converts `x` to a null-terminated string (similar to the `%g` format of “*fprintf()* - *printf()* - *sprintf()* — Format and Write Data” on page 436) in the array pointed to by `buf` and returns `buf`. It produces `ndigit` significant digits (limited to an unspecified value determined by the precision of a double) in `%f` if possible, or `%e` (scientific notation) otherwise. A minus sign is included in the returned string if `value` is less than 0. A radix character is included in the returned string if `value` is not a whole number. Trailing zeros are suppressed where `value` is not a whole number. The radix character is determined by the current locale. If “*setlocale()* — Set Locale” on page 1241 has not been called successfully, the default locale, “POSIX”, is used. The default locale specifies a period (.) as the radix character. The `LC_NUMERIC` category determines the value of the radix character within the current locale.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

If it succeeds, `gcvt()` returns the character equivalent of `x` as specified above.

If the conversion fails, `gcvt()` returns `NULL`.

### Related Information

- “*stdlib.h*” on page 45
- “*ecvt()* — Convert Double to String” on page 303
- “*fcvt()* — Convert Double to String” on page 358
- “*\_\_isBFP()* — Determine Application Floating-Point Format” on page 705



## getc() - getchar() — Read a Character

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#define _ALL_SOURCE_NO_THREADS
```

```
#include <stdio.h>
```

```
int getc(FILE *stream);
```

```
int getchar(void);
```

### General Description

Reads a single character from the current *stream* position and advances the *stream* position to the next character. The `getchar()` function is identical to `getc(stdin)`.

The `getc()` and `fgetc()` functions are identical. However, `getc()` and `getchar()` are provided in a highly efficient macro form. For performance purposes, it is recommended that the macro forms be used rather than the functional forms or `fgetc()`. By default, `stdio.h` provides the macro versions of these functions.

However, to get the functional forms, do one or more of the following:

- For C only: do *not* include `stdio.h`.
- Specify `#undef`, for example, `#undef getc`
- Surround the call statement by parentheses, for example, `(getc)`

`getc()` and `getchar()` are not supported for files opened with `type=record`.

`getc()` and `getchar()` have the same restriction as any read operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

If the application is not multi-threaded, then setting the `_ALL_SOURCE_NO_THREADS` feature test macro may improve performance of the application, because it allows use of the inline version of this function.

### Special Behavior for POSIX

In a multithreaded C application that uses POSIX(ON), in the presence of the feature test macro, `_OPEN_THREADS`, these macros are in an `#undef` status because they are not thread-safe.

**Note:** Because the `getc()` macro reevaluates its input argument more than once, you should never pass a stream argument that is an expression with side effects.

## Returned Value

The `getc()` and `getchar()` functions return the character read. A returned value of EOF indicates either an error or an EOF condition. If a read error occurs, the error indicator is set. If an EOF is encountered, the EOF indicator is set.

Use `ferror()` or `feof()` to determine whether an error or an EOF condition occurred. Note that EOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

## Example CBC3BG02

```
/* CBC3BG02
   This example gets a line of input from the stdin stream.
   You can also use getc(stdin) instead of getchar() in the for statement
   to get a line of input from stdin.
*/
#include <stdio.h>

#define LINE 80

int main(void)
{
    char buffer[LINE+1];
    int i;
    int ch;

    printf( "Please enter string\n" );

    /* Keep reading until either:
       1. the length of LINE is exceeded or
       2. the input character is EOF or
       3. the input character is a new-line character
    */

    for ( i = 0; ( i < LINE ) && (( ch = getchar()) != EOF) &&
           ( ch != '\n' ); ++i )
        buffer[i] = ch;

    buffer[i] = '\0'; /* a string should always end with '\0' */

    printf( "The string is %s\n", buffer );
}
```

## Output

```
Please enter string
hello world
The string is hello world
```

## Related Information

- “stdio.h” on page 43
- “fgetc() — Read a Character” on page 388
- “gets() — Read a String” on page 595
- “putc() - putchar() — Write a Character” on page 1052
- “ungetc() — Push Character onto Input Stream” on page 1655

## getclientid() — Get the Identifier for the Calling Application

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#define _OPEN_SYS_SOCKET_EXT
#include <sys/socket.h>
#include <sys/types.h>
```

```
int getclientid(int domain, struct clientid *clientid);
```

### General Description

The `getclientid()` function call returns the identifier by which the calling application is known to the TCP/IP address space. The *clientid* can be used in the `givesocket()` and `takesocket()` calls. However, this function is supplied for use by existing programs that depend on the address space name returned. Even for these programs it is recommended that the name be saved for its later use and the `__getclientid()` function be issued to reconstruct the *clientid* structure for use by `givesocket()` and `takesocket()`.

Parameter	Description
<i>domain</i>	The address domain requested.
<i>clientid</i>	The pointer to a <i>clientid</i> structure to be filled.

The *clientid* structure is filled in by the call and returned as follows:

The *clientid* structure:

```
struct clientid {
    int domain;
    union {
        char name[8];
        struct {
            int NameUpper;
            pid_t pid;
        } c_pid;
    } c_name;
    char subtaskname[8];

    struct {
        char type;
        union {
            char specific[19];
            struct {
                char unused[3];
                int SockToken;
            } c_close;
        } c_func;
    } c_reserved;
};
```

Element	Description
domain	The input <i>domain</i> value returned in the domain field of the <i>clientid</i> structure.

c_name.name	The application program's address space name, left-justified and padded with blanks.
subtaskname	The calling program's task identifier.
c_reserved	Specifies binary zeros.

**Returned Value**

The value 0 indicates success. The value -1 indicates an error. The value of errno indicates the specific error.

**Errno Code      Description**

EFAULT	Using the <i>clientid</i> parameter as specified would result in an attempt to access storage outside the caller's address space, or storage not modifiable by the caller.
--------	--

**Related Information**

- “sys/socket.h” on page 48
- “sys/types.h” on page 49
- “\_\_getclientid() — Get the PID Identifier for the Calling Application” on page 503

## \_\_getclientid() — Get the PID Identifier for the Calling Application

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>
#include <sys/types.h>
```

```
int __getclientid(int domain, struct clientid *clientid);
```

### General Description

The `__getclientid()` function call returns the process identifier (PID) by which the calling application is known to the TCP/IP address space. The *clientid* is used in the `givesocket()` and `takesocket()` calls. Use the `__getclientid()` function call to transfer sockets between the caller and the selected application. The `__getclientid()` function provides improved performance and integrity over the `getclientid()` function for applications that use the output of `__getclientid()` as input *clientids* for `givesocket()` and `takesocket()`.

Parameter	Description
<i>domain</i>	The address domain requested.
<i>clientid</i>	The pointer to a <i>clientid</i> structure to be filled.

The *clientid* structure:

```
struct clientid {
    int domain;
    union {
        char name[8];
        struct {
            int NameUpper;
            pid_t pid;
        } c_pid;
    } c_name;
    char subtaskname[8];

    struct {
        char type;
        union {
            char specific[19];
            struct {
                char unused[3];
                int SockToken;
            } c_close;
        } c_func;
    } c_reserved;
};
```

Element	Description
<i>domain</i>	The input <i>domain</i> value returned in the domain field of the <i>clientid</i> structure.

<i>c_pid.pid</i>	Is the label in the <i>clientid</i> structure that is filled in by the function call to the PID of the requestor (caller of <code>__getclientid()</code> ). It should be left as set because it is used by the <i>takesocket()</i> and <i>givesocket()</i> functions.
subtaskname	Blanks
c_reserved	Binary zeroes

### Returned Value

The value 0 indicates success. The value -1 indicates an error. The value of `errno` indicates the specific error.

Errno Code	Description
------------	-------------

EFAULT	Using the <i>clientid</i> parameter as specified would result in an attempt to access storage outside the caller's address space, or storage not modifiable by the caller.
--------	--

### Related Information

- “sys/types.h” on page 49
- “sys/socket.h” on page 48
- “getclientid() — Get the Identifier for the Calling Application” on page 501
- “givesocket() — Make the Specified Socket Available” on page 633
- “takesocket() — Acquire a Socket from Another Program” on page 1496

## getcontext() — Get User Context

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON)

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ucontext.h>
```

```
int getcontext(ucontext_t *ucp);
```

### General Description

The `getcontext()` function initializes the structure pointed to by `ucp` to the current user context of the calling process. The **ucontext\_t** type that `ucp` points to defines the user context and includes the contents of the calling process' machine registers, the signal mask, and the current execution stack. A subsequent call to `setcontext()` restores the saved context and returns control to a point in the program corresponding to the `getcontext()` call. Execution resumes as if the `getcontext()` call had just returned. The return value from `getcontext()` is the same regardless of whether the return is from the initial invocation or via a call to `setcontext()`.

The context created by `getcontext()` may be modified by the `makecontext()` function. Refer to `makecontext` for details.

`getcontext()` is similar in some respects to `sigsetjmp()` (and `setjmp()` and `_setjmp()`). The `getcontext()`–`setcontext()` pair, the `sigsetjmp()`–`siglongjmp()` pair, the `setjmp()`–`longjmp()` pair, and the `_setjmp()`–`_longjmp()` pair cannot be intermixed. A context saved by `getcontext()` should be restored only by `setcontext()`.

**Note:** Some compatibility exists with `siglongjmp()`, so it is possible to use `siglongjmp()` from a signal handler to restore a context created with `getcontext()`, but it is not recommended.

Portable applications should not modify or access the **uc\_mcontext** member of **ucontext\_t**. A portable application cannot assume that context includes any process-wide static data, possibly including `errno`. Users manipulating contexts should take care to handle these explicitly when required.

This function is supported only in a POSIX program.

The `<ucontext.h>` header file defines the **ucontext\_t** type as a structure that includes the following members:

<code>mcontext_t</code>	<code>uc_mcontext</code>	A machine-specific representation of the saved context.
<code>ucontext_t</code>	<code>*uc_link</code>	Pointer to the context that will be resumed when this context returns.
<code>sigset_t</code>	<code>uc_sigmask</code>	The set of signals that are blocked when this context is active.
<code>stack_t</code>	<code>uc_stack</code>	The stack used by this context.

### Special Behavior for C++

If `getcontext()` and `setcontext()` are used to transfer control in a OS/390 C++ program, the behavior in terms of the destruction of automatic objects is undefined. This applies to both OS/390 C++ and C/C++ OS/390 ILC modules. The use of `getcontext()` and `setcontext()` in conjunction with `try()`, `catch()`, and `throw()` is also undefined.

Do not issue `getcontext()` in a C++ constructor or destructor, since the saved context would not be usable in a subsequent `setcontext()` or `swapcontext()` after the constructor or destructor returns.

## Returned Value

If successful, `getcontext()` returns 0.

If unsuccessful, `getcontext()` returns -1. There are no defined `errno` values.

## Example

This example saves the context in `main` with the `getcontext()` statement. It then returns to that statement from the function `func` using the `setcontext()` statement. Since `getcontext()` always returns 0 if successful, the program uses the variable `x` to determine if `getcontext()` returns as a result of `setcontext()` or not.

```
/* This example shows the usage of getcontext() and setcontext(). */

#define _XOPEN_SOURCE_EXTENDED 1
#include <stdio.h>
#include <ucontext.h>

void func(void);

int x = 0;
ucontext_t context, *cp = &context;

int main(void) {

    getcontext(cp);
    if (!x) {
        printf("getcontext has been called\n");
        func();
    }
    else {
        printf("setcontext has been called\n");
    }
}

void func(void) {

    x++;
    setcontext(cp);
}
```

## Output

```
getcontext has been called
setcontext has been called
```



**Related Information**

- “ucontext.h” on page 52
- “makecontext() — Modify User Context” on page 783
- “setcontext() — Restore User Context” on page 1210
- “setjmp() — Preserve Stack Environment” on page 1234
- “\_setjmp() — Set Jump Point for a Non-Local Goto” on page 1237
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sigsetjmp() — Save Stack Environment and Signal Mask” on page 1346
- “swapcontext() — Save and Restore User Context” on page 1472

# getcwd() — Get Path Name of the Working Directory

## Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

## Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
char *getcwd(char *buffer, size_t size);
```

## General Description

Determines the path name of the working directory and stores it in *buffer*.

- size*            The number of characters in the *buffer* area.
- buffer*          The name of the buffer that will be used to hold the path name of the working directory. *buffer* must be big enough to hold the working directory name, plus a terminating null to mark the end of the name.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

## Returned Value

If successful, `getcwd()` returns a pointer to the buffer. If unsuccessful, it returns a NULL pointer.

If `getcwd()` fails, it sets `errno` to one of the following:

- EACCES**        The process did not have read or search permission on some component of the working directory's path name.
- EINVAL**        *size* is less than or equal to zero.
- EIO**            An input/output error occurred.
- ENOENT**        A component of the working directory's path name does not exist.
- ENOTDIR**       A directory component of the working directory's path name is not really a directory.
- ERANGE**        *size* is greater than zero, but less than the length of the working directory's path name, plus 1 for the terminating null.

## Example CBC3BG03

```
/* CBC3BG03
   This example determines the working directory.
*/
#define _POSIX_SOURCE
#include <unistd.h>
#include <stdio.h>

main() {
    char cwd[256];
```

```
if (chdir("/tmp") != 0)
    perror("chdir() error");
else {
    if (getcwd(cwd, sizeof(cwd)) == NULL)
        perror("getcwd() error");
    else
        printf("current working directory is: %s\n", cwd);
}
```

**Output**

current working directory is: /tmp

**Related Information**

- “unistd.h” on page 53
- “chdir() — Change the Working Directory” on page 167

## getdate() — Convert User Format Date and Time

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <time.h>
struct tm *getdate(const char *string);

extern int getdate_err;
```

### General Description

The `getdate()` function converts definable date and/or time specifications pointed to by *string* into a `tm` structure. The `tm` structure declaration is in the header `<time.h>`.

Templates are used to parse and interpret the input string. The templates are contained in text files created by the process and identified via the environment variable `DATMSK`. The `DATMSK` variable should be set to indicate the full pathname of the file that contains the templates. The first line in the template that matches the input specification is used for the interpretation and conversion into the internal time format.

The following field descriptors are supported:

%%	same as %.
%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%c	locale's appropriate date and time representation
%d	day of month (01-31; the leading 0 is optional)
%D	date as %m/%d/%y
%e	same as %d
%h	same as %b
%H	hour (00-23; the leading 0 is optional)
%I	hour (01-12; the leading 0 is optional)
%m	month number (00-11; the leading 0 is optional)
%M	minute (00-59; the leading 0 is optional)
%n	same as \n
%p	locale's equivalent of either AM or PM
%r	locale's 12 hour time representation. In the POSIX locale this is equivalent to %I:%M:%S %p

%R	time as %H:%M
%S	seconds (00-61; the leading 0 is optional). Leap seconds (60 and 61) are treated as though 59 were specified i.e., <code>tm_sec</code> in the <code>tm</code> structure returned by <code>getdate()</code> is set to 59 if the input string specifies 59, 60 or 61 for %S.
%t	same as <code>\t</code> (tab)
%T	time as %H:%M:%S
%w	weekday number (0-6; 0 indicates Sunday)
%x	locale's date representation. In the POSIX locale this is equivalent to <code>%m/%d/%y</code> .
%X	locale's time representation. In the POSIX locale this is equivalent to <code>%H:%M:%S</code> .
%y	year within century. When a century is not otherwise specified, values in the range 69-99 refer to years in the twentieth century (1969 to 1999 inclusive); values in the range 00-68 refer to years in the twenty-first century (2000 to 2068 inclusive).
%Y	year as <code>ccyy</code> (1969-2037)
%Z	time zone name or no characters if no time zone exists. If the time zone supplied for %Z is not the time zone <code>getdate()</code> expects, an invalid input specification error will result. The <code>getdate()</code> function calculates an expected time zone based on time and date information supplied to it.

The match between the template and input specification performed by `getdate()` is case insensitive.

The month and weekday names can consist of any combination of upper or lower case letters. The process can request that the input date and time specification be in a specific language by setting the `LC_TIME` category (see `setlocale()`).

Leading 0's are not necessary for the descriptors that allow leading 0's. However, at most two digits are allowed for those descriptors, including leading 0's. Extra whitespace in either the template file or in *string* is ignored.

The field descriptors %c, %x, and %X will not be supported if they include unsupported field descriptors.

The following rules apply for converting the input specification into a `tm` structure:

- If only weekday is given, today is assumed if the given day is equal to the current day and next week if it is less,
- If only the month is given, the current month is assumed if the given month is equal to the current month and next year if it is less and no year is given (the first day of the month is assumed if no day is given),
- If no hour, minute, and second are given, the current hour, minute and second are assumed,
- If no date is given, today is assumed if the given hour is greater than the current hour and tomorrow is assumed if it is less.

## Returned Value

If successful, `getdate()` returns a pointer to a `tm` structure. Otherwise, it returns a null pointer and sets the external variable `getdate_err` to a value indicating the error.

The `tm` structure to which `getdate()` returns a pointer is not shared with any other functions. Also, the `getdate()` function produces a `tm` structure unique to the thread on which it runs.

As is true for all external variables, C/370 allocates storage for the `getdate_err` external variable in writeable static storage which is shared among all threads. Thus, `getdate_err` is not intrinsically “thread-safe”.

C/370 allocates storage on a per thread basis for an analog of `getdate_err`. The `__gderr()` function returns a pointer to this storage. It is recommended that multi-thread applications and applications running from a DLL use the `__gderr()` function rather than `getdate_err` if `getdate()` returns a null pointer to determine in a thread-safe manner why `getdate()` was unsuccessful.

The `__gderr()` is defined as follows:

```
#include <time.h>
```

```
int *__gderr(void);
```

The `__gderr()` function returns a pointer to the thread specific value of `getdate_err`.

The following is a list of `getdate_err` settings and their description:

- 1      The `DATMSK` environment variable is null or undefined.
- 2      The template file cannot be opened for reading.
- 3      Failed to get file status information.
- 4      The template file is not a regular file.
- 5      An error was encountered while reading the template file.
- 6      Memory allocation failed (not enough memory available).
- 7      No line in the template file matches the input specification.
- 8      Invalid input specification. For example, February 31; or a time that can not be represented in a `time_t` (representing the time is seconds since Epoch - midnight, January 1, 1970).
- 9      Unable to determine current time.

**Note:** This value is unique for OS/390 UNIX services.

## Related Information

- “time.h” on page 51

## getdtablesize() — Get the File Descriptor Table Size

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
```

```
int getdtablesize(void);
```

### General Description

The `getdtablesize()` function is equivalent to `getrlimit()` with the `RLIMIT_NOFILE` option.

### Returned Value

The `getdtablesize()` function returns the current soft limit as if obtained from a call to `getrlimit()`.

There are no `errno` values defined for `getdtablesize()`.

### Related Information

- “`unistd.h`” on page 53
- “`close()` — Close a File” on page 191
- “`getrlimit()` — Control Maximum Resource Consumption” on page 591
- “`open()` — Open a File” on page 872
- “`select()` — Monitor Activity on Files/Sockets and Message Queues” on page 1159
- “`setrlimit()` — Control Maximum Resource Consumption” on page 1264

## getegid() — Get the Effective Group ID

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
gid_t getegid(void);
```

### General Description

Finds the effective group ID (*GID*) of the calling process.

### Returned Value

Returns the effective group ID (GID). It is always successful. There are no documented errors for this function.

### Example

#### CBC3BG04

```
/* CBC3BG04
   This example finds the group ID.
*/
#define _POSIX_SOURCE
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

main() {
    printf("my group id is %d\n", (int) getgid());
}
```

### Output

```
my group id is 500
```

### Related Information

- “sys/types.h” on page 49
- “getgid() — Get the Real Group ID” on page 521
- “setegid() — Set the Effective Group ID” on page 1213
- “setgid() — Set the Group ID” on page 1220



# getenv() — Get Value of Environment Variables

## Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

## Format

```
#include <stdlib.h>

char *getenv(const char *varname);
```

## General Description

Searches the table of environment variables for an entry corresponding to *varname* and returns a pointer to a buffer containing the current string value of *varname*.

## Special Behavior for POSIX

Under POSIX, the value of the char \*\*environ pointer is honored and used by getenv(). You can declare and use this pointer. Under POSIX(OFF) this is not the case: the table start cannot be modified. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information.

To provide an ASCII input/output format for applications using this function, define feature test macro \_\_LIBASCII as described on page 22.

## Returned Value

Returns a pointer to a buffer containing the current string value of *varname*.

If the *varname* is not found, getenv() returns a null pointer. The returned value is NULL if the given variable is not currently defined.

## Example CBC3BG05

```
/* CBC3BG05
   In this example, *pathvar points to the value of the PATH environment
   variable. In a POSIX environment, this variable would be from the CENV group ID.
*/
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *pathvar;

    pathvar = getenv("PATH");
    printf("pathvar=%s",pathvar);
}
```

### **Related Information**

- “Using Environment Variables” in the *OS/390 C/C++ Programming Guide*
- “stdlib.h” on page 45
- “clearenv() — Clear Environment Variables” on page 184
- “\_\_getenv() — Get an Environment Variable” on page 517
- “setenv() — Add, Delete, and Change Environment Variables” on page 1215
- “putenv() — Change or Add an Environment Variable” on page 1054

## \_\_getenv() — Get an Environment Variable

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#include <stdlib.h>
```

```
char *__getenv(const char *varname);
```

### General Description

\_\_getenv() returns a unique character pointer for each environmental variable. For single-threaded applications, this eliminates the need to copy the string returned by previous \_\_getenv() calls.

This function should not be used by multi-threaded applications. Updates to the environmental variable on another thread may invalidate the address returned by \_\_getenv() before the application copies the returned value.

The format of an environment variable is made up of three parts that are combined to form:

*name=value*

Where:

1. The first part, *name*, is a character string that represents the name of the environment variable. It is this part of the environment variable that \_\_getenv() tries to match with *varname*.
2. The second part, =, is a separator character (since the equal-sign is used as a separator character it cannot appear in the *name*).
3. The third part, *value*, is a null-terminated character string that represents the value that the environment variable, *name*, is set to. This is the part of the environment variable that \_\_getenv() returns a pointer to.

There are several ways to establish a set of environment variables.

- Set at program initialization time from the LE runtime option ENVAR.
- Set at program initialization time from a data set.
- If the program was invoked with a system() call, they can be inherited from the calling enclave.
- In the OS/390 UNIX environment they can also be inherited from the parent process if the program was invoked with one of the exec functions.
- During the running of a program they can be set with the setenv() function or the putenv() function.

For a list of the environment variables that OS/390 UNIX services support, see the chapter “Using Environment Variables” in the *OS/390 C/C++ Programming Guide*.

### Special Behavior for POSIX

Under POSIX, the value of the `char **environ` pointer is honored and used by `getenv()`. You can declare and use this pointer. Under POSIX(OFF) this is not the case: the table start cannot be modified. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information.

### Returned Value

If successful, `__getenv()` returns a pointer to the string containing the value of the environment variable specified by *varname*.

If unsuccessful, `__getenv()` returns a null pointer. The returned value is `NULL` if the given variable is not currently defined or if the system does not support environment variables.

### Related Information

- “Using Environment Variables” in the *OS/390 C/C++ Programming Guide*
- “stdlib.h” on page 45
- “clearenv() — Clear Environment Variables” on page 184
- “getenv() — Get Value of Environment Variables” on page 515
- “putenv() — Change or Add an Environment Variable” on page 1054
- “setenv() — Add, Delete, and Change Environment Variables” on page 1215

## geteuid() — Get the Effective User ID

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
uid_t geteuid(void);
```

### General Description

Finds the effective user ID (UID) of the calling process.

### Returned Value

Returns the effective user ID of the calling process. It is always successful. There are no documented errno's for this function.

### Example CBC3BG06

```
/* CBC3BG06
   This example returns information for your user ID.
*/
#define _POSIX_SOURCE
#include <sys/types.h>
#include <unistd.h>
#include <pwd.h>

main() {
    struct passwd *p;
    uid_t uid;

    if ((p = getpwuid(uid = geteuid())) == NULL)
        perror("getpwuid() error");
    else {
        puts("getpwuid() returned the following info for your userid:");
        printf(" pw_name  : %s\n",      p->pw_name);
        printf(" pw_uid   : %d\n", (int) p->pw_uid);
        printf(" pw_gid   : %d\n", (int) p->pw_gid);
        printf(" pw_dir   : %s\n",      p->pw_dir);
        printf(" pw_shell : %s\n",      p->pw_shell);
    }
}
```

### Output

getpwuid() returns the following information for your userid:

```
pw_name  : MVSUSR1
pw_uid   : 25
pw_gid   : 500
pw_dir   : /u/mvsusr1
pw_shell : /bin/sh
```

### **Related Information**

- “sys/types.h” on page 49
- “getuid() — Get the Real User ID” on page 618
- “seteuid() — Set the Effective User ID” on page 1218
- “setreuid() — Set Real and Effective User IDs” on page 1262
- “setuid() — Set the Effective User ID” on page 1278

## getgid() — Get the Real Group ID

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
gid_t getgid(void);
```

### General Description

Finds the real group ID (GID) of the calling process.

### Returned Value

Returns the real group ID of the calling process. It is always successful. There are no documented errors for this function.

### Example

#### CBC3BG07

```
/* CBC3BG07
   This example gets the real group ID.
*/
#define _POSIX_SOURCE
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

main() {
    printf("my group id is %d\n", (int) getgid());
}
```

### Output

```
my group id is 500
```

### Related Information

- “sys/types.h” on page 49
- “getegid() — Get the Effective Group ID” on page 514
- “geteuid() — Get the Effective User ID” on page 519
- “getuid() — Get the Real User ID” on page 618
- “setgid() — Set the Group ID” on page 1220

## **getgrent() — Get Group Database Entry**

The information for this function is included in “endgrent() — Group Database Entry Functions” on page 307.



## getgrgid() — Access the Group Database by ID

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <grp.h>
```

```
struct group *getgrgid(gid_t gid);
```

### General Description

Provides information about the group specified by *gid* and its members.

### Returned Value

If successful, `getgrgid()` returns a pointer to a group structure containing an entry from the group data base with the specified *gid*. The return value may point to static data that is overwritten by each call. This group structure, defined in the `grp.h` header file, contains the following members:

```
gr_name      The name of the group
gr_gid       The numerical group ID (GID)
gr_mem       A null-terminated vector of pointers to the individual member names
```

If unsuccessful, it returns a NULL pointer. There are no documented errors for this function.

### Example

#### CBC3BG08

```
/* CBC3BG08
   This example provides the root GID and group name.
*/
#define _POSIX_SOURCE
#include <sys/types.h>
#include <grp.h>
#include <stdio.h>
#include <stat.h>

main() {
    struct stat info;
    struct group *grp;

    if (stat("/", &info) < 0)
        perror("stat() error");
    else {
        printf("The root is owned by gid %d\n", info.st_gid);
        if ((grp = getgrgid(info.st_gid)) == NULL)
            perror("getgrgid() error");
        else
            printf("This group name is %s\n", grp->gr_name);
    }
}
```

### Output

The root is owned by gid 500  
This group name is SYS1

### Related Information

- “grp.h” on page 29
- “sys/types.h” on page 49
- “endgrent() — Group Database Entry Functions” on page 307
- “getgrnam() — Access the Group Database by Name” on page 525
- “getlogin() — Get the User Login Name” on page 550

## getgrnam() — Access the Group Database by Name

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <grp.h>
```

```
struct group *getgrnam(const char *name);
```

### General Description

Accesses the group structure containing an entry from the group data base with the specified *name*.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

If successful, `getgrnam()` returns a pointer to a group structure. The return value may point to static data that is overwritten by each call.

The group structure, defined in the `grp.h` header file, contains the following members:

```
gr_name    The name of the group
gr_gid     The numerical group ID (GID)
gr_mem     A null-terminated vector of pointers to the individual member names.
```

If unsuccessful or if the requested entry is not found, the function returns a NULL pointer. There are no documented `errno`s for this function.

### Example

#### CBC3BG09

```
/* CBC3BG09
   This example provides the members of a group.
*/
#define _POSIX_SOURCE
#include <sys/types.h>
#include <grp.h>
#include <stdio.h>

main() {
    struct group *grp;
    char  grpname[]="USERS", **curr;

    if ((grp = getgrnam(grpname)) == NULL)
        perror("getgrnam() error");
    else {
        printf("The following are members of group %s:\n", grpname);
        for (curr=grp->gr_mem; (*curr) != NULL; curr++)
```

```
        printf("  %s\n", *curr);  
    }  
}
```

### Output

The following are members of group USERS:

```
MVSUSR1  
MVSUSR2  
MVSUSR3  
MVSUSR4  
MVSUSR5  
MVSUSR6  
MVSUSR7  
MVSUSR8  
MVSUSR9
```

### Related Information

- “grp.h” on page 29
- “sys/types.h” on page 49
- “endgrent() — Group Database Entry Functions” on page 307
- “getgrgid() — Access the Group Database by ID” on page 523

## getgroups() — Get a List of Supplementary Group IDs

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
int getgroups(int size, gid_t list[ ]);
```

### General Description

Stores the supplementary group IDs of the calling process in the *list* array. *size* gives the number of *gid\_t* elements that can be stored in the *list* array.

### Returned Value

If successful, *getgroups()* returns the number of supplementary group IDs that it puts into *list*. This value is always greater than or equal to 1 and less than or equal to the value of *NGROUPS\_MAX* (which is defined in the *limits.h* header file).

If *size* is zero, *getgroups()* returns the total number of supplementary group IDs for the process. *getgroups()* does not try to store group IDs in *list*.

If unsuccessful, *getgroups()* returns *-1* and sets *errno* to *EINVAL*, which indicates that *size* was not equal to 0 and is less than the total number of supplementary group IDs for the process. *list* may or may not contain a subset of the supplementary group IDs for the process.

### Example

#### CBC3BG10

```
/* CBC3BG10
   This example provides a list of the supplementary group IDs.
*/
#define _POSIX_SOURCE
#include <sys/types.h>
#include <grp.h>
#include <unistd.h>
#include <stdio.h>

#define dim(x) (sizeof(x)/sizeof(x[0]))

main() {
    gid_t gids[500];
    struct group *grp;
    int count, curr;

    if ((count = getgroups(dim(gids), gids)) == -1)
        perror("getgroups() error");
    else {
        puts("The following is the list of my supplementary groups:");
        for (curr=0; curr<count; curr++) {
            if ((grp = getgrgid(gids[curr])) == NULL)
                perror("getgrgid() error");
```

```
        else
            printf(" %8s (%d)\n", grp->gr_name, (int) gids[curr]);
    }
}
```

### Output

The following is the list of my supplementary groups:

```
    SYS1 (500)
    KINGS (512)
    NOBLES (513)
    KNIGHTS (514)
    WIZARDS (515)
    SCRIBES (516)
    JESTERS (517)
    PEASANTS (518)
```

### Related Information

- “sys/types.h” on page 49
- “getegid() — Get the Effective Group ID” on page 514
- “getgid() — Get the Real Group ID” on page 521
- “getgrnam() — Access the Group Database by Name” on page 525
- “setgid() — Set the Group ID” on page 1220

## getgroupsbyname() — Get Supplementary Group IDs by User Name

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
int getgroupsbyname(char username[ ], int size, gid_t list[ ]);
```

### General Description

Stores the supplementary group IDs of the specified *username* in the array, *list*. *size* gives the number of *gid\_t* elements that can be stored in the array, *list*.

### Returned Value

If successful, `getgroupsbyname()` returns the number of supplementary group IDs that it puts into *list*. This value is always greater than or equal to one, and less than or equal to the value of `NGROUPS_MAX`.

If *size* is zero, `getgroupsbyname()` returns the total number of supplementary group IDs for the process. `getgroupsbyname()` does not try to store group IDs in *list*.

If unsuccessful, `getgroupsbyname()` returns `-1` and sets `errno` to `EINVAL`, which indicates that *size* was less than or equal to the total number of supplementary group IDs for the process. *list* may or may not contain a subset of the supplementary group IDs for the process.

### Example

#### CBC3BG11

```
/* CBC3BG11
   This example provides a list of the supplementary group IDs for MVSUSR1.
*/
#define _POSIX_SOURCE
#include <sys/types.h>
#include <grp.h>
#include <stdio.h>
#include <unistd.h>

#define dim(x) (sizeof(x)/sizeof(x[0]))

main() {
    gid_t gids[500];
    struct group *grp;
    int count, curr;
    char user[]="MVSUSR1";

    if ((count = getgroupsbyname(user, dim(gids), gids)) == -1)
        perror("getgroups() error");
    else {
        printf("The following is the list of %s's supplementary groups:\n",
            user);
        for (curr=0; curr<count; curr++) {
            if ((grp = getgrgid(gids[curr])) == NULL)
                perror("getgrgid() error");
```

## getgroupsbyname

```
        else
            printf(" %8s (%d)\n", grp->gr_name, (int) gids[curr]);
    }
}
```

### Output

The following is the list of MVSUSR1's supplementary groups:

```
    SYS1 (500)
    USERS (523)
```

### Related Information

- “unistd.h” on page 53
- “getegid() — Get the Effective Group ID” on page 514
- “getgid() — Get the Real Group ID” on page 521
- “getgroups() — Get a List of Supplementary Group IDs” on page 527
- “setgid() — Set the Group ID” on page 1220



## gethostbyaddr() — Get a Host Entry by Address

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>
extern int h_errno;

struct hostent *gethostbyaddr(const void *address,
                              size_t len, int type);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <netdb.h>

struct hostent *gethostbyaddr(char *address, int address_len,
                              int domain);
```

### General Description

The `gethostbyaddr()` call tries to resolve the host address through a name server, if one is present. If a server is present, a cache will be set up upon the first call to either `gethostbyaddr()` or `gethostbyname()` to store host names and IP addresses. The size of the cache may be set with the environmental variable `_EDC_IP_CACHE_ENTRIES`. The size of the cache may be set only once. If `_EDC_IP_CACHE_ENTRIES` is not specified, a default value of 20 will be used. If a name server is not present, `gethostbyaddr()` searches the `/etc/hosts/` or the `tcpip.HOSTS.ADDRINFO` data set until a matching host address is found or an EOF marker is reached.

Parameter	Description
<i>address</i>	The pointer to a structure containing the address of the host. (An unsigned long for AF_INET.)
<i>address_len</i>	The size of <i>address</i> in bytes.
<i>domain</i>	The address domain supported (AF_INET).

If you want `gethostbyaddr()` to bypass the name server and instead resolve the host address using the `/etc/hosts/` or `tcpip.HOSTS.ADDRINFO` file, you must define the `RESOLVE_VIA_LOOKUP` symbol before including any sockets-related include files in your source program.

You can use the **X\_ADDR** environment variable to specify a data set other than `tcpip.HOSTS.ADDRINFO`. For more information on these data sets and environment variables, see *OS/390 UNIX System Services Planning*.

**Note:** `tcpip.HOSTS.LOCAL`, `tcpip.HOSTS.ADDRINFO`, and `tcpip.HOSTS.SITEINFO` are described in *TCP/IP for MVS: Customization and Administration Guide*.

The `gethostbyaddr()` call returns a pointer to a **hostent** structure for the host address specified on the call.

`gethostent()`, `gethostbyaddr()`, and `gethostbyname()` all use the same static area to return the **hostent** structure. This static area is only valid until the next one of these functions is called on the same thread.

The **netdb.h** include file defines the **hostent** structure and contains the following elements:

Element	Description
<i>h_addr_list</i>	A pointer to a zero-terminated list of host network addresses.
<i>h_addrtype</i>	The type of address returned; currently, it is always set to <code>AF_INET</code> .
<i>h_aliases</i>	A zero-terminated array of alternative names for the host.
<i>h_length</i>	The length of the address in bytes.
<i>h_name</i>	The official name of the host.

The following function (X/Open sockets only) is defined in **netdb.h** and should be used by multithreaded applications when attempting to reference *h\_errno* return on error:

```
int *__h_errno(void);
```

Also use this function when you invoke `gethostbyaddr()` in a DLL.

This function returns a pointer to a thread-specific value for the *h\_errno* variable.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

The return value points to static data that is overwritten by subsequent calls. A pointer to a **hostent** structure indicates success. A NULL pointer indicates an error or end-of-file.

On unsuccessful completion in X/Open, this function sets *h\_errno* to indicate the error as follows:

Error Code	Description
<code>HOST_NOT_FOUND</code>	No such host is known.
<code>TRY_AGAIN</code>	A temporary error such as no response from a server, indicating the information is not available now but may be at a later time.
<code>NO_RECOVERY</code>	An unexpected server failure occurred from which there is no recovery.

**NO\_DATA**      The server recognized the request and the name but no address is available. Another type of request to the name server might return an answer.

**Related Information**

- “endhostent() — Work with a Host Entry” on page 308
- “gethostbyname() — Get a Host Entry by Name” on page 534
- “gethostent() — Get the Next Host Entry” on page 537
- “sethostent() — Open the Host Information Data Set” on page 1224

## gethostbyname() — Get a Host Entry by Name

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>
extern int h_errno;

struct hostent *gethostbyname(const char *name);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <netdb.h>

struct hostent *gethostbyname(char *name);
```

### General Description

The `gethostbyname()` call tries to resolve the host name through a name server, if one is present. If a server is present, a cache will be set up upon the first call to either `gethostbyaddr()` or `gethostbyname()` to store host names and IP addresses. The size of the cache may be set with the environmental variable `_EDC_IP_CACHE_ENTRIES`. The size of the cache may be set only once. If `_EDC_IP_CACHE_ENTRIES` is not specified, a default value of 20 will be used. `gethostbyname()` will search for the host name in the cache first. If it doesn't exist, the long name will be constructed from the name and domain and sent to the Name Server(s). If a name server is not present, `gethostbyname()` searches the `/etc/hosts/` or the `tcpip.HOSTS.SITEINFO` data set until a matching host name is found or an EOF marker is reached.

Parameter	Description
<i>name</i>	The name of the host.

The `gethostbyname()` call returns a pointer to a **hostent** structure for the host name specified on the call.

`gethostent()`, `gethostbyaddr()`, and `gethostbyname()` all use the same static area to return the **hostent** structure. This static area is only valid until the next one of these functions is called on the same thread.

If you want `gethostbyname()` to bypass the name server and instead resolve the host name using the `/etc/hosts/` or `tcpip.HOSTS.SITEINFO` file, you must define the `RESOLVE_VIA_LOOKUP` symbol before including any sockets-related include files in your source program.

If the name server is not present or the `RESOLVE_VIA_LOOKUP` option is in effect, you can use the **X\_SITE** environment variable to specify a data set other than `tcpip.HOSTS.SITEINFO`. For more information on these data sets and environment variables, see *OS/390 UNIX System Services Planning*.

**Note:** *tcpip.HOSTS.LOCAL*, *tcpip.HOSTS.ADDRINFO*, and *tcpip.HOSTS.SITEINFO* are described in *TCP/IP for MVS: Customization and Administration Guide*.

`gethostent()`, `gethostbyaddr()`, and `gethostbyname()` all use the same static area to return the `HOSTENT` structure. This static area is only valid until the next one of these functions is called on the same thread.

The **netdb.h** include file defines the **hostent** structure and contains the following elements:

Element	Description
<i>h_addr_list</i>	A pointer to a zero-terminated list of host network addresses.
<i>h_addrtype</i>	The type of address returned; currently, it is always set to <code>AF_INET</code> .
<i>h_aliases</i>	A zero-terminated array of alternative names for the host.
<i>h_length</i>	The length of the address in bytes.
<i>h_name</i>	The official name of the host.

The following function (X/Open sockets only) is defined in **netdb.h** and should be used by multithreaded applications when attempting to reference *h\_errno* return on error:

```
int *__h_errno(void);
```

Also use this function when you invoke `gethostbyname()` in a DLL. This function returns a pointer to a thread-specific value for the *h\_errno* variable.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

The return value points to static data that is overwritten by subsequent calls. A pointer to a **hostent** structure indicates success. A NULL pointer indicates an error or end-of-file.

On unsuccessful completion in X/Open, this function sets *h\_errno* to indicate the error as follows:

Error Code	Description
<code>HOST_NOT_FOUND</code>	No such host is known.
<code>TRY_AGAIN</code>	A temporary error such as no response from a server, indicating the information is not available now but may be at a later time.
<code>NO_RECOVERY</code>	An unexpected server failure occurred from which there is no recovery.

**NO\_DATA**      The server recognized the request and the name but no address is available. Another type of request to the name server might return an answer.

### Related Information

- “endhostent() — Work with a Host Entry” on page 308
- “gethostbyaddr() — Get a Host Entry by Address” on page 531
- “gethostent() — Get the Next Host Entry” on page 537
- “gethostname() — Get the Name of the Host Processor” on page 540
- “sethostent() — Open the Host Information Data Set” on page 1224

## gethostent() — Get the Next Host Entry

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct hostent *gethostent(void);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <netdb.h>

struct hostent *gethostent(void);
```

### General Description

The `gethostent()` call reads the next line of the `/etc/hosts` or `tcpip.HOSTS.SITEINFO` data set.

The `gethostent()` call returns a pointer to the next entry in the `/etc/hosts` or `tcpip.HOSTS.SITEINFO` data set. `gethostent()` uses `tcpip.HOSTS.ADDRINFO` to get aliases.

You can use the **X\_SITE** environment variable to specify a data set other than `tcpip.HOSTS.SITEINFO`. For more information on these data sets and environment variables, see *OS/390 UNIX System Services Planning*.

**Note:** `tcpip.HOSTS.LOCAL`, `tcpip.HOSTS.ADDRINFO`, and `tcpip.HOSTS.SITEINFO` are described in *TCP/IP for MVS: Customization and Administration Guide*.

`gethostent()`, `gethostbyaddr()`, and `gethostbyname()` all use the same static area to return the **hostent** structure. This static area is only valid until the next one of these functions is called on the same thread.

The **netdb.h** include file defines the **hostent** structure and contains the following elements:

Element	Description
<code>h_addrtype</code>	The type of address returned; currently, it is always set to <code>AF_INET</code> .
<code>h_addr</code>	A pointer to the network address of the host.
<code>h_aliases</code>	A zero-terminated array of alternative names for host.
<code>h_length</code>	The length of the address in bytes.
<code>h_name</code>	The official name of the host.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

The return value points to data that is overwritten by subsequent calls returning the same data structure. A pointer to a **hostent** structure indicates success. A NULL pointer indicates an error or end-of-file.

### Related Information

- “gethostbyaddr() — Get a Host Entry by Address” on page 531
- “gethostbyname() — Get a Host Entry by Name” on page 534
- “sethostent() — Open the Host Information Data Set” on page 1224



## gethostid() — Get the Unique Identifier of the Current Host

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
```

```
long gethostid(void);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <unistd.h>
```

```
int gethostid();
```

### General Description

The `gethostid()` call gets the unique 32-bit identifier for the current host. This value is the default home Internet address.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

If successful, the `gethostid()` call returns the 32-bit identifier of the current host, which should be unique across all hosts.

If unsuccessful, `gethostid()` returns `-1`, and returns the error value in `errno`. For return codes, see *OS/390 UNIX System Services Messages and Codes*.

### Related Information

- “`gethostname()` — Get the Name of the Host Processor” on page 540

# gethostname() — Get the Name of the Host Processor

## Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

## Format

### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>

int gethostname(char *name, size_t namelen);
```

### Berkeley Sockets

```
#define _OE_SOCKETS
#include <unistd.h>

int gethostname(char *name, int namelen);
```

## General Description

The `gethostname()` call returns the name of the host processor that the program is running on. Up to *namelen* characters are copied into the name array. The returned name is null-terminated unless there is insufficient room in the name array.

Parameter	Description
<i>name</i>	The character array to be filled with the host name.
<i>namelen</i>	The length of <i>name</i> .

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

## Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

## Returned Value

The value 0 indicates success; the value -1 indicates an error. The value of the error code indicates the specific error.

Error Code	Description
EFAULT	Using <i>name</i> and <i>namelen</i> would result in an attempt to copy the address into a portion of the caller's address space to which data cannot be written.
EMVSPARM	Incorrect parameters were passed to the service.

**Related Information**

- “gethostbyname() — Get a Host Entry by Name” on page 534
- “gethostid() — Get the Unique Identifier of the Current Host” on page 539

## getibmopt() — Get IBM TCP/IP Image

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R3

### Format

```
#define _OPEN_SYS SOCK_EXT
#include <sys/socket.h>
```

```
int getibmopt(int cmd, struct ibm_gettcpinfo *bfrp);
```

### General Description

The getibmopt() function call returns -1 with errno EOPNOTSUPP to indicate that this function is not currently supported.

#### Parameter Description

<i>cmd</i>	The value in domain must be AF_INET.
<i>bfrp</i>	The pointer to an ibm_gettcpinfo structure.

### Returned Value

This function always returns -1, indicating that this function is not currently supported.

#### Errno Code Description

EOPNOTSUPP	This function is not supported.
------------	---------------------------------

### Related Information

- “sys/socket.h” on page 48

# getibmssockopt() — Get the Options Associated with a Bulk Mode Socket

## Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R3

## Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>

int getibmssockopt(int s,int level,int optname,
    char *optval,int *optlen);
```

## General Description

Like getsockopt(), the getibmssockopt() gets the options associated with a socket in the AF\_INET domain. Only SOL\_SOCKET is supported. This call is for options specific to the IBM implementation of sockets. Currently, only the SOL\_SOCKET level and the socket options SO\_BULKMODE, SO\_NONBLOCKLOCAL, and SO\_IGNOREINCOMINGPUSH are supported.

Bulk mode is supported only for receive-type socket calls. Currently, send-type socket calls are not supported for bulk mode.

Use getibmssockopt() with the socket option SO\_BULKMODE to test whether the UDP socket *s* is in bulk mode. Normally, UNIT transactions occur between the socket application and the TCP/IP address space for every receive (read(), recv(), recvfrom(), or recvmsg()) or send (send(), sendto(), sendmsg(), or write()) issued on a socket. The bulk mode socket option enables an application to queue multiple datagrams, sending all of the datagrams in one UNIT transaction. This reduces the CPU consumption for each datagram.

This call is used only in the AF\_INET domain.

Parameter	Description
<i>s</i>	The socket descriptor.
<i>level</i>	The level for which the option is set.
<i>optname</i>	The name of a specified socket option.
<i>optval</i>	The pointer to option data.
<i>optlen</i>	The pointer to the length of the option data.

For SO\_BULKMODE, *optval* should point to an ibm\_bulkmode\_struct, which is defined in SOCKET.H. The ibm\_bulkmode\_struct contains the following fields:

Element	Description
b_onoff	1 means bulk mode is on; 0 means bulk mode is off.
b_max_receive_queue_size	The maximum receiving queue size in bytes.

b_max_send_queue_size	The maximum sending queue size in bytes. This value is set to zero, since send-type socket calls are not currently supported for bulk mode.
b_move_data	For outbound sockets, if b_move_data is nonzero, the data is moved into buffers in the queue. The client's buffers can be reused right away. If b_move_data is zero, pointers to the data are saved in the queue. The buffers should not be reused until the queue has been flushed (generally by issuing an ibmsflush()).
b_teststor	If this element is nonzero, the message buffer address and the message buffer are checked for addressability during each socket call. errno is set to EFAULT if either address or buffer cannot be addressed. If this element is zero, no checking is performed.
b_max_send_queue_size_avail	The maximum send queue size in bytes that can be set for the b_max_send_queue_size field of ibm_bulkmode_struct. This value will be set to zero, since send-type socket calls are not currently supported for bulk mode.
b_num_UNITS_sent	The number of actual UNITS issued in sending datagrams to TCP/IP. This value will be set to zero, since send-type socket calls are not currently supported for bulk mode.
b_num_UNITS_received	The number of actual UNITS issued in receiving datagrams from TCP/IP.

The fields b\_num\_UNITS\_sent and b\_num\_UNITS\_received represent cumulative totals for this socket since the time the application was started.

For SO\_NONBLOCKLOCAL, *optval* should point to an integer. getibmssockopt() returns 0 in *optval* if the socket is in blocking mode, and returns 1 in *optval* if the socket is in nonblocking mode.

For SO\_IGNOREINCOMINGPUSH, *optval* should point to an integer. getibmssockopt() returns 0 in *optval* if the option is not set, and returns 1 in *optval* if the option is set.

### Returned Value

The value 0 indicates success. The value -1 indicates an error. The value of errno indicates the specific error.

Errno Code	Description
EBADF	The s parameter is not a valid socket descriptor (outside the range of descriptors as specified with maxdesc() ).
EFAULT	Using optval and optlen parameters would result in an attempt to access storage outside the caller's address space.
ENOPROTOPT	The optname parameter is unrecognized, or the level parameter is not SOL_SOCKET.

## Example

The following is an example of the `getibmssockopt()` call.

```
#include <stdio.h>
#include <sys/socket.h>

{ struct ibm_bulkmode_struct bulkstr;
  int optlen, rc, s;
  FILE *stream;

  /* Create, bind, etc done for socket s */
  .
  .
  .
  optlen = sizeof(bulkstr);
  rc = getibmssockopt(s, SOL_SOCKET, SO_BULKMODE, (char *) &bulkstr, &optlen);
  if (rc < 0)
  { tcperror("on getibmssockopt()");
    exit(-1);
  }
  fprintf(stream,"%d byte buffer available for outbound queue.\n",
          bulkstr.b_max_send_queue_size_avail);
}
```

## Related Information

- “sys/socket.h” on page 48
- “ibmsflush() — Flush the Application-side Datagram Queue” on page 652
- “setibmssockopt() — Set Options Associated with a Bulk Mode Socket” on page 1227

## \_\_getipc() — Query Interprocess Communications

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE
#include <sys/__getipc.h>
```

```
int __getipc(int token_id, IPCQPROC *bufptr, size_t buflen, int cmd);
```

### General Description

The \_\_getipc() function provides means for obtaining information about the status of interprocess communications (IPC) resources, message queues, semaphores, and shared memory.

The argument, *token\_id*, is a number that identifies the relative position of an IPC member in the system or specifies a message queue ID, semaphore ID, or shared memory ID. Zero represents the first IPC member ID in the system. On the first call to \_\_getipc(), pass the a *token\_id* of zero; the function will return the token that identifies the next IPC resource to which the caller has access. Use this token on the next call to \_\_getipc().

The argument, *bufptr*, is the address where the data is to be stored.

The argument, *buflen*, is the length of the buffer.

The argument, *cmd*, specifies one of the following commands:

IPCQALL	Retrieve the next shared memory, semaphore, or message queue
IPCQMSG	Retrieve the next message member
IPCQSEM	Retrieve the next semaphore member
IPCQSHM	Retrieve the next shared memory member
IPCQOVER	Overview of system variables. Ignores the value of the first argument, <i>token_id</i> .

### Returned Value

If successful, \_\_getipc() returns zero.

If unsuccessful, \_\_getipc() returns -1, and returns the error value in errno. The following are the possible values of errno:

#### EACCES

Operation permission (read) is denied to the calling process for the member ID specified by *token\_id*.

EINVAL The member ID specified in the argument, *token\_id*, is not valid for the command specified, or the argument, *cmd*, is not a valid command.

#### EFAULT

The argument *bufptr* contains an invalid address.



**Related Information**

- “sys/\_\_getipc.h” on page 47
- “sys/ipc.h” on page 47
- “msgctl() — Message Control Operations” on page 845
- “msgget() — Get Message Queue” on page 847
- “msgrcv() — Message Receive Operation” on page 850
- “msgsnd() — Message Send Operations” on page 852
- “msgxrcv() — Extended Message Receive Operation” on page 854
- “semctl() — Semaphore Control Operations” on page 1169
- “semget() — Get a Set of Semaphores” on page 1172
- “semop() — Semaphore Operations” on page 1175
- “shmat() — Shared Memory Attach Operation” on page 1285
- “shmctl() — Shared Memory Control Operations” on page 1287
- “shmdt() — Shared Memory Detach Operation” on page 1289
- “shmget() — Get a Shared Memory Segment” on page 1290

## getitimer() — Get Value of an Interval Timer

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/time.h>
```

```
int getitimer(int which, struct itimerval * value);
```

### General Description

getitimer() gets the current value of an (previously set) interval timer. An interval timer is a timer which sends a signal after each repetition (interval) of time.

The *which* argument indicates what kind of time is being controlled. Values for *which* are:

ITIMER\_REAL      This timer is marking real (clock) time. A SIGALRM signal is generated after each interval of time.

**Note:** alarm() also sets the real interval timer.

ITIMER\_VIRTUAL      This timer is marking process virtual time. Process virtual time is the amount of time spent while executing in the process, and can be thought of as a CPU timer. A SIGVTALRM signal is generated after each interval of time.

ITIMER\_PROF      This timer is marking process virtual time plus time spent while the system is running on behalf of the process. A SIGPROF signal is generated after each interval of time.

**Note:** In a multi-threaded environment, each of the above timers is specific to a thread of execution for both the generation of the time interval and the measurement of time. For example, an ITIMER\_VIRTUAL timer will mark execution time for just the thread, not the entire process.

The *value* argument is a pointer to a structure containing:

```
it_interval      timer interval
it_value          current timer value (time remaining)
```

Each of these fields is a timeval structure, and contains:

```
tv_sec            seconds since January 1, 1970
tv_usec           microseconds
```

### Returned Value

If successful, getitimer() returns zero, and *value* points to the itimerval structure.

If unsuccessful, getitimer() returns -1, and returns the error value in errno. The following are the possible values of errno:

EINVAL          *which* is not a valid timer type.

**Related Information**

- “sys/time.h” on page 49
- “setitimer() — Set Value of an Interval Timer” on page 1232
- “gettimeofday() — Get Date and Time” on page 616
- “alarm() — Set an Alarm” on page 102
- “sleep() — Suspend Execution of a Thread” on page 1364
- “ualarm() — Set the Interval Timer” on page 1640
- “usleep() — Suspend Execution for an Interval” on page 1663

## getlogin() — Get the User Login Name

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

#### **\_POSIX\_SOURCE**

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
char *getlogin(void);
```

#### **\_XOPEN\_SOURCE**

```
#define _XOPEN_SOURCE
#include <unistd.h>
```

```
char *getlogin(void);
```

### General Description

Finds the name that the login process associated with the current terminal. This string is stored in a static data area and, therefore, may be overwritten with every call to `getlogin()`.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Special Behavior for \_POSIX\_SOURCE

If called from a batch program, a TSO command, or a shell command, `getlogin()` returns the MVS user name associated with the program. With OS/390 UNIX services, this name is a TSO/E user ID. When `_POSIX_SOURCE` is defined and `_XOPEN_SOURCE` is not defined, then `getlogin()` is the same as `__getlogin1()`.

### Special Behavior for XPG4.2

You must have a TTY at file descriptor 0, 1, or 2, and the TTY must be recorded in the `/etc/utmpx` database. Someone must have logged in using the TTY. Also, the program must be invoked from a shell session, and file descriptors 0, 1, and 2 are not all redirected.

If `getlogin()` cannot determine the login name, you can call `getuid()` to get the user ID of the process, and then call `getpwuid()` to get a login name associated with that user ID. `getpwuid()` always returns the `passwd` struct for the same user, even if multiple users have the same UID.

## Returned Value

If successful, `getlogin()` returns a pointer to a string that has the login name for the current terminal.

## Special Behavior for `_POSIX_SOURCE`

If unsuccessful, `getlogin()` returns the NULL pointer. There are no documented errors for this function.

## Special Behavior for XPG4.2

If unsuccessful, `getlogin()` returns a null pointer and sets `errno` to indicate the error. The following are the possible values of `errno`:

EMFILE	{OPEN_MAX} file descriptors are currently open in the calling process.
ENFILE	The maximum allowable number of files is currently open in the system.
ENXIO	The calling process has no controlling terminal.

## Example CBC3BG12

```
/* CBC3BG12
   This example gets the user login name.
*/
#define _POSIX_SOURCE
#include <unistd.h>
#include <stdio.h>

main() {
    char *user;

    if ((user = getlogin()) == NULL)
        perror("getlogin() error");
    else printf("getlogin() returned %s\n", user);
}
```

## Output

`getlogin()` returned MEGA

## Related Information

- “`unistd.h`” on page 53
- “`__getlogin1()` — Get the User Login Name” on page 552
- “`getpwuid()` — Access the User Database by User ID” on page 589
- “`getuid()` — Get the Real User ID” on page 618

## \_\_getlogin1() — Get the User Login Name

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	

### Format

#### **\_POSIX\_SOURCE**

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
char *__getlogin1(void);
```

### General Description

Finds the name that the login process associated with the current terminal. If called from batch, \_\_getlogin1() finds the name associated with the batch program. With OS/390 UNIX services, this name is a TSO/E user ID. This string is stored in a static data area and, therefore, may be overwritten with every call to \_\_getlogin1().

If \_\_getlogin1() cannot determine the login name, you can call getuid() to get the user ID of the process, and then call getpwuid() to get a login name associated with that user ID. getpwuid() always returns the passwd struct for the same user, even if multiple users have the same UID.

### Returned Value

If successful, \_\_getlogin1() returns a pointer to a string that has the login name for the current terminal. If unsuccessful, it returns the NULL pointer. There are no documented errno values for this function.

### Example

#### **CBC3BG12**

```
/* CBC3BG12
   This example gets the user login name.
*/
#define _POSIX_SOURCE
#include <unistd.h>
#include <stdio.h>

main() {
    char *user;

    if ((user = __getlogin1()) == NULL)
        perror("__getlogin1() error");
    else printf("__getlogin1() returned %s\n", user);
}
```

### Output

```
getlogin() returned MEGA
```

## **Related Information**

- “unistd.h” on page 53
- “getlogin() — Get the User Login Name” on page 550
- “getpwuid() — Access the User Database by User ID” on page 589
- “getuid() — Get the Real User ID” on page 618

## getmccoll() — Get Next Collating Element from String

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <collate.h>
```

```
coll_t getmccoll(char **src);
```

### General Description

If the object pointed to by *src* is not a null pointer, the `getmccoll()` library function determines the longest sequence of bytes in the array pointed to by *src* that constitute a valid multi-character collating element. It then produces the value of type `coll_t` corresponding to that collating element. The object pointed to by *src* is assigned the address just past the last byte of the multi-character collating element processed.

### Returned Value

If the object pointed to by *src* is a null pointer, or if it points to null character, the function returns the value 0. Otherwise, it returns the value of type `coll_t` that represents the collating element found.

### Related Information

- “collate.h” on page 23
- “cclass() — Return Characters in a Character Class” on page 149
- “collequiv() — Return a List of Equivalent Collating Elements” on page 200
- “collorder() — Return List of Collating Elements” on page 202
- “collrange() — Calculate the Range List of Collating Elements” on page 204
- “colltostr() — Return a String for a Collating Element” on page 206
- “getwmccoll() — Get Next Collating Element from Wide String” on page 632
- “ismccollet() — Identify a Multi-Character Collating Element” on page 710
- “maxcoll() — Return Maximum Collating Element” on page 788
- “strtocoll() — Return Collating Element for String” on page 1454



# getmsg() - getpmsg() — Receive Next Message from a STREAMS File

## Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

## Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stropts.h>

int getmsg(int fildes, struct strbuf ctlptr,
           struct strbuf dataptr, int *flagsp);
int getpmsg(int fildes, struct strbuf ctlptr,
            struct strbuf dataptr, int *bandp, int *flagsp);
```

## General Description

The `getmsg()` function retrieves the contents of a message located at the head of the STREAM head read queue associated with a STREAMS file and places the contents into one or more buffers. The message contains either a data part, a control part, or both. The data and control parts of the message are placed into separate buffers, as described below. The semantics of each part is defined by the originator of the message.

The `getpmsg()` function does the same thing as `getmsg()`, but provides finer control over the priority of the messages received. Except where noted, all requirements on `getmsg()` also pertain to `getpmsg()`.

The *fildes* argument specifies a file descriptor referencing a STREAMS-based file.

The *ctlptr* and *dataptr* arguments each point to a **strbuf** structure, in which the **buf** member points to a buffer in which the data or control information is to be placed, and the **maxlen** member indicates the maximum number of bytes this buffer can hold. On return, the **len** member contains the number of bytes of data or control information actually received. The **len** member is set to 0 if there is a zero-length control or data part and **len** is set to -1 if no data or control information is present in the message.

When `getmsg()` is called, *flagsp* should point to an integer that indicates the type of message the process is able to receive. This is described further below.

The *ctlptr* argument is used to hold the control part of the message, and *dataptr* is used to hold the data part of the message. If *ctlptr* (or *dataptr*) is a null pointer or the **maxlen** member is -1, the control (or data) part of the message is not processed and is left on the STREAM head read queue. If the *ctlptr* (or *dataptr*) is not a null pointer, **len** is set to -1. If the **maxlen** member is set to 0 and there is a zero-length control (or data) part, that zero-length part is removed from the read queue and **len** is set to 0. If the **maxlen** member is set to 0 and there are more than 0 bytes of control (or data) information, that information is left on the read queue and **len** is set to 0. If the **maxlen** member in *ctlptr* (or *dataptr*) is less than the control (or data) part of the message, **maxlen** bytes are retrieved. In this case, the remainder of the message is left on the STREAM head read queue and a nonzero return value is provided.

By default, `getmsg()` processes the first available message on the STREAM head read queue. However, a process may choose to retrieve only high-priority messages by setting the integer pointed to by *flagsp* to `RS_HIPRI`. In this case, `getmsg()` and `getpmsg()` will only process the next message if it is a high-priority message. When the integer pointed to by *flagsp* is 0, any message will be retrieved. In this case, on return, the integer pointed to by *flagsp* will be set to `RS_HIPRI` if a high-priority message was retrieved, or 0 otherwise.

For `getpmsg()`, the flags are different. The *flagsp* argument points to a bitmask with the following mutually-exclusive flags defined: `MSG_HIPRI`, `MSG_BAND`, and `MSG_ANY`. Like `getmsg()`, `getpmsg()` processes the first available message on the STREAM head read queue. A process may choose to retrieve only high-priority messages by setting the integer pointed to by *flagsp* to `MSG_HIPRI` and the integer pointed to by *bandp* to 0. In this case, `getpmsg()` will only process the next message if it is a high-priority message. In a similar manner, a process may choose to retrieve a message from a particular priority band by setting the integer pointed to by *flagsp* to `MSG_BAND` and the integer pointed to by *bandp* to the priority band of interest. In this case, `getpmsg()` will only process the next message if it is in a priority band equal to, or greater than, the integer pointed to by *bandp*, or if it is a high-priority message. If a process just wants to get the first message off the queue, the integer pointed to by *flagsp* should be set to `MSG_ANY` and the integer pointed to by *bandp* should be set to 0. On return, if the message retrieved was a high-priority message, the integer pointed to by *flagsp* will be set to `MSG_HIPRI` and the integer pointed to by *bandp* will be set to 0. Otherwise, the integer pointed to by *flagsp* will be set to `MSG_BAND` and the integer pointed to by *bandp* will be set to the priority band of the message.

If `O_NONBLOCK` is not set, `getmsg()` and `getpmsg()` will block until a message of the type specified by *flagsp* is available at the front of the STREAM head read queue. If `O_NONBLOCK` is set and a message of the specified type is not present at the front of the read queue, `getmsg()` and `getpmsg()` fail and set `errno` to `EAGAIN`.

If a hang-up occurs on the STREAM from which messages are to be retrieved, `getmsg()` and `getpmsg()` continue to operate normally, as described above, until the STREAM head read queue is empty. Thereafter, they return 0 in the *len* members of *ctlptr* and *dataptr*.

The following symbolic constants are defined under `_XOPEN_SOURCE_EXTENDED 1` in `<stropts.h>`.

<code>MSG_ANY</code>	Receive any message.
<code>MSG_BAND</code>	Receive message from specified band.
<code>MSG_HIPRI</code>	Send/Receive high priority message.
<code>MORECTL</code>	More control information is left in message.
<code>MOREDATA</code>	More data is left in message.

## Returned Value

Upon successful completion, `getmsg()` and `getpmsg()` return a non-negative value. A value of 0 indicates that a full message was read successfully. A return value of `MORECTL` indicates that more control information is waiting for retrieval. A return value of `MOREDATA` indicates that more data is waiting for retrieval. A return value of the bitwise logical OR of `MORECTL` and `MOREDATA` indicates that both types of information remain. Subsequent `getmsg()` and `getpmsg()` calls retrieve the remainder of

the message. However, if a message of higher priority has come in on the STREAM head read queue, the next call to `getmsg()` or `getpmsg()` retrieves that higher-priority message before retrieving the remainder of the previous message.

Upon failure, `getmsg()` and `getpmsg()` return -1 and set `errno` to indicate the error.

**Note:** OS/390 UNIX services do not supply any STREAMS devices or pseudodevices. It is impossible for `getmsg()` and `getpmsg()` to get a message from a STREAMS file. It will always return -1 with `errno` set to indicate the failure. See “`open()` — Open a File” on page 872 for more information.

The `getmsg()` and `getpmsg()` functions will fail if:

EAGAIN	The <code>O_NONBLOCK</code> flag is set and no messages are available.
EBADF	The <i>fildev</i> argument is not a valid file descriptor open for reading.
EBADMSG	The queued message to be read is not valid for <code>getmsg()</code> or <code>getpmsg()</code> or a pending file descriptor is at the STREAM head.
EINTR	A signal was caught during <code>getmsg()</code> or <code>getpmsg()</code>
EINVAL	An illegal value was specified by <i>flagsp</i> , or the STREAM or multiplexer referenced by <i>fildev</i> is linked (directly or indirectly) downstream from a multiplexer.
ENOSTR	A STREAM is not associated with <i>fildev</i> .

In addition, `getmsg()` and `getpmsg()` will fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of `errno` does not reflect the result of `getmsg()` or `getpmsg()` but reflects the prior error.

### Related Information

- “`stropts.h`” on page 46
- “`poll()` — Monitor Activity on File Descriptors and Message Queues” on page 910
- “`putmsg()` - `putpmsg()` — Send a Message on a STREAM” on page 1056
- “`read()` — Read From a File or Socket” on page 1080
- “`write()` — Write Data on a File or Socket” on page 1780

## getnetbyaddr() — Get a Network Entry by Address

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct netent *getnetbyaddr(ip_addr_t net, int type);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <netdb.h>

struct netent *getnetbyaddr(unsigned long net, int type);
```

### General Description

The `getnetbyaddr()` call searches the `tcpip.HOSTS.ADDRINFO` data set for the specified network address.

Parameter	Description
<i>net</i>	The network address.
<i>type</i>	The address domain supported (AF_INET).

If the name server is not present or the `RESOLVE_VIA_LOOKUP` option is in effect, you can use the **X\_ADDR** environment variable to specify a data set other than `tcpip.HOSTS.ADDRINFO`. For more information on these data sets and environment variables, see *OS/390 UNIX System Services Planning*.

**Note:** `tcpip.HOSTS.LOCAL`, `tcpip.HOSTS.ADDRINFO`, and `tcpip.HOSTS.SITEINFO` are described in *TCP/IP for MVS: Customization and Administration Guide*.

`getnetbyaddr()`, `getnetbyname()`, and `getnetent()` all use the same static area to return the **netent** structure. This static area is only valid until the next one of these functions is called on the same thread.

The **netent** structure is defined in the **netdb.h** include file and contains the following elements:

Element	Description
<i>n_addrtype</i>	The type of network address returned. The call always sets this value to AF_INET.
<i>n_aliases</i>	An array, terminated with a NULL pointer, of alternative names for the network.
<i>n_name</i>	The official name of the network.
<i>n_net</i>	The network number, returned in host byte order.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

The return value points to data that is overwritten by subsequent calls returning the same data structure. A pointer to a **netent** structure indicates success. A NULL pointer indicates an error or end-of-file.

### Related Information

- “endhostent() — Work with a Host Entry” on page 308
- “endnetent() — Close Network Information Data Sets” on page 309
- “getnetbyname() — Get a Network Entry by Name” on page 560
- “getnetent() — Get the Next Network Entry” on page 562
- “setnetent() — Open the Network Information Data Set” on page 1251

## getnetbyname() — Get a Network Entry by Name

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct netent *getnetbyname(const char *name);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <netdb.h>

struct netent *getnetbyname(name);
```

### General Description

The `getnetbyname()` call searches the `tcpip.HOSTS.SITEINFO` data set for the specified network name.

Parameter	Description
<i>name</i>	The pointer to a network name.

You can use the **X\_SITE** environment variable to specify a data set other than `tcpip.HOSTS.SITEINFO`. For more information on these data sets and environment variables, see *OS/390 UNIX System Services Planning*.

**Note:** `tcpip.HOSTS.LOCAL`, `tcpip.HOSTS.SITEINFO`, and `tcpip.HOSTS.SITEINFO` are also described in *TCP/IP for MVS: Customization and Administration Guide*.

The `getnetbyname()` call returns a pointer to a **netent** structure for the network name specified on the call. `getnetbyaddr()`, `getnetbyname()`, and `getnetent()` all use the same static area to return the **netent** structure. This static area is only valid until the next one of these functions is called on the same thread.

The **netent** structure is defined in the **netdb.h** include file and contains the following elements:

Element	Description
<i>n_addrtype</i>	The type of network address returned. The call always sets this value to <code>AF_INET</code> .
<i>n_aliases</i>	An array, terminated with a NULL pointer, of alternative names for the network.
<i>n_name</i>	The official name of the network.
<i>n_net</i>	The network number, returned in host byte order.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

The return value points to static data that is overwritten by subsequent calls. A pointer to a **netent** structure indicates success. A NULL pointer indicates an error or end-of-file.

### Related Information

- “endhostent() — Work with a Host Entry” on page 308
- “endnetent() — Close Network Information Data Sets” on page 309
- “getnetbyaddr() — Get a Network Entry by Address” on page 558
- “getnetent() — Get the Next Network Entry” on page 562
- “setnetent() — Open the Network Information Data Set” on page 1251

## getnetent() — Get the Next Network Entry

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>
```

```
struct netent *getnetent(void);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <netdb.h>
```

```
struct netent *getnetent(void);
```

### General Description

The `getnetent()` call reads the next entry of the `tcpip.HOSTS.ADDRINFO` data set.

You can use the **X\_ADDR** environment variable to specify a data set other than `tcpip.HOSTS.ADDRINFO`. For more information on these data sets and environment variables, see *OS/390 UNIX System Services Planning*.

**Note:** `tcpip.HOSTS.LOCAL`, `tcpip.HOSTS.ADDRINFO`, and `tcpip.HOSTS.SITEINFO` are described in *TCP/IP for MVS: Customization and Administration Guide*.

The `getnetent()` call returns a pointer to the next entry in the `tcpip.HOSTS.SITEINFO` data set.

`getnetbyaddr()`, `getnetbyname()`, and `getnetent()` all use the same static area to return the **netent** structure. This static area is only valid until the next one of these functions is called on the same thread.

The **netent** structure is defined in the **netdb.h** include file and contains the following elements:

Element	Description
<i>n_addrtype</i>	The type of network address returned. The call always sets this value to <code>AF_INET</code> .
<i>n_aliases</i>	An array, terminated with a NULL pointer, of alternative names for the network.
<i>n_name</i>	The official name of the network.
<i>n_net</i>	The network number, returned in host byte order.

### Special Behavior for C++



To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

The return value points to data that is overwritten by subsequent calls returning the same data structure. A pointer to a **netent** structure indicates success. A NULL pointer indicates an error or end-of-file.

### Related Information

- “endhostent() — Work with a Host Entry” on page 308
- “endnetent() — Close Network Information Data Sets” on page 309
- “gethostbyaddr() — Get a Host Entry by Address” on page 531
- “gethostbyname() — Get a Host Entry by Name” on page 534
- “setnetent() — Open the Network Information Data Set” on page 1251

## getopt() — Command Option Parsing

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <stdio.h>
```

```
int getopt(int argc, char * const argv[], const char *optstring);
```

```
extern char *optarg;
extern int optind, opterr, optopt;
```

### General Description

The `getopt()` function is a command-line parser that can be used by applications that follow Utility Syntax Guidelines 3, 4, 5, 6, 7, 9 and 10 in the *X/Open CAE Specification, System Interface Definitions, Issue 4, Version 2* Section 10.2, Utility Syntax Guidelines. The `getopt()` function provides the identical functionality described in the X/Open CAE Specification System Interfaces and Headers, Issue 4, Version 2 for the `getopt()` function with the following extensions:

- If the external variable `optind` is set to zero, the `getopt()` function treats this as an indication to restart the scan at the first byte of `argv[1]`.

If `getopt()` encounters an option character that is not contained in `optstring`, it returns the question-mark (?) character. If it detects a missing option-argument, it returns the colon character (:) if the first character of `optstring` was a colon, or a question-mark character (?) otherwise. In either case, `getopt()` sets the variable `optopt` to the option character that caused the error. If the application has not set the variable `opterr` to 0 and the first character of `optstring` is not a colon, `getopt()` also prints a diagnostic message to `stderr` in the format specified for the `getopts` utility.

Because the `getopt()` function returns thread specific data the `getopt()` function can be used safely from a multi-threaded application.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

If successful, `getopt()` returns the value of the next option character from `argv` that matches a character in `optstring`.

A colon (:) is returned if `getopt()` detects a missing argument and the first character of `optstring` was a colon (:).

A question-mark (?) is returned if `getopt()` encounters an option character not in `optstring` or detects a missing argument and the first character of `optstring` was not a colon (:).

Otherwise getopt() returns -1 when all command line arguments have been parsed or an unexpected error is encountered in the command line.

getopt() sets the external variables optind, optarg and optopt as described in the X/Open CAE Specification System Interfaces and Headers, Issue 4, Version 2 for the getopt() function.

The following functions defined in <stdio.h> should be used by multi-threaded applications when attempting to reference or change the optind, optopt, optarg and opterr external variables:

```
int *__opindf(void);  
  
int *__opoptf(void);  
  
char **__opargf(void);  
  
int *__operrf(void);
```

Also use these functions when you invoke getopt() in a DLL. These functions return a pointer to a thread specific value for each variable.

getopt() does not return any errno values.

If getopt() detects a missing argument or an option character not in optstring it will write an error message to stderr describing the option character in error and the invoking program.

### **Related Information**

- “getsubopt() — Parse Suboption Arguments” on page 612

## getpagesize() — Get the Current Page Size

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
```

```
int getpagesize(void);
```

### General Description

The `getpagesize()` function returns the current page size. The `getpagesize()` function is equivalent to `sysconf(_SC_PAGE_SIZE)` and `sysconf(_SC_PAGESIZE)`.

### Returned Value

The `getpagesize()` function returns the current page size.

### Related Information

- “`unistd.h`” on page 53
- “`sysconf()` — Determine System Configuration Options” on page 1483

## getpass() — Read a String of Characters Without Echo

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE
#include <unistd.h>
```

```
char *getpass(const char *prompt);
```

### General Description

The `getpass()` function opens the process' controlling terminal, writes to that device the null-terminated string *prompt*, disables echoing, reads a string of characters up to the next newline character or EOF, restores the terminal state and closes the terminal.

`getpass()` only works in an environment where either a controlling terminal exists, or `stdin` and `stderr` refer to tty devices. Specifically, it does not work in a TSO environment.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

**Note:** The `getpass()` function is marked to be withdrawn from the X/Open standard.

### Returned Value

Upon successful completion, `getpass()` returns a pointer to a null terminated string of at most `{PASS_MAX}` bytes that were read from the terminal device. If an error is encountered, the terminal state is restored and a null pointer is returned.

### Related Information

- “`unistd.h`” on page 53

## getpeername() — Get the Name of the Peer Connected to a Socket

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>
```

```
int getpeername(int socket, struct sockaddr *address, size_t *address_len);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>
```

```
int getpeername(int socket, struct sockaddr *name, int namelen);
```

### General Description

The `getpeername()` call returns the name of the peer connected to socket descriptor *socket*. *namelen* must be initialized to indicate the size of the space pointed to by *name* and is set to the number of bytes copied into the space before the call returns. The size of the peer name is returned in bytes. If the actual length of the address is greater than the length of the supplied *sockaddr*, the stored address is truncated. The *sa\_len* member of the store structure contains the length of the untruncated address.

#### Parameter Description

Parameter	Description
<i>socket</i>	The socket descriptor.
<i>name</i>	The Internet address of the connected socket that is filled by <code>getpeername()</code> before it returns. The exact format of <i>name</i> is determined by the domain in which communication occurs.
<i>namelen</i>	The size of the address structure pointed to by <i>name</i> in bytes.

#### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

The value 0 indicates success; the value -1 indicates an error. The value of the error code indicates the specific error.

#### Error Code Description

EBADF	The <i>socket</i> parameter is not a valid socket descriptor.
EFAULT	Using the <i>name</i> and <i>namelen</i> parameters as specified would result in an attempt to access storage outside of the caller's address space.

EINVAL	The <i>namelen</i> parameter is not a valid length. The socket has been shut down.
ENOBUFS	getpeername() is unable to process the request due to insufficient storage.
ENOTCONN	The socket is not in the connected state.
ENOTSOCK	The descriptor is for a file, not for a socket.
EOPNOTSUPP	The operation is not supported for the socket protocol.

**Related Information**

- “accept() — Accept a New Connection on a Socket” on page 75
- “connect() — Connect a Socket” on page 214
- “getsockname() — Get the Name of a Socket” on page 604
- “socket() — Create a Socket” on page 1371

## getpgid() — Get Process Group ID

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
pid_t getpgid(pid_t pid);
```

### General Description

The `getpgid()` function returns the process group ID of the process whose process ID is equal to *pid*. If *pid* is 0, `getpgid()` returns the PID of the calling process.

### Returned Value

Upon successful completion, `getpgid()` returns a process group ID. Otherwise, it returns (pid\_t) -1 and sets `errno` to indicate the error.

The `getpgid()` function will fail if:

- EPERM** The process whose process ID is equal to *pid* is not the same session as the calling process, and the implementation does not allow to the process group ID of that process from the calling process.
- ESRCH** There is no process with a process ID equal to *pid*.

The `getpgid()` function may fail if:

- EINVAL** The value of the *pid* argument is invalid.

### Related Information

- “exec Functions” on page 322
- “fork() — Create a New Process” on page 422
- “getpgrp() — Get the Process Group ID” on page 571
- “getsid() — Get Process Group ID of Session Leader” on page 603
- “setregid() — Set Real and Effective Group IDs” on page 1261
- “setsid() — Create Session, Set Process Group ID” on page 1268



## getpgrp() — Get the Process Group ID

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
pid_t getpgrp(void);
```

### General Description

Finds the process group ID of the calling process.

### Returned Value

Returns the found value. It is always successful. There are no documented errors for this function.

### Example CBC3BG13

```
/* CBC3BG13
   This example gets all the process group IDs.
*/
#define _POSIX_SOURCE
#include <unistd.h>
#include <wait.h>

main() {
    int status;

    if (fork() == 0) {
        if (fork() == 0) {
            printf("grandchild's pid is %d, process group id is %d\n",
                (int) getpid(), (int) getpgrp());
            exit(0);
        }
        printf("child's pid is %d, process group id is %d\n",
            (int) getpid(), (int) getpgrp());
        wait(&status);
        exit(0);
    }
    printf("parent's pid is %d, process group id is %d\n",
        (int) getpid(), (int) getpgrp());
    printf("the parent's parent's pid is %d\n", (int) getppid());
    wait(&status);
}
```

### Output

```
parent's pid is 5373959, process group id is 5111816
the parent's parent's pid is 5111816
child's pid is 5832710, process group id is 5111816
grandchild's pid is 196617, process group id is 5111816
```

**Related Information**

- “sys/types.h” on page 49
- “setpgid() — Set Process Group ID for Job Control” on page 1254
- “setsid() — Create Session, Set Process Group ID” on page 1268

## getpid() — Get the Process ID

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
pid_t getpid(void);
```

### General Description

Finds the process ID (PID) of the calling process.

### Returned Value

Returns the found value. It is always successful. There are no documented ernos for this function.

### Example CBC3BG14

```
/* CBC3BG14 */
#define _POSIX_SOURCE
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>

void catcher(int signum) {
    puts("catcher has control!");
}

main() {
    struct sigaction sact;

    sigemptyset(&sact.sa_mask);
    sact.sa_flags = 0;
    sact.sa_handler = catcher;
    sigaction(SIGUSR1, &sact, NULL);

    printf("sending SIGUSR1 to pid %d\n", (int) getpid());
    kill(getpid(), SIGUSR1);
}
```

### Output

```
sending SIGUSR1 to pid 5570567
catcher has control!
```

### **Related Information**

- “sys/types.h” on page 49
- “exec Functions” on page 322
- “fork() — Create a New Process” on page 422
- “getppid() — Get the Parent Process ID” on page 576
- “kill() — Send a Signal to a Process” on page 728

**getpmsg() — Receive Next Message from a STREAMS File**

The information for this function is included in “getmsg() - getpmsg() — Receive Next Message from a STREAMS File” on page 555.

## getppid() — Get the Parent Process ID

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
pid_t getppid(void);
```

### General Description

Gets the parent process ID (PPID).

### Returned Value

Returns the parent process ID. It is always successful. There are no documented errno's for this function.

### Example CBC3BG15

```
/* CBC3BG15 */
#define _POSIX_SOURCE
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <wait.h>

volatile short footprint=0;

void catcher(int signum) {
    switch (signum) {
        case SIGALRM: puts("caught SIGALRM");
                     break;
        case SIGUSR2: puts("caught SIGUSR2");
                     break;
        default: printf("caught unexpected signal %d\n", signum);
    }
    footprint++;
}

main() {
    struct sigaction sact;
    int status;

    sigemptyset(&sact.sa_mask);
    sact.sa_flags = 0;
    sact.sa_handler = catcher;
    sigaction(SIGUSR2, &sact, NULL);

    sigemptyset(&sact.sa_mask);
    sact.sa_flags = 0;
    sact.sa_handler = catcher;
    sigaction(SIGALRM, &sact, NULL);
```

```

printf("parent (pid %d) is about to fork child\n", (int) getpid());

if (fork() == 0) {
    printf("child is sending SIGUSR2 to pid %d\n", (int) getppid());
    kill(getppid(), SIGUSR2);
    exit(0);
}

alarm(30);
while (footprint == 0);
wait(&status);
puts("parent is exiting");
}

```

### Output

```

parent (pid 6094854) is about to fork child
is sending SIGUSR2 to pid 6094854
caught SIGUSR2
parent is exiting

```

### Related Information

- “sys/types.h” on page 49
- “exec Functions” on page 322
- “fork() — Create a New Process” on page 422
- “getpid() — Get the Process ID” on page 573
- “kill() — Send a Signal to a Process” on page 728

## getpriority() — Get Process Scheduling Priority

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/resource.h>
```

```
int getpriority(int which, id_t who);
```

### General Description

getpriority() obtains the current priority of a process, process group or user.

Processes are specified by the values of the *which* and *who* arguments. The *which* argument may be any one of the following set of symbols defined in the *sys/resource.h* include file:

PRIO_PROCESS	indicates that the <i>who</i> argument is to be interpreted as a process ID
PRIO_PGRP	indicates that the <i>who</i> argument is to be interpreted as a process group ID
PRIO_USER	indicates that the <i>who</i> argument is to be interpreted as a user ID

The *who* argument specifies the ID (process, process group, or user). A 0 (zero) value for the *who* argument specifies the current process, process group or user ID.

### Returned Value

If successful, getpriority() returns the priority of the process, process group, or user ID requested in *who*. The priority is returned as an integer in the range -20 to 19 (the lower the numerical value, the higher the priority).

If more than one process is specified, getpriority() returns the highest priority pertaining to any of the specified processes.

If unsuccessful, getpriority() returns -1, and returns the error value in errno. The following are the possible values of errno:

ESRCH	No process could be located using the <i>which</i> and <i>who</i> argument values specified. <i>stufname</i> .
EINVAL	The symbol specified in the <i>which</i> argument was not recognized, or the value of the <i>who</i> argument is not a valid process ID, process group ID or user ID.

Because getpriority() can return the value -1 on successful completion, it is necessary to set the external variable errno to 0 prior to a call to getpriority(). If getpriority() returns the value -1, then errno can be checked to see if an error occurred or if the value is a legitimate priority.



**Related Information**

- “sys/resource.h” on page 48
- “nice() — Change Priority of a Process” on page 864
- “setpriority() — Set Process Scheduling Priority” on page 1257

## getprotobyname() — Get a Protocol Entry by Name

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>
```

```
struct protoent *getprotobyname(const char *name);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <netdb.h>
```

```
struct protoent *getprotobyname(char name);
```

### General Description

The `getprotobyname()` call searches the `/etc/protocol` or `tcpip.ETC.PROTO` data set for the specified protocol name.

Parameter	Description
<i>name</i>	The name of the protocol.

The `getprotobyname()` call returns a pointer to a **protoent** structure for the network protocol specified on the call. `getprotobyname()`, `getprotobynumber()`, and `getprotoent()` all use the same static area to return the **protoent** structure. This static area is only valid until the next one of these functions is called on the same thread.

The **protoent** structure is defined in the **netdb.h** include file and contains the following elements:

Element	Description
<i>p_aliases</i>	An array, terminated with a NULL pointer, of alternative names for the protocol.
<i>p_name</i>	The official name of the protocol.
<i>p_proto</i>	The protocol number.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

**Returned Value**

The return value points to data that is overwritten by subsequent calls returning the same data structure. A pointer to a **protoent** structure indicates success. A NULL pointer indicates an error or end-of-file.

**Related Information**

- “endprotoent() — Work with a Protocol Entry” on page 310
- “getprotobyname() — Get a Protocol Entry by Name” on page 582
- “getprotoent() — Get the Next Protocol Entry” on page 584
- “setprotoent() — Open the Protocol Information Data Set” on page 1259

## getprotobynumber() — Get a Protocol Entry by Number

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>
```

```
struct protoent *getprotobynumber(int proto);
```

#### Berkeley Sockets

```
#define OE_SOCKETS
#include <netdb.h>
```

```
struct protoent *getprotobynumber(int proto);
```

### General Description

The `getprotobynumber()` call searches the `/etc/protocol` or `tcpip.ETC.PROTO` data set for the specified protocol number.

Parameter	Description
<i>proto</i>	The protocol number.

The `getprotobynumber()` call returns a pointer to a **protoent** structure for the network protocol specified on the call. `getprotobyname()`, `getprotobynumber()`, and `getprotoent()` all use the same static area to return the **protoent** structure. This static area is only valid until the next one of these functions is called on the same thread.

The **protoent** structure is defined in the **netdb.h** include file and contains the following elements:

Element	Description
<i>p_aliases</i>	An array, terminated with a NULL pointer, of alternative names for the protocol.
<i>p_name</i>	The official name of the protocol.
<i>p_proto</i>	The protocol number.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

## Returned Value

The return value points to data that is overwritten by subsequent calls returning the same data structure. A pointer to a **protoent** structure indicates success. A NULL pointer indicates an error or end-of-file.

## Related Information

- “endprotoent() — Work with a Protocol Entry” on page 310
- “getprotobyname() — Get a Protocol Entry by Name” on page 580
- “getprotoent() — Get the Next Protocol Entry” on page 584
- “setprotoent() — Open the Protocol Information Data Set” on page 1259

## getprotoent() — Get the Next Protocol Entry

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct protoent *getprotoent(void);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <netdb.h>

struct protoent *getprotoent(void);
```

### General Description

The `getprotoent()` call reads */etc/protocol* or the *tcpip.ETC.PROTO* data set.

The `getprotoent()` call returns a pointer to the next entry in the */etc/protocol* or the *tcpip.ETC.PROTO* data set.

`getprotobyname()`, `getprotobynumber()`, and `getprotoent()` all use the same static area to return the **protoent** structure. This static area is only valid until the next one of these functions is called on the same thread.

The **protoent** structure is defined in the **netdb.h** include file and contains the following elements:

Element	Description
<i>p_aliases</i>	An array, terminated with a NULL pointer, of alternative names for the protocol.
<i>p_name</i>	The official name of the protocol.
<i>p_proto</i>	The protocol number.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

The return value points to data that is overwritten by subsequent calls returning the same data structure. A pointer to a **protoent** structure indicates success. A NULL pointer indicates an error or end-of-file.

**Related Information**

- “endprotoent() — Work with a Protocol Entry” on page 310
- “getprotobyname() — Get a Protocol Entry by Name” on page 580
- “getprotobynumber() — Get a Protocol Entry by Number” on page 582
- “setprotoent() — Open the Protocol Information Data Set” on page 1259

## **getpwent() — Get User Database Entry**

The information for this function is included in “endpwent() — User Database Functions” on page 311.



getpwnam() — Access the User Database by User Name

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

Format

```
#define _POSIX_SOURCE
#include <pwd.h>

struct passwd *getpwnam(const char *name);
```

General Description

Accesses the passwd structure (defined in the pwd.h header file), which contains the following members:

- pw\_name      User name
- pw\_uid      User ID (UID) number
- pw\_gid      Group ID (GID) number
- pw\_dir      Initial working directory
- pw\_shell    Initial user program

|  
|

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

Returned Value

If successful, `getpwnam()` returns a pointer to a `passwd` structure containing an entry from the user database with the specified *name*. If unsuccessful, `getpwnam()` returns a `NULL` pointer and sets `errno` to `EINVAL`, which indicates that the user *name* length is incorrect.

Return values may point to the static data that is overwritten on each call.

Example  
CBC3BG16

```
/* CBC3BG16
   This example provides information for the user database, MEGA.
*/
#define _POSIX_SOURCE
#include <sys/types.h>
#include <pwd.h>

main() {
    struct passwd *p;
    char user[]="MEGA";

    if ((p = getpwnam(user)) == NULL)
        perror("getpwnam() error");
    else {
        printf("getpwnam() returned the following info for user %s:\n",
            user);
        printf(" pw_name : %s\n",      p->pw_name);
    }
}
```

```
    printf(" pw_uid   : %d\n", (int) p->pw_uid);  
    printf(" pw_gid   : %d\n", (int) p->pw_gid);  
    printf(" pw_dir   : %s\n",    p->pw_dir);  
    printf(" pw_shell : %s\n",    p->pw_shell);  
}  
}
```

### Output

```
pw_name  : MEGA  
pw_uid   : 0  
pw_gid   : 512  
pw_dir   : /u/mega  
pw_shell : /bin/sh
```

### Related Information

- “pwd.h” on page 39
- “sys/types.h” on page 49
- “endpwent() — User Database Functions” on page 311
- “getlogin() — Get the User Login Name” on page 550
- “getpwuid() — Access the User Database by User ID” on page 589

## getpwuid() — Access the User Database by User ID

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <pwd.h>
```

```
struct passwd *getpwuid(uid_t uid);
```

### General Description

Gets information about a user with the specified *uid*. `getpwuid()` returns a pointer to a `passwd` structure containing an entry from the user database for the specified *uid*. This structure (defined in the `pwd.h` header file), contains the following members:

```
pw_name    User name
pw_uid     User ID (UID) number
pw_gid     Group ID (GID) number
pw_dir     Initial working directory
pw_shell   Initial user program
```

Return values may point to the static data that is overwritten on each call.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

If successful, `getpwuid()` returns a pointer. If unsuccessful, it returns a `NULL` pointer. There are no documented `errno`s for this function.

### Example

#### CBC3BG17

```
/* CBC3BG17
   This example provides information for user ID 0.
*/
#define _POSIX_SOURCE
#include <sys/types.h>
#include <pwd.h>

main() {
    struct passwd *p;
    uid_t uid=0;

    if ((p = getpwuid(uid)) == NULL)
        perror("getpwuid() error");
    else {
        printf("getpwuid() returned the following info for uid %d:\n",
               (int) uid);
        printf(" pw_name  : %s\n",      p->pw_name);
        printf(" pw_uid   : %d\n", (int) p->pw_uid);
        printf(" pw_gid   : %d\n", (int) p->pw_gid);
        printf(" pw_dir   : %s\n",      p->pw_dir);
    }
}
```

```
        printf(" pw_shell : %s\n",      p->pw_shell);  
    }  
}
```

### Output

getpwuid() returned the following info for uid 0:

```
pw_name  : MEGA  
pw_uid   : 0  
pw_gid   : 512  
pw_dir   : /u/mega  
pw_shell : /bin/sh
```

### Related Information

- “pwd.h” on page 39
- “sys/types.h” on page 49
- “endpwent() — User Database Functions” on page 311
- “getlogin() — Get the User Login Name” on page 550
- “getpwnam() — Access the User Database by User Name” on page 587

## getrlimit() — Control Maximum Resource Consumption

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/resource.h>
```

```
int getrlimit(int resource, struct rlimit *rlp);
```

### General Description

The `getrlimit()` function gets resource limits for the calling process. A resource limit is a pair of values; one specifying the current (soft) limit, the other a maximum (hard) limit.

The value **RLIM\_INFINITY** defined in `<sys/resource.h>`, is considered to be larger than any other limit value. If a call to `getrlimit()` returns **RLIM\_INFINITY** for a resource, it means the implementation does not enforce limits on that resource.

The *resource* argument specifies which resource to get the hard and/or soft limits for, and may be one of the following values:

RLIMIT_CORE	The maximum size of a dump of memory (in bytes) allowed for the process. A value of 0 (zero) prevents file creation. Dump file creation will stop at this limit.
RLIMIT_CPU	The maximum amount of CPU time (in seconds) allowed for the process. If the limit is exceeded, a SIGXCPU signal is sent to the process and the process is granted a small CPU time extension to allow for signal generation and delivery. If the extension is used up, the process is terminated with a SIGKILL signal.
RLIMIT_DATA	The maximum size of the break value for the process, in bytes. In this implementation, this resource always has a hard and soft limit value of <b>RLIM_INFINITY</b> .
RLIMIT_FSIZE	The maximum file size (in bytes) allowed for the process. A value of 0 (zero) prevents file creation. If the size is exceeded, a SIGXFSZ signal is sent to the process. If the process is blocking, catching, or ignoring SIGXFSZ, continued attempts to increase the size of a file beyond the limit will fail with an <code>errno</code> of <code>EFBIG</code> .
RLIMIT_NOFILE	The maximum number of open file descriptors allowed for the process. This number is one greater than the maximum value that may be assigned to a newly created descriptor. (That is, it is one-based.) Any function that attempts to create a new file descriptor beyond the limit will fail with an <code>EMFILE</code> <code>errno</code> .
RLIMIT_STACK	The maximum size of the stack for a process, in bytes. Note that in OS/390 UNIX services, the stack is a per-thread resource. In this implementation, this resource always has a

hard and soft limit value of RLIM\_INFINITY. A call to getrlimit() to set this resource to any value other than RLIM\_INFINITY will fail with an errno of EINVAL.

**RLIMIT\_AS**            The maximum address space size for the process, in bytes. If the limit is exceeded, malloc() and mmap() functions will fail with an errno of ENOMEM. Automatic stack growth will also fail.

The *rlp* argument points to a `rlimit` structure. This structure contains the following members:

`rlim_cur`            The current (soft) limit  
`rlim_max`            The maximum (hard) limit

Refer to the `<sys/resource.h>` header for more detail.

The resource limit values are propagated across exec and fork.

### Special Behavior for OS/390 UNIX Services

An exception exists for exec processing in conjunction with daemon support. If a daemon process invokes exec and it had previously invoked setuid() prior to exec, the RLIMIT\_CPU, RLIMIT\_AS, RLIMIT\_CORE, RLIMIT\_FSIZE, and RLIMIT\_NOFILE limit values are set based on the limit values specified in the kernel parmlib member BPXPRMxx.

For processes which are not the only process within an address space, the RLIMIT\_CPU and RLIMIT\_AS limits are shared with all the processes within the address space. For RLIMIT\_CPU, when the soft limit is exceeded, action will be taken on the first process within the address space. If the action is termination, all processes within the address space will be terminated.

In addition to the RLIMIT\_CORE limit values, the dump file defaults are set by SYSMDUMP defaults. Refer to the *OS/390 MVS Initialization and Tuning Reference* for information on setting up SYSMDUMP defaults via the IEADMR00 parmlib member.

Core dumps are taken in 4160 byte increments. Therefore, RLIMIT\_CORE values affect the size of core dumps in 4160 byte increments. For example, if the RLIMIT\_CORE soft limit value is 4000, the dump will contain no data. If the RLIMIT\_CORE soft limit value is 8000, the maximum size of a core dump is 4160 bytes.

When setting RLIMIT\_NOFILE, the hard limit cannot exceed the system defined limit of 65535.

### Returned Value

Upon successful completion, a 0 is returned. Otherwise, a -1 is returned, and the error value in errno is set. The following are the possible values of errno:

**EINVAL**    An invalid *resource* was specified.

## Related Information

- “sys/resource.h” on page 48
- “stropts.h” on page 46
- “brk() — Change Space Allocation” on page 133
- “fork() — Create a New Process” on page 422
- “getdtablesize() — Get the File Descriptor Table Size” on page 513
- “malloc() — Reserve Storage Block” on page 786
- “open() — Open a File” on page 872
- “rexec() — Execute Commands One at a Time on a Remote Host” on page 1143
- “setrlimit() — Control Maximum Resource Consumption” on page 1264
- “sigaltstack() — Set and/or Get Signal Alternate Stack Context” on page 1314
- “sysconf() — Determine System Configuration Options” on page 1483
- “ulimit() — Get/Set Process File Size Limits” on page 1645

## getrusage() — Get Information About Resource Utilization

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/resource.h>
int getrusage(int who, struct rusage *r_usage);
```

### General Description

The `getrusage()` function provides measures of the resources used by the current process or its terminated and waited-for-child processes. If the value of the *who* argument is `RUSAGE_SELF`, information is returned about resources used by the current process. If the value of the *who* argument is `RUSAGE_CHILDREN`, information is returned about resources used by the terminated and waited-for-children of the current process. If the child is never waited for (for instance, if the parent has `SA_NOCLDWAIT` set or sets `SIGCHLD` to `SIG_IGN`), the resource information for the child process is discarded and not included in the resource information provided by `getrusage()`.

The *r\_usage* argument is a pointer of an object of type `struct rusage` in which the returned information is stored.

### Returned Value

Upon successful completion, `getrusage()` returns 0. Otherwise, -1 is returned, and `errno` is set to indicate the error.

The `getrusage()` function will fail if:

`EINVAL`      The value of the *who* argument is not valid.

### Related Information

- “`exit()` — End Program” on page 330
- “`sigaction()` — Examine or Change a Signal Action” on page 1295
- “`time()` — Determine Current Time” on page 1570
- “`times()` — Get Process and Child Process Times” on page 1572
- “`wait()` — Wait for a Child Process to End” on page 1687



## gets() — Read a String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
char *gets(char *buffer);
```

### General Description

Reads bytes from the standard input stream `stdin`, and stores them in the array pointed to by *buffer*. The line consists of all characters up to and including the first new-line character (`\n`) or EOF. The `gets()` function discards any new-line character, and the null character (`\0`) is placed immediately after the last byte read. If there is an error, the value stored in *buffer* is undefined.

`gets()` is not supported for files opened with `type=record`.

`gets()` has the same restriction as any read operation, such as a read immediately following a write, or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

### Returned Value

If successful, the `gets()` function returns its argument. A NULL pointer returned value indicates an error or an EOF condition with no characters read.

Use `ferror()` or `feof()` to determine which of these conditions occurred. Note that EOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

### Example CBC3BG18

```
/* CBC3BG18
   This example gets a line of input from stdin.
*/
#include <stdio.h>
#define MAX_LINE 100

int main(void)
{
    char line[MAX_LINE];
    char *result;

    printf("Enter string:\n");
    if ((result = gets(line)) != NULL)
        printf("string is %s\n", result);
}
```

```
else
    if (ferror(stdin))
        printf("Error\n");
}
```

**Related Information**

- “stdio.h” on page 43
- “feof() — Test End-of-File Indicator” on page 365
- “ferror() — Test for Read/Write Errors” on page 367
- “fgets() — Read a String from a Stream” on page 392
- “fputs() — Write a String” on page 450
- “puts() — Write a String” on page 1059

# getservbyname() — Get a Server Entry by Name

## Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

## Format

### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>

struct servent *getservbyname(const char *name, const char *proto);
```

### Berkeley Sockets

```
#define _OE_SOCKETS
#include <netdb.h>

struct servent *getservbyname(char *name, char *proto);
```

## General Description

The `getservbyname()` call searches the `/etc/services` or `tcpip.ETC.SERVICES` data set for the first entry that matches the specified service name and protocol name. If `proto` is `NULL`, only the service name must match.

Parameter	Description
<i>name</i>	The service name.
<i>proto</i>	The protocol name.

The `getservbyname()` call returns a pointer to a **servent** structure for the network service specified on the call. `getservbyname()`, `getservbyport()`, and `getservent()` all use the same static area to return the **servent** structure. This static area is only valid until the next one of these functions is called on the same thread.

The **servent** structure is defined in the **netdb.h** include file and contains the following elements:

Element	Description
<i>s_aliases</i>	An array, terminated with a <code>NULL</code> pointer, of alternative names for the service.
<i>s_name</i>	The official name of the service.
<i>s_port</i>	The port number of the service.
<i>s_proto</i>	The protocol required to contact the service.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

## Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

The return value points to data that is overwritten by subsequent calls returning the same data structure. A pointer to a **servent** structure indicates success. A NULL pointer indicates an error or end-of-file.

### Related Information

- “endservent() — Close Network Services Information Data Sets” on page 312
- “getservbyport() — Get a Service Entry by Port” on page 599
- “getservent() — Get the Next Service Entry” on page 601
- “setservent() — Open the Network Services Information Data Set” on page 1267

## getservbyport() — Get a Service Entry by Port

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>
```

```
struct servent *getservbyport(int port, const char *proto);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <netdb.h>
```

```
struct servent *getservbyport(int port, char *proto);
```

### General Description

The `getservbyport()` call searches the `/etc/services` or the `tcpip.ETC.SERVICES` data set for the first entry that matches the specified port number and protocol name. If `proto` is `NULL`, only the port number must match.

Parameter	Description
<i>port</i>	The port number.
<i>proto</i>	The protocol name.

The `getservbyport()` call returns a pointer to a **servent** structure for the port number specified on the call. `getservbyname()`, `getservbyport()`, and `getservent()` all use the same static area to return the **servent** structure. This static area is only valid until the next one of these functions is called on the same thread.

The **servent** structure is defined in the **netdb.h** include file and contains the following elements:

Element	Description
<i>s_aliases</i>	An array, terminated with a <code>NULL</code> pointer, of alternative names for the service.
<i>s_name</i>	The official name of the service.
<i>s_port</i>	The port number of the service.
<i>s_proto</i>	The protocol required to contact the service.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

**Returned Value**

The return value points to data that is overwritten by subsequent calls returning the same data structure. A pointer to a **servent** structure indicates success. A NULL pointer indicates an error or end-of-file.

**Related Information**

- “endservent() — Close Network Services Information Data Sets” on page 312
- “getservbyname() — Get a Server Entry by Name” on page 597
- “getservent() — Get the Next Service Entry” on page 601
- “setservent() — Open the Network Services Information Data Set” on page 1267

## getservent() — Get the Next Service Entry

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>
```

```
struct servent *getservent(void);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <netdb.h>
```

```
struct servent *getservent(void);
```

### General Description

The `getservent()` call reads the next line of the */etc/services* or the *tcpip.ETC.SERVICES* data set.

The `getservent()` call returns a pointer to the next entry in the */etc/services* or the *tcpip.ETC.SERVICES* data set.

`getservbyname()`, `getservbyport()`, and `getservent()` all use the same static area to return the **servent** structure. This static area is only valid until the next one of these functions is called on the same thread.

The **servent** structure is defined in the **netdb.h** include file and contains the following elements:

Element	Description
<i>s_aliases</i>	An array, terminated with a NULL pointer, of alternative names for the service.
<i>s_name</i>	The official name of the service.
<i>s_port</i>	The port number of the service.
<i>s_proto</i>	The protocol required to contact the service.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

The return value points to data that is overwritten by subsequent calls returning the same data structure. A pointer to a **servent** structure indicates success. A NULL pointer indicates an error or end-of-file.

### Related Information

- “endservent() — Close Network Services Information Data Sets” on page 312
- “getservbyname() — Get a Server Entry by Name” on page 597
- “getservbyport() — Get a Service Entry by Port” on page 599
- “setservent() — Open the Network Services Information Data Set” on page 1267



## getsid() — Get Process Group ID of Session Leader

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
pid_t getsid(pid_t pid);
```

### General Description

The `getsid()` function obtains the process group ID of the process that is the session leader of the process specified by *pid*. If *pid* is 0, the system uses the PID of the process calling `getsid()`.

### Returned Value

Upon successful completion, `getsid()` returns the process group ID of the session leader of the specified process. Otherwise, it returns `(pid_t)-1` and sets `errno` to indicate the error.

The `getsid()` function will fail if:

- EPERM** The process specified by *pid* is not in the same session as the calling process, and the implementation does not allow access to the process group ID of the session leader of that process from the calling process.
- ESRCH** There is no process with a process ID equal to *pid*.

### Related Information

- “exec Functions” on page 322
- “fork() — Create a New Process” on page 422
- “getpid() — Get the Process ID” on page 573
- “getppid() — Get the Parent Process ID” on page 576
- “setpgid() — Set Process Group ID for Job Control” on page 1254
- “setsid() — Create Session, Set Process Group ID” on page 1268

## getsockname() — Get the Name of a Socket

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>
```

```
int getsockname(int socket, struct sockaddr *address, size_t *address_len);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>
```

```
int getsockname(int socket, struct sockaddr name, int namelen);
```

### General Description

The `getsockname()` call stores the current name for the socket specified by the *socket* parameter into the structure pointed to by the *name* parameter. It returns the address to the socket that has been bound. If the socket is not bound to an address, the call returns with the family set, and the rest of the structure set to zero. For example, an unbound socket in the Internet domain would cause the name to point to a **sockaddr\_in** structure with the *sin\_family* field set to `AF_INET` and all other fields zeroed.

If the actual length of the address is greater than the length of the supplied *sockaddr*, the stored address is truncated. The *sa\_len* member of the store structure contains the length of the untruncated address.

Parameter	Description
<i>socket</i>	The socket descriptor.
<i>name</i>	The address of the buffer into which <code>getsockname()</code> copies the name of <i>socket</i> .
<i>namelen</i>	Must initially point to an integer that contains the size in bytes of the storage pointed to by <i>name</i> . Upon return, that integer contains the size of the data returned in the storage pointed to by <i>name</i> .

The `getsockname()` call is often used to discover the port assigned to a socket after the socket has been implicitly bound to a port. For example, an application can call `connect()` without previously calling `bind()`. In this case, the `connect()` call completes the binding necessary by assigning a port to the socket. This assignment can be discovered with a call to `getsockname()`.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

## Returned Value

The value 0 indicates success; the value -1 indicates an error. The value of the error code indicates the specific error.

Error Code	Description
EBADF	The <i>socket</i> parameter is not a valid socket descriptor.
EFAULT	Using the <i>name</i> and <i>namelen</i> parameters as specified would result in an attempt to access storage outside of the caller's address space.
ENOBUFS	getsockname() is unable to process the request due to insufficient storage.
ENOTCONN	The socket is not in the connected state.
ENOTSOCK	The descriptor is for a file, not for a socket.
EOPNOTSUPP	The operation is not supported for the socket protocol.

## Related Information

- “accept() — Accept a New Connection on a Socket” on page 75
- “bind() — Bind a Name to a Socket” on page 128
- “connect() — Connect a Socket” on page 214
- “getpeername() — Get the Name of the Peer Connected to a Socket” on page 568
- “socket() — Create a Socket” on page 1371

## getsockopt() — Get the Options Associated with a Socket

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int getsockopt(int socket, int level,
               int option_name, void *option_value,
               size_t *option_len);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt(int socket, int level, int option_name,
               char *option_value, int *option_len);
```

### General Description

The `getsockopt()` call gets options associated with a socket. Not all options are supported by all address families. See each option for details. Options can exist at multiple protocol levels; they are always present at the highest socket level.

Parameter	Description
<i>socket</i>	The socket descriptor.
<i>level</i>	The level for which the option is set. Only <code>SOL_SOCKET</code> and <code>IPPROTO_IP</code> are supported.
<i>option_name</i>	The name of a specified socket option.
<i>option_value</i>	The pointer to option data.
<i>option_len</i>	The pointer to the length of the option data.

When manipulating socket options, you must specify the level at which the option resides and the name of the option. To manipulate options at the socket or IP level, the *level* parameter must be set to `SOL_SOCKET` or `IPPROTO_IP` as defined in **sys/socket.h**. To manipulate options at any other level, such as the TCP level, supply the appropriate protocol number for the protocol controlling the option. Currently, only the `SOL_SOCKET` and `IPPROTO_IP` levels are supported. The `getprotobyname()` call can be used to return the protocol number for a named protocol.

The *option\_value* and *option\_len* parameters are used to return data used by the particular get command. The *option\_value* parameter points to a buffer that is to receive the data requested by the get command. The *option\_len* parameter points to the size of the buffer pointed to by the *option\_value* parameter. It must be initially

set to the size of the buffer before calling `getsockopt()`. On return it is set to the actual size of the data returned.

All the socket level options except `SO_LINGER` expect *option\_value* to point to an integer and *option\_len* to be set to the size of an integer. When the integer is nonzero, the option is enabled. When it is zero, the option is disabled. The `SO_LINGER` option expects *option\_value* to point to a **linger** structure as defined in **sys/socket.h**. This structure is defined in the following example:

```
struct linger
{
    int    l_onoff;           /* option on/off */
    int    l_linger;         /* linger time */
};
```

The *l\_onoff* field is set to zero if the `SO_LINGER` option is being disabled. A nonzero value enables the option. The *l\_linger* field specifies the amount of time to linger on close.

The following options are recognized at the IP level:

Option	Description
<code>IP_MULTICAST_TTL</code>	This option is used to get the IP time-to-live of outgoing multicast datagrams. The TTL value is passed back as <code>u_char</code> .
<code>IP_MULTICAST_LOOP</code>	This option is used to determine whether loopback is enabled or disabled. The loopback indicator is passed back as <code>u_char</code> . 0 means loopback is disabled and 1 means it is enabled.
<code>IP_MULTICAST_IF</code>	This option is used to get the interface IP address used for sending outbound multicast datagrams. The IP address is passed back using struct <code>in_addr</code> .

The following options are recognized at the socket level:

Option	Description
<code>SO_DEBUG</code>	Reports whether debugging information is being recorded. This option stores an <code>int</code> value.
<code>SO_ACCEPTCONN</code>	The socket had a <code>listen()</code> call.
<code>SO_BROADCAST</code>	Toggles the ability to broadcast messages. If this option is enabled, it allows the application to send broadcast messages over <i>socket</i> , if the interface specified in the destination supports the broadcasting of packets. This option has no meaning for stream sockets. This option is valid only for the <code>AF_INET</code> domain.
<code>SO_ERROR</code>	Returns any pending error on the socket and clears the error status. You can use <code>SO_ERROR</code> to check for asynchronous errors on connected datagram sockets or for other asynchronous errors (errors that are not returned explicitly by one of the socket calls).

**SO\_KEEPAIVE**

Toggles the TCP keep-alive mechanism for a stream socket. When activated, the keep-alive mechanism periodically sends a packet on an otherwise idle connection. If the remote TCP does not respond to the packet or to retransmissions of the packet, the connection is ended with the error ETIMEDOUT. Processes writing to that socket are notified with a SIGPIPE signal. This option stores an int value. This option is valid only for the AF\_INET domain.

**SO\_LINGER**

Lingers on close if data is present. When this option is enabled and there is unsent data present when `close()` is called, the calling application is blocked during the `close()` call until the data is transmitted or the connection has timed out. If this option is disabled, the TCP/IP address space waits to try to send the data. Although the data transfer is usually successful, it cannot be guaranteed, because the TCP/IP address space waits only a finite amount of time trying to send the data. The `close()` call returns without blocking the caller. This option has meaning only for stream sockets.

**SO\_OOBINLINE**

Toggles reception of out-of-band data. When this option is enabled, out-of-band data is placed in the normal data input queue as it is received; it is then available to `recv()`, `recvfrom()`, and `recvmsg()` without the need to specify the MSG\_OOB flag in those calls. When this option is disabled, out-of-band data is placed in the priority data input queue as it is received; it is then available to `recv()`, `recvfrom()`, and `recvmsg()` only if the MSG\_OOB flag is specified in those calls. This option has meaning only for stream sockets.

**\_SO\_PROPAGATEUSERID**

Toggles propagating a user ID (UID) over a socket. When enabled, user (UID) information is extracted from the system when the `connect()` function is invoked and presented over the socket when the `accept()` function is invoked.

**SO\_RCVBUF**

Reports receive buffer size information. This option stores an int value.

**SO\_REUSEADDR**

Toggles local address reuse. When enabled, this option allows local addresses that are already in use to be bound. SO\_REUSEADDR alters the normal algorithm used in the `bind()` call. The system checks at connect time to ensure that the local address and port do not have the same foreign address and port. The error EADDRINUSE is returned if the association already exists. This option is valid only for the AF\_INET domain.

**SO\_SNDBUF**

Reports send buffer size information. This option stores an int value.

**SO\_TYPE**

This option returns the type of the socket. On return, the integer pointed to by *option\_value* is set to SOCK\_STREAM or SOCK\_DGRAM. This option is valid for both the AF\_UNIX and the AF\_INET domains.

**Special Behavior for C++**

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

## Returned Value

The value 0 indicates success; the value -1 indicates an error. The value of the error code indicates the specific error.

Error Code	Description
EBADF	The <i>socket</i> parameter is not a valid socket descriptor.
EFAULT	Using <i>option_value</i> and <i>option_len</i> parameters would result in an attempt to access storage outside the caller's address space.
EINVAL	The specified option is invalid at the specified socket level.
ENOBUFS	Buffer space is not available to send the message.
ENOPROTOOPT	The <i>option_name</i> parameter is unrecognized, or the <i>level</i> parameter is not <code>SOL_SOCKET</code> .
ENOSYS	The function is not implemented. You attempted to use a function that is not yet available.
ENOTSOCK	The descriptor is for a file, not for a socket.
EOPNOTSUPP	The operation is not supported by the socket protocol.

## Example

The following are examples of the `getsockopt()` call. See “`setsockopt()` — Set Options Associated with a Socket” on page 1270 for examples of how the `setsockopt()` call options are set.

```
int rc;
int s;
int option_value;
int option_len;
struct linger l;
int getsockopt(int s, int level, int option_name, char *option_value, int *option_len);

:
/* Is out-of-band data in the normal input queue? */
option_len = sizeof(int);
rc = getsockopt(
    s, SOL_SOCKET, SO_OOBINLINE, (char *) &option_value, &option_len);
if (rc == 0)
{
    if (option_len == sizeof(int))
    {
        if (option_value)
            /* yes it is in the normal queue */
        else
            /* no it is not */
    }
}

:
/* Do I linger on close? */
option_len = sizeof(l);
rc = getsockopt(
    s, SOL_SOCKET, SO_LINGER, (char *) &l, &option_len);
if (rc == 0)
{
```

```
    if (option_len == sizeof(l))
    {
        if (l.l_onoff)
            /* yes I linger */
        else
            /* no I do not */
    }
}
```

**Related Information**

- “bind() — Bind a Name to a Socket” on page 128
- “close() — Close a File” on page 191
- “getprotobyname() — Get a Protocol Entry by Name” on page 580
- “setsockopt() — Set Options Associated with a Socket” on page 1270
- “socket() — Create a Socket” on page 1371



## getstablesizesize() — Get the Socket Table Size

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#define _OPEN_SYS_SOCKET_EXT
#include <sys/socket.h>
```

```
int getstablesizesize(void);
```

### General Description

The `getstablesizesize()` function obtains the number of sockets that are allowed for use in bulk mode operations for a process.

### Returned Value

The `getstablesizesize()` function returns the current limit for this process. If it has not been changed by the `maxdesc()` function then the default is returned. The default is the hard limit returned by `getrlimit()` for `RLIMIT_NOFILE`. This is the value set by a `BPXPRMnn` parmlib member on its `MAXFILEPROC` statement.

There are no `errno` values defined for `getstablesizesize()`.

### Related Information

- “`sys/socket.h`” on page 48
- “`maxdesc()` — Get Socket Numbers to Extend Beyond the Default Range” on page 789

## getsubopt() — Parse Suboption Arguments

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>
```

```
int getsubopt(char **optionp, char * const * tokens, char **valuep);
```

### General Description

The `getsubopt()` function parses suboption arguments in a flag argument that was initially parsed by `getopt()`. These suboption arguments must be separated by commas and may consist of either a single token, or a token-value pair separated by an equal sign. Because commas delimit suboption arguments in the option string, they are not allowed to be part of the suboption arguments or the value of a suboption argument. Similarly, because the equal sign separates a token from its value, a token must not contain an equal sign.

The `getsubopt()` function takes the address of a pointer to the option argument string, a vector of possible tokens, and the address of a value string pointer. If the option argument string at *optionp* contains only one suboption argument, `getsubopt()` updates *optionp* to point to the null at the end of the string. Otherwise, it isolates the suboption argument by replacing the comma separator with a null, and updates *optionp* to point to the start of the next suboption argument. If the suboption argument has an associated value, `getsubopt()` updates *valuep* to point to the value's first character. Otherwise it sets *valuep* to a null pointer.

The token vector is organized as a series of pointers to strings. The end of the token vector is identified by a null pointer.

When `getsubopt()` returns, if *valuep* is not a null pointer, then the suboption argument processed included a value. The calling program may use this information to determine if the presence or lack of a value for the suboption is an error.

Additionally, when `getsubopt()` fails to match the suboption argument with the tokens in the *tokens* array, the calling program should decide if this is an error, or if the unrecognized option should be passed on to another program.

Because the `getsubopt()` function returns thread specific data the `getsubopt()` function can be used safely from a multi-threaded application.

### Returned Value

If successful, `getsubopt()` returns the index of the matched token string, or -1 if no token strings were matched.

`getsubopt()` does not return any `errno` values.

**Related Information**

- “stdlib.h” on page 45
- “getopt() — Command Option Parsing” on page 564

## getsyntax() — Return LC\_SYNTAX Characters

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <variant.h>
```

```
struct variant *getsyntax(void);
```

### General Description

Determines the encoding of the special characters defined in the LC\_SYNTAX category of the current locale, and stores the encoding values in the structure of type *variant*. For details of the variant structure, see “variant.h” on page 54.

### Returned Value

Returns the pointer to the structure containing the values of the special characters.

If the information about the special characters is not available in the current locale, getsyntax() returns a null pointer.

The structure returned is not modified by the program that this function is used in. The structure may be invalidated by calls to the setlocale() function with LC\_ALL, LC\_CTYPE, LC\_COLLATE, or LC\_SYNTAX.

### Example

#### CBC3BG19

```
/* CBC3BG19 */
#include <stdio.h>
#include <stdlib.h>
#include <variant.h>
#include <wchar.h>

int main(void)
{
    struct variant *var;

    var = getsyntax();
    printf("codeset           : %s\n", var->codeset      );
    printf("backslash          : %3d\n", var->backslash     );
    printf("right_bracket         : %3d\n", var->right_bracket    );
    printf("left_bracket          : %3d\n", var->left_bracket     );
    printf("right_brace           : %3d\n", var->right_brace      );
    printf("left_brace            : %3d\n", var->left_brace       );
    printf("circumflex            : %3d\n", var->circumflex       );
    printf("tilde                 : %3d\n", var->tilde            );
    printf("exclamation_mark      : %3d\n", var->exclamation_mark );
    printf("number_sign           : %3d\n", var->number_sign      );
    printf("vertical_line         : %3d\n", var->vertical_line     );
    printf("dollar_sign           : %3d\n", var->dollar_sign       );
    printf("commercial_at         : %3d\n", var->commercial_at    );
    printf("grave_accent          : %3d\n", var->grave_accent     );
}
```

**Related Information**

- “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*
- “locale.h” on page 33
- “variant.h” on page 54
- “setlocale() — Set Locale” on page 1241

## gettimeofday() — Get Date and Time

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1

#undef _ALL_SOURCE
#include <sys/time.h>
int gettimeofday(struct timeval *tp, void *tzp);

#define _ALL_SOURCE
#include <sys/time.h>
int gettimeofday(struct timeval *tp, struct timezone *tzp);
```

### General Description

The `gettimeofday()` function obtains the current time, expressed as seconds and microseconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970, and stores it in the `timeval` structure pointed to by `tp`.

### Special Behavior for `_ALL_SOURCE`

The `gettimeofday()` function has two prototypes. Which one is used depends on whether or not you define the `_ALL_SOURCE` feature test macro when you compile your program. If `_ALL_SOURCE` is NOT defined when the C/370 preprocessor processes the `<sys/time.h>` header, it includes a prototype for `gettimeofday()` which defines the second argument, `tzp`, as a void pointer and includes a C/370 pragma map statement for a C/370 version of `gettimeofday()` which ignores `tzp`.

If `_ALL_SOURCE` is defined, the C/370 preprocessor includes a prototype for `gettimeofday()` which defines `tzp` as a pointer to a `timezone` structure and includes a pragma map statement for a C/370 version of `gettimeofday()` which stores time zone information in the `timezone` structure to which the second argument points. The `timezone` structure contains the following members:

```
int tz_minuteswest; /* Time west of Greenwich in minutes */
int tz_dsttime;     /* Type of DST correction to apply */
```

When `_ALL_SOURCE` is defined, the `gettimeofday()` function:

1. invokes `tzset()` to set the values of the `timezone` and `daylight` external variables.
2. converts the value of the `timezone` external variable to minutes and stores the converted value, rounded up to the nearest minute, in `tzp->tz_minuteswest`.
3. stores the value of the `daylight` external variable in `tzp->tz_dsttime`.

### Returned Value

`gettimeofday()` returns 0 unless overflow occurs. It returns a nonzero value if overflow occurs. Overflow occurs when the current time in seconds since 00:00:00 UTC, January 1, 1970 exceeds the capacity of the `tv_sec` member of the `timeval` structure pointed to by `tp`. The `tv_sec` member is type `time_t`.

**Related Information**

- “limits.h” on page 32
- “sys/time.h” on page 49
- “ctime() — Convert Time to a Character String” on page 250
- “ftime() — Get the Date and Time” on page 487

## getuid() — Get the Real User ID

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
uid_t getuid(void);
```

### General Description

Finds the real user ID (UID) of the calling process.

### Returned Value

Returns the found value. It is always successful. There are no documented errors for this function.

### Example CBC3BG20

```
/* CBC3BG20
   This example provides information for your user ID.
*/
#define _POSIX_SOURCE
#include <pwd.h>
#include <sys/types.h>
#include <unistd.h>

main() {
    struct passwd *p;
    uid_t uid;

    if ((p = getpwuid(uid = getuid())) == NULL)
        perror("getpwuid() error");
    else {
        puts("getpwuid() returned the following info for your userid:");
        printf(" pw_name  : %s\n",      p->pw_name);
        printf(" pw_uid   : %d\n", (int) p->pw_uid);
        printf(" pw_gid   : %d\n", (int) p->pw_gid);
        printf(" pw_dir   : %s\n",      p->pw_dir);
        printf(" pw_shell : %s\n",      p->pw_shell);
    }
}
```

### Output

```
pw_name  : MVSUSR1
pw_uid   : 25
pw_gid   : 500
pw_dir   : /u/mvsusr1
pw_shell : /bin/sh
```



**Related Information**

- “sys/types.h” on page 49
- “geteuid() — Get the Effective User ID” on page 519
- “seteuid() — Set the Effective User ID” on page 1218
- “setreuid() — Set Real and Effective User IDs” on page 1262
- “setuid() — Set the Effective User ID” on page 1278

## getutxent() — Read Next Entry in utmpx Database

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <utmpx.h>
```

```
struct utmpx *getutxent(void);
```

### General Description

The `getutxent()` function reads in the next entry from the `utmpx` database. If the database is not already open, it opens it. If it reaches the end of the database, it fails.

The functions `getutxent()`, `getutxid()`, and `getutxline()` obtain a shared lock on the database. The function `pututxline()` releases the shared lock, obtains an exclusive lock while writing to the database, releases the exclusive lock, and obtains the shared lock on the database. This lock is obtained using the `fcntl()` command **F\_SETLKW** on the first record in the database.

Because the `getutxent()` function returns thread specific data the `getutxent()` function can be used safely from a multi-threaded application. If multiple threads in the same process open the database, then each thread opens the database with a different file descriptor. The thread's database file descriptor is closed when the calling thread terminates or the `endutxent()` function is called by the calling thread.

The name of the database file defaults to `/etc/utmpx`. To process a different database file name use the `__utmpxname()` function.

For all entries that match a request, the `ut_type` member indicates the type of the entry. Other members of the entry will contain meaningful data based on the value of the `ut_type` member as follows:

<b>EMPTY</b>	No other members have meaningful data.
<b>BOOT_TIME</b>	<code>ut_tv</code> is meaningful.
<b>__RUN_LVL</b>	<code>ut_tv</code> and <code>ut_line</code> are meaningful
<b>OLD_TIME</b>	<code>ut_tv</code> is meaningful.
<b>NEW_TIME</b>	<code>ut_tv</code> is meaningful.
<b>USER_PROCESS</b>	<code>ut_id</code> , <code>ut_user</code> (login name of the user), <code>ut_line</code> , <code>ut_pid</code> , and <code>ut_tv</code> are meaningful.
<b>INIT_PROCESS</b>	<code>ut_id</code> , <code>ut_pid</code> , and <code>ut_tv</code> are meaningful.
<b>LOGIN_PROCESS</b>	<code>ut_id</code> , <code>ut_user</code> (implementation-specific name of the login process), <code>ut_pid</code> , and <code>ut_tv</code> are meaningful.
<b>DEAD_PROCESS</b>	<code>ut_id</code> , <code>ut_pid</code> , and <code>ut_tv</code> are meaningful.

**Returned Value**

If successful, `getutxent()` returns a pointer to a `utmpx` structure containing a copy of the requested entry in the user accounting database. Otherwise a null pointer is returned.

No errors are defined for this function.

**Related Information**

- “`endutxent()` — Close the `utmpx` Database” on page 313
- “`getutxid()` — Search by ID `utmpx` Database” on page 622
- “`getutxline()` — Search by Line `utmpx` Database” on page 624
- “`pututxline()` — Write Entry to `utmpx` Database” on page 1061
- “`setutxent()` — Reset to Start of `utmpx` Database” on page 1282
- “`__utmpxname()` — Change the `utmpx` Database Name” on page 1669

## getutxid() — Search by ID utmpx Database

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <utmpx.h>
```

```
struct utmpx *getutxid(const struct utmpx *id);
```

### General Description

The `getutxid()` function searches forward from the current point in the utmpx database. If the database is not already open, it opens it. If the `ut_type` value of the utmpx structure pointed to by `id` is **BOOT\_TIME**, **\_\_RUN\_LVL**, **OLD\_TIME**, or **NEW\_TIME**, then it stops when it finds an entry with a matching `ut_type` value. If the `ut_type` value is **INIT\_PROCESS**, **LOGIN\_PROCESS**, **USER\_PROCESS**, or **DEAD\_PROCESS**, then it stops when it finds an entry whose type is one of these four and whose `ut_id` member matches the `ut_id` member of the utmpx structure pointed to by `id`. If the `UT_type` value is **EMPTY**, `getutxid()` fails (returns **NULL**) without repositioning the utmpx database to the end. If the end of the of the database is reached without a match, `getutxid()` fails.

The functions `getutxent()`, `getutxid()`, and `getutxline()` obtain a shared lock on the database. The function `pututxline()` releases the shared lock, obtains an exclusive lock while writing to the database, releases the exclusive lock, and obtains the shared lock on the database. This lock is obtained using the `fcntl()` command **F\_SETLKW** on the first record in the database.

Because the `getutxid()` function returns thread specific data the `getutxid()` function can be used safely from a multi-threaded application. If multiple threads in the same process open the database, then each thread opens the database with a different file descriptor. The thread's database file descriptor is closed when the calling thread terminates or the `endutxent()` function is called by the calling thread.

The name of the database file defaults to `/etc/utmpx`. To process a different database file name use the `__utmpxname()` function.

For all entries that match a request, the `ut_type` member indicates the type of the entry. Other members of the entry will contain meaningful data based on the value of the `ut_type` member as follows:

<b>EMPTY</b>	No other members have meaningful data.
<b>BOOT_TIME</b>	<code>ut_tv</code> is meaningful.
<b>__RUN_LVL</b>	<code>ut_tv</code> and <code>ut_line</code> are meaningful
<b>OLD_TIME</b>	<code>ut_tv</code> is meaningful.
<b>NEW_TIME</b>	<code>ut_tv</code> is meaningful.

<b>USER_PROCESS</b>	ut_id, ut_user (login name of the user), ut_line, ut_pid, and ut_tv are meaningful.
<b>INIT_PROCESS</b>	ut_id, ut_pid, and ut_tv are meaningful.
<b>LOGIN_PROCESS</b>	ut_id, ut_user (implementation-specific name of the login process), ut_pid, and ut_tv are meaningful.
<b>DEAD_PROCESS</b>	ut_id, ut_pid, and ut_tv are meaningful.

### Returned Value

If successful, getutxid() returns a pointer to a utmpx structure containing a copy of the requested entry in the user accounting database. Otherwise a null pointer is returned.

No errors are defined for this function.

### Related Information

- “endutxent() — Close the utmpx Database” on page 313
- “getutxent() — Read Next Entry in utmpx Database” on page 620
- “getutxline() — Search by Line utmpx Database” on page 624
- “pututxline() — Write Entry to utmpx Database” on page 1061
- “setutxent() — Reset to Start of utmpx Database” on page 1282
- “\_\_utmpxname() — Change the utmpx Database Name” on page 1669

## getutxline() — Search by Line utmpx Database

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <utmpx.h>
```

```
struct utmpx *getutxline(const struct utmpx *line);
```

### General Description

The `getutxline()` function searches forward from the current point in the utmpx database until it finds an entry of the type **LOGIN\_PROCESS** or **USER\_PROCESS** which also has a `ut_line` value matching that in the utmpx structure pointed to by argument `line`. If the database is not already open, it opens it. If it reaches the end of the database, it fails.

The functions `getutxent()`, `getutxid()`, and `getutxline()` obtain a shared lock on the database. The function `pututxline()` releases the shared lock, obtains an exclusive lock while writing to the database, releases the exclusive lock, and obtains the shared lock on the database. This lock is obtained using the `fcntl()` command **F\_SETLKW** on the first record in the database.

Because the `getutxline()` function returns thread specific data the `getutxline()` function can be used safely from a multi-threaded application. If multiple threads in the same process open the database, then each thread opens the database with a different file descriptor. The thread's database file descriptor is closed when the calling thread terminates or the `endutxent()` function is called by the calling thread.

The name of the database file defaults to `/etc/utmpx`. To process a different database file name use the `__utmpxname()` function.

The functions `getutxent()`, `getutxid()`, and `getutxline()` cache the last entry read from the database. For this reason, to use `getutxline()` function to search for multiple occurrences, it is necessary to zero out the utmpx structure pointed to by the return value from these functions.

For all entries that match a request, the `ut_type` member indicates the type of the entry. Other members of the entry will contain meaningful data based on the value of the `ut_type` member as follows:

<b>EMPTY</b>	No other members have meaningful data.
<b>BOOT_TIME</b>	<code>ut_tv</code> is meaningful.
<b>__RUN_LVL</b>	<code>ut_tv</code> and <code>ut_line</code> are meaningful
<b>OLD_TIME</b>	<code>ut_tv</code> is meaningful.
<b>NEW_TIME</b>	<code>ut_tv</code> is meaningful.

<b>USER_PROCESS</b>	ut_id, ut_user (login name of the user), ut_line, ut_pid, and ut_tv are meaningful.
<b>INIT_PROCESS</b>	ut_id, ut_pid, and ut_tv are meaningful.
<b>LOGIN_PROCESS</b>	ut_id, ut_user (implementation-specific name of the login process), ut_pid, and ut_tv are meaningful.
<b>DEAD_PROCESS</b>	ut_id, ut_pid, and ut_tv are meaningful.

### Returned Value

If successful, `getutxline()` returns a pointer to a `utmpx` structure containing a copy of the requested entry in the user accounting database. Otherwise a null pointer is returned.

No errors are defined for this function.

### Related Information

- “`endutxent()` — Close the `utmpx` Database” on page 313
- “`getutxent()` — Read Next Entry in `utmpx` Database” on page 620
- “`getutxid()` — Search by ID `utmpx` Database” on page 622
- “`pututxline()` — Write Entry to `utmpx` Database” on page 1061
- “`setutxent()` — Reset to Start of `utmpx` Database” on page 1282
- “`__utmpxname()` — Change the `utmpx` Database Name” on page 1669

## getw() — Get a Machine Word from a Stream

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <stdio.h>
```

```
int getw(FILE *stream);
```

### General Description

The `getw()` function reads the next word from the *stream*. The size of the word is the size of an int, and varies from machine to machine. The `getw()` function presumes no special alignment in the file.

The `getw()` function may mark the *st\_atime* field of the file associated with *stream* for update. The *st\_atime* field will be marked for update by the first successful execution of `fgetc()`, `fgets()`, `fread()`, `getc()`, `getchar()`, `gets()`, `fscanf()` or `scanf()` using *stream* that returns data not supplied by a prior call to `ungetc()`.

### Returned Value

If successful, `getw()` returns the next word from the input stream pointed to by *stream*. If the stream is at end-of-file, the end-of-file indicator for the stream is set and `getw()` returns EOF. If a read error occurs, the error indicator for the stream is set, `getw()` returns EOF and sets `errno` to indicate the error.

Refer to “`fgetc()` — Read a Character” on page 388 for `errno` values.

Because the representation of EOF is a valid integer, applications wishing to check for errors should use `ferror()` and `feof()`.

### Related Information

- “`stdio.h`” on page 43
- “`fopen()` — Open a File” on page 417
- “`fwrite()` — Write Items” on page 495
- “`putw()` — Put a Machine Word on a Stream” on page 1063



## getwc() — Get a Wide Character

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h>
```

```
wint_t getwc(FILE *stream);
```

### General Description

Obtains the next multibyte character from `stdin`, converts it to a wide character, and advances the associated file position indicator for `stdin`.

The `getwc()` function is equivalent to the `fgetwc()` function. Therefore, the argument should never be an expression with side effects.

The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you change the category, undefined results can occur.

Using non-wide-character functions with `getwc()` results in undefined behavior. This happens because `getwc()` processes a whole multibyte character and does not expect to be “within” such a character. In addition, `getwc()` expects state information to be set already. Because functions like `fgetc()` and `fputc()` do not obey such rules, their results fail to meet the assumptions made by `getwc()`.

`getwc()` has the same restriction as any read operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

### Returned Value

Returns the next wide character from the input stream pointed to by *stream* or else the function returns WEOF. If there is an error, `getwc()` sets the error indicator. If the EOF is encountered, it sets the EOF indicator. If an encoding error is encountered, it sets EILSEQ in `errno`.

Use `ferror()` or `feof()` to determine whether an error or an EOF condition occurred. Note that EOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

### Example

#### CBC3BG21

```
/* CBC3BG21
   This example attempts to get a wide character from a file myfile.dat
*/
#include <errno.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
    FILE    *stream;
    wint_t   wc;

    if ((stream = fopen("myfile.dat", "r")) == NULL) {
        printf("Unable to open file.");
        exit(1);
    }

    errno = 0;
    while ((wc = getwc(stream)) != WEOF)
        printf("wc=0x%lx\n", wc);

    if (errno == EILSEQ) {
        printf("An invalid wide character was encountered.\n");
        exit(1);
    }

    fclose(stream);
}
```

**Related Information**

- “stdio.h” on page 43
- “wchar.h” on page 54
- “fgetwc() — Get Next Wide Character” on page 394

## getwchar() — Get a Wide Character

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h>
```

```
wint_t getwchar(void);
```

### General Description

The `getwchar()` function is equivalent to `getwc()` with the argument `stdin`.

The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you change the category, undefined results can occur.

### Returned Value

Returns the next wide character from the input stream pointed to by `stdin` or else the function returns `WEOF`. If the stream is at EOF, the EOF indicator for the stream is set and `fgetwc()` returns `WEOF`. If a read error occurs, the error indicator for the stream is set and `fgetwc()` returns `WEOF`. If an encoding error occurs, the value of the macro `EILSEQ` is stored in `errno` and `WEOF` is returned.

Use `ferror()` or `feof()` to determine whether an error or an EOF condition occurred. Note that EOF is only reached when an attempt is made to read past the last byte of data. Reading up to and including the last byte of data does *not* turn on the EOF indicator.

### Example

#### CBC3BG22

```
/* CBC3BG22
   This example attempts to read all of the wide characters in stdin and
   print them out.
*/
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
    wint_t    wc;

    errno = 0;
    while ((wc = getwchar()) != WEOF)
        printf("wc=0x%X\n", wc);

    if (errno == EILSEQ) {
        printf("An invalid wide character was encountered.\n");
        exit(1);
    }
}
```

**Related Information**

- “wchar.h” on page 54
- “fgetwc() — Get Next Wide Character” on page 394
- “getwc() — Get a Wide Character” on page 627

## getwd() — Get the Current Working Directory

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
```

```
char *getwd(char *path_name);
```

### General Description

The `getwd()` function determines an absolute pathname of the current working directory of the calling process, and copies that pathname into the array pointed to by `path_name` argument.

If the length of the pathname of the current working directory is greater than  $(\{\text{PATH\_MAX}\}+1)$  including the null byte, `getwd()` fails and returns a null pointer.

For portability to implementations conforming to earlier versions of the standards, `getcwd()` is preferred over this function.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

Upon successful completion, a pointer to the string containing the absolute pathname of the current working directory is returned. Otherwise, `getwd()` returns a null pointer and the contents of the array pointed to by `path_name` are undefined.

There are no `errno` values defined for `getwd()`.

### Related Information

- “`unistd.h`” on page 53
- “`getcwd()` — Get Path Name of the Working Directory” on page 508

## getwmccoll() — Get Next Collating Element from Wide String

### Standards

Standards / Extensions	C or C++	Dependencies
C/370	both	

### Format

```
#include <collate.h>
```

```
coll_t getwmccoll(wchar_t **src);
```

### General Description

If the object pointed to by *src* is not a null pointer, the `getwmccoll()` library function determines the longest sequence of wide characters in the array pointed to by *str* that constitute a valid multi-wide-character collating element. It then produces the value of type `coll_t` corresponding to that collating element. The object pointed to by *src* is assigned the address just past the last wide character of the multi-wide-character collating element processed.

### Returned Value

If the object pointed to by *src* is a null pointer or if it points to a null wide character, the function returns the value 0. If the object pointed to by *src* points to an invalid wide character, it returns the value -1 and sets `errno` to `EILSEQ`. Otherwise, it returns the value of type `coll_t` that represents the collating element found.

### Related Information

- “`collate.h`” on page 23
- “`wchar.h`” on page 54
- “`cclass()` — Return Characters in a Character Class” on page 149
- “`collequiv()` — Return a List of Equivalent Collating Elements” on page 200
- “`collorder()` — Return List of Collating Elements” on page 202
- “`collrange()` — Calculate the Range List of Collating Elements” on page 204
- “`colltostr()` — Return a String for a Collating Element” on page 206
- “`getmccoll()` — Get Next Collating Element from String” on page 554
- “`ismccollel()` — Identify a Multi-Character Collating Element” on page 710
- “`maxcoll()` — Return Maximum Collating Element” on page 788
- “`strtocoll()` — Return Collating Element for String” on page 1454

## givesocket() — Make the Specified Socket Available

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#define _OPEN_SYS_SOCKET_EXT
#include <sys/socket.h>
```

```
int givesocket(int d, struct clientid *clientid);
```

### General Description

The givesocket() call makes the specified socket available to a takesocket() call issued by another program. Any socket can be given. Typically, givesocket() is used by a master program that obtains sockets by means of accept() and gives them to application programs that handle one socket at a time.

#### Parameter      Description

*d*                      The descriptor of a socket to be given to another application.

*clientid*              A pointer to a client ID structure specifying the program to which the socket is to be given.

To pass a socket, the giving program first calls givesocket() with the client ID structure filled in as follows:

The clientid structure:

```
struct clientid {
    int domain;
    union {
        char name[8];
        struct {
            int NameUpper;
            pid_t pid;
        } c_pid;
    } c_name;
    char subtaskname[8];

    struct {
        char type;
        union {
            char specific[19];
            struct {
                char unused[3];
                int SockToken;
            } c_close;
        } c_func;
    } c_reserved;
};
```

#### Element      Description

*domain*              The domain of the input socket descriptor.

*c\_name.name*

If the *clientid* was set by a `getclientid()` call, *c\_name.name* can be

- set to the application program's address space name, left-justified and padded with blanks. The application program can run in the same address space as the master program, in which case this field is set to the master program's address space.
- set to blanks, so any OS/390 address space can take the socket.

*subtaskname*

If the *clientid* was set by a `getclientid()` call, *subtaskname* can be

- set to the task identifier of the taker. This, combined with a *c\_name.name* value, allows only a process with this *c\_name.name* and *subtaskname* to take the socket.
- set to blanks. If *c\_name.name* has a value and *subtaskname* is blank, any task with that *c\_name.name* can take the socket.

*c\_pid.pid*

If the *clientid* was set by a `__getclientid()` call, *c\_pid.pid* should be set to the process id (PID) of the taker, so only a process with that PID can take the socket. The *subtaskname* field is ignored when the *c\_pid* has a value.

*c\_reserved.type*

When set to `SO_CLOSE`, this indicates the socket should be automatically closed by `givesocket()`, and a unique socket identifying token is to be returned in *c\_close.SockToken*. The *c\_close.SockToken* should be passed to the taking program to be used as input to `takesocket()` instead of the socket descriptor. The now closed socket descriptor could be re-used by the time the `takesocket()` is called, so the *c\_close.SockToken* should be used for `takesocket()`.

*c\_close.SockToken*

The unique socket identifying token returned by `givesocket` to be used as input to `takesocket()`, instead of the socket descriptor when *c\_reserved.type* has been set to `SO_CLOSE`.

*c\_reserved*

Specifies binary zeros if an automatic close of a socket is not to be done by `givesocket()`.

**Using name and subtaskname for givesocket/takesocket:**

1. The giving program calls `getclientid()` to obtain its client ID. The giving program calls `givesocket()` to make the socket available for a `takesocket()` call. The giving program passes its client ID along with the descriptor of the socket to be given to the taking program by the taking program's startup parameter list.
2. The taking program calls `takesocket()`, specifying the giving program's client ID and socket descriptor.
3. Waiting for the taking program to take the socket, the giving program uses `select()` to test the given socket for an exception condition. When `select()` reports that an exception condition is pending, the giving program calls `close()` to free the given socket.
4. If the giving program closes the socket before a pending exception condition is indicated, the connection is immediately reset, and the taking program's call to `takesocket()` is unsuccessful. Calls other than the `close()` call issued on a given socket return -1, with `errno` set to `EBADF`.



Note: For backward compatibility, a client ID can point to the struct client ID structure obtained when the target program calls `getclientid()`. In this case, only the target program, and no other programs in the target program's address space, can take the socket.

### Using process id (PID) for givesocket/takesocket:

1. The giving program calls `__getclientid()` to obtain its client ID. The giving program sets the `c_pid.pid` in the clientid structure to the PID of the taking program that will take the socket (i.e. issue the `takesocket()` call). This ensures only a process that has obtained the giver's PID can take the specified socket. If the giving program wants the socket to be automatically closed by `givesocket()`, `c_reserved.type` should be set to `SO_CLOSE`. The giving program calls `givesocket()` to make the socket available for a `takesocket()` call. The giving program passes its client ID, the descriptor of the socket to be given, and the giving program's PID to the taking program by the taking program's startup parameter list.
2. The taking program sets the `c_pid.pid` in the clientid structure to the PID of the giving program to identify the process from which the socket is to be taken. If the `c_reserved.type` field was set to `SO_CLOSE` on `givesocket()`, the `c_close.SockToken` should be used as input to the `takesocket()` instead of the normal socket descriptor. The taking program calls `takesocket()`, specifying the giving program's client ID and either the socket descriptor or `c_close.SockToken`.
3. If the `c_reserved.type` field in the clientid structure was set to `SO_CLOSE` on the `givesocket()` call, the socket is closed and the giving program does not have to wait for the taking program to issue the `takesocket()`. Otherwise, steps 3 and 4 of "Using name and subtaskname for givesocket/takesocket" should be followed.

### Returned Value

The value 0 indicates success. The value -1 indicates an error. The value of `errno` indicates the specific error.

### Errno Code Description

EBADF	The <code>d</code> parameter is not a valid socket descriptor. The socket has already been given.
EFAULT	Using the clientid parameter as specified would result in an attempt to access storage outside the caller's address space.
EINVAL	The clientid parameter does not specify a valid client identifier or the clientid domain does not match the domain of the input socket descriptor.

### Related Information

- "sys/socket.h" on page 48
- "accept() — Accept a New Connection on a Socket" on page 75
- "close() — Close a File" on page 191
- "getclientid() — Get the Identifier for the Calling Application" on page 501
- "listen() — Prepare the Server for Incoming Client Requests" on page 752
- "select() — Monitor Activity on Files/Sockets and Message Queues" on page 1159
- "takesocket() — Acquire a Socket from Another Program" on page 1496

## glob() — Generate Pathnames Matching a Pattern

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE
#include <glob.h>

int glob(const char *pattern, int flags,
         int (*errfunc)(const char *epath,
                        int eerrno),
         glob_t *pglob);
```

### General Description

The `glob()` function is a pathname generator that implements the rules defined in *X/Open CAE Specification, Commands and Utilities, Issue 4, Version 2* Section 2.13, **Pattern Matching Notation**, with optional support for rule 3 in Section 2.13.3, **Patterns Used for Filename Expansion**.

The structure `glob_t` is defined in the header `<glob.h>` and includes at least the following members:

`gl_pathc`     Count of paths matched by *pattern*.  
`gl_pathv`     Pointer to a list of matched filenames.  
`gl_offs`       Slots to reserve at the beginning of *gl\_pathv*.

The argument *pattern* is a pointer to a pathname pattern to be expanded. The `glob()` function matches all accessible pathnames against this pattern and develops a list of all pathnames that match. In order to have access to a pathname, `glob()` requires search permission on every component of a path except the last, and read permission on each directory of any filename component of *pattern* that contains any of the following special characters:

\*           ?           [

The `glob()` function stores the number of matched pathnames into `pglob->gl_pathc` and a pointer to a list of pointers to pathnames into `pglob->gl_pathv`. The pathnames are in sort order as defined by the current setting of the `LC_COLLATE` category, see *X/Open CAE Specification, System Interface Definitions, Issue 4, Version 2* Section 5.3.2, `LC_COLLATE`. The first pointer after the last pathname is a null pointer. If the pattern does not match any pathnames, the returned number of matched paths is set to 0, and the contents of `pglob->gl_pathv` are implementation-dependent.

It is the caller's responsibility to create the structure pointed to by *pglob*. The `glob()` function allocates other space as needed., including the memory pointed to by *gl\_pathv*.

The *flags* argument is used to control the behavior of `glob()`. The value of *flags* is a bitwise inclusive OR of zero or more of the following constants, which are defined in the header `<glob.h>`:

#### GLOB\_APPEND

Append pathnames generated to the ones from a previous call to `glob()`.

#### GLOB\_DOOFFS

Make use of `pglob->gl_offs`. If this flag is set, `pglob->gl_offs` is used to specify how many null pointers to add to the beginning of `pglob->gl_pathv`. In other words, `pglob->gl_pathv` will point to `pglob->gl_offs` null pointers, followed by `pglob->gl_pathc` pathname pointers, followed by a null pointer.

**GLOB\_ERR** Causes `glob()` to return when it encounters a directory that it cannot open or read. Ordinarily, `glob()` continues to find matches.

**GLOB\_MARK** Each pathname that is a directory that matches *pattern* has a slash appended.

#### GLOB\_NOCHECK

Support rule 3 in the XCU specification, Section 2.13.3, Patterns Used for Filename Expansion. If *pattern* does not match any pathname, then `glob()` returns a list consisting of only *pattern*, and the number of matched pathnames is 1.

#### GLOB\_NOESCAPE

Disable backslash escaping.

#### GLOB\_NOSORT

Ordinarily, `glob()` sorts the matching pathnames according to the current setting of the `LC_COLLATE` category, see the XBD specification, Section 5.3.2, `LC_COLLATE`. When this flag is used the order of pathnames returned is unspecified.

The `GLOB_APPEND` flag can be used to append a new set of pathnames to those found in a previous call to `glob()`. The following rules apply when two or more calls to `glob()` are made with the same value of `pglob` and without intervening calls to `globfree()`:

1. The first such call must not set `GLOB_APPEND`. All subsequent calls must set it.
2. All calls must set `GLOB_DOOFFS`, or all must not set it.
3. After the second call, `pglob->gl_pathv` points to a list containing the following:
  - a. Zero or more null pointers, as specified by `GLOB_DOOFFS` and `pglob->gl_offs`.
  - b. Pointers to the pathnames that were in the `pglob->gl_pathv` list before the call, in the same order as before.
  - c. Pointers to the new pathnames generated by the second call, in the specified order.
4. The count returned in `pglob->gl_pathc` will be the total number of pathnames from the two calls.

5. The application can change any of the fields after a call to `glob()`. If it does, it must reset them to the original value before a subsequent call, using the same `pglob` value, to `globfree()` or `glob()` with the `GLOB_APPEND` flag.

If, during the search, a directory is encountered that cannot be opened or read and `errfunc` is not a null pointer, `glob()` calls `(*errfunc())` with two arguments:

1. The `epath` argument is a pointer to the path that failed.
2. The `eerrno` argument is the value of `errno` from the failure, as set by `opendir()`, `readdir()` or `stat()`. (Other values may be used to report other errors not explicitly documented for those functions.)

## Returned Value

If successful, `glob()` returns 0. The argument `pglob->gl_pathc` returns the number of matched pathnames and the argument `pglob->gl_pathv` contains a pointer to a null-terminated list of matched and sorted pathnames. However, if `pglob->gl_pathc` is 0, the content of `pglob->gl_pathv` is undefined.

If `glob()` terminates due to an error, it returns one of the following nonzero constants defined in `<glob.h>` as error return values for `glob()`:

### GLOB\_ABORTED

The scan was stopped because `GLOB_ERR` was set or `(*errfunc())` returned nonzero.

### GLOB\_NOMATCH

The pattern does not match any existing pathname, and `GLOB_NOCHECK` was set in `flags`.

### GLOB\_NOSPACE

An attempt to allocate memory failed.

If `(*errfunc())` is called and returns nonzero, or if the `GLOB_ERR` flag is set in `flags`, `glob()` stops the scan and returns `GLOB_ABORTED` after setting `gl_pathc` and `gl_pathv` in `pglob` to reflect the paths already scanned. If `GLOB_ERR` is not set and either `errfunc` is a null pointer or `(*errfunc())` returns 0, the error is ignored.

## Related Information

- “`glob.h`” on page 29
- “exec Functions” on page 322
- “`fnmatch()` — Match Filename or Pathname” on page 415
- “`opendir()` — Open a Directory” on page 877
- “`readdir()` — Read an Entry from a Directory” on page 1086
- “`stat()` — Get File Information” on page 1404
- “`wordexp()` — Perform Shell Word Expansions” on page 1774

## globfree() — Free Storage Allocate by glob()

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE  
#include <glob.h>
```

```
void globfree(glob_t *pglob);
```

### General Description

The `globfree()` function frees storage associated with *pglob* by a previous call to `glob()`.

### Returned Value

`globfree()` returns no value.

### Related Information

- “glob.h” on page 29
- “glob() — Generate Pathnames Matching a Pattern” on page 636

## gmtime() — Convert Time to Broken-Down UTC Time

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <time.h>
```

```
struct tm *gmtime(const time_t *timer);
```

### General Description

Converts the calendar time pointed to by *timer* into a broken-down time, expressed as coordinated universal time (UTC).<sup>2</sup>

The value pointed to by *timer* is usually obtained by a call to the `time()` function.

Table 27. Fields of the *tm* structure

Field	Description
<i>tm_sec</i>	Seconds (0-61)
<i>tm_min</i>	Minutes (0-59)
<i>tm_hour</i>	Hours (0-23)
<i>tm_mday</i>	Day of month (1-31)
<i>tm_mon</i>	Month (0-11; January = 0)
<i>tm_year</i>	Year (current year minus 1900)
<i>tm_wday</i>	Day of week (0-6; Sunday = 0)
<i>tm_yday</i>	Day of year (0-365; January 1 = 0)
<i>tm_isdst</i>	Zero if Daylight Saving Time is not in effect; positive if Daylight Saving Time is in effect; negative if the information is not available.

### Returned Value

Returns a pointer to a *tm* structure containing the broken-down time, expressed in coordinated universal time (UTC) corresponding to calendar time pointed to by *timer*. The fields in *tm* are shown in Table 27. If the calendar time pointed to by *timer* cannot be converted to broken-down time (in UTC), `gmtime()` returns a null pointer.

### Notes:

- The range (0-61) for *tm\_sec* allows for as many as two leap seconds.

<sup>2</sup> Coordinated Universal Time was formerly known as Greenwich Mean Time (GMT).

- The `gmtime()` and `localtime()` functions may use a common, statically allocated buffer for the conversion. Each call to one of these functions may alter the result of the previous call.
- The calendar time returned by the `time()` function begins at the epoch, which was at 00:00:00 Coordinated Universal Time, January 1, 1970.

### Example

#### CBC3BG23

```
/* CBC3BG23
   This example uses the gmtime() function to convert a time_t
   representation to a Coordinated Universal Time character string and
   then converts it to a printable string using asctime().
*/
#include <stdio.h>
#include <time.h>

int main(void)
{
    time_t ltime;
    time(&ltime);
    printf ("Coordinated Universal Time is %s\n",
           asctime(gmtime(&ltime)));
}
```

### Output

Coordinated Universal Time is Fri Jun 16 21:01:44 1995

### Related Information

- “time.h” on page 51
- “asctime() — Convert Time to Character String” on page 106
- “ctime() — Convert Time to a Character String” on page 250
- “localdtconv() — Date/Time Formatting Convention Inquiry” on page 754
- “localtime() — Convert Time and Correct for Local Time” on page 758
- “mktime() — Convert Local Time” on page 828
- “strftime() — Convert to Formatted Time” on page 1432
- “time() — Determine Current Time” on page 1570
- “tzset() — Set the Time Zone” on page 1637

## grantpt() — Grant Access to the Slave Pseudoterminal Device

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>
int grantpt(int fildev);
```

### General Description

The `grantpt()` function changes the mode and ownership of the slave pseudoterminal device. *fildev* should be the file descriptor of the corresponding master pseudoterminal. The user ID of the slave is set to the real UID of the calling process and the group ID is set to the group ID associated with the group name specified by the installation in the `TTYGROUP()` initialization parameter. The permission mode of the slave pseudoterminal is set to readable and writable by the owner, and writable by the group.

You can provide secure connections by either using `grantpt()` and `unlockpt()`, or by simply issuing the first open against the slave pseudoterminal from the first `userid` or process that opened the master terminal.

### Returned Value

Upon successful completion, `grantpt()` returns 0. Otherwise, it returns -1 and sets `errno` to indicate the error.

The `grantpt()` function will fail if:

- |        |  |
|--------|--|
| EACCES | The slave pseudoterminal was opened before <code>grantpt()</code> , or a <code>grantpt()</code> was already issued. In either case, slave pseudoterminal permissions and ownership have already been updated. If you use <code>grantpt()</code> to change slave pseudoterminal permissions, you must issue <code>grantpt()</code> between the master open and the first pseudoterminal open, and <code>grantpt()</code> can only be issued once. |
| EBADF  | The <i>fildev</i> argument is not a valid open file descriptor.  |
| EINVAL | The <i>fildev</i> argument is not associated with a master pseudoterminal device.  |
| ENOENT | The slave pseudoterminal device was not found during lookup.   |

### Related Information

- “`open()` — Open a File” on page 872
- “`ptsname()` — Get Name of the Slave Pseudoterminal Device” on page 1051
- “`unlockpt()` — Unlock a Pseudoterminal Master/Slave Pair” on page 1662



## hcreate() — Create Hash Search Tables

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <search.h>
```

```
int hcreate(size_t nel);
```

### General Description

The `hcreate()` function allocates sufficient space for a hash table containing *nel* elements, and must be called before `hsearch()` is used.

The *nel* argument is an estimate of the maximum number of entries that the table will contain. This number may be adjusted upward by `hcreate()` for the actual table allocation in order to obtain certain mathematically favorable circumstances.

Threading Behavior: see “`hsearch()` — Search Hash Tables” on page 647.

### Returned Value

The `hcreate()` function returns 0 if it cannot allocate sufficient space for the table, and returns nonzero otherwise.

The following are the possible values of `errno`:

**ENOMEM** Insufficient storage space is available.

### Related Information

- “`search.h`” on page 40
- “`bsearch()` — Search Arrays” on page 137
- “`hsearch()` — Search Hash Tables” on page 647
- “`hdestroy()` — Destroy Hash Search Tables” on page 644
- “`lsearch()` — Linear Search and Update” on page 775
- “`malloc()` — Reserve Storage Block” on page 786
- “`strcmp()` — Compare Strings” on page 1418
- “`tsearch()` — Binary Tree Search” on page 1617

## hdestroy() — Destroy Hash Search Tables

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <search.h>
```

```
void hdestroy(void);
```

### General Description

The `hdestroy()` function disposes of the search table, and may be followed by another call to `hcreate()`. After the call to `hdestroy()`, the data can no longer be considered accessible.

Threading Behavior: see “`hsearch()` — Search Hash Tables” on page 647.

### Returned Value

The `hdestroy()` function returns no value.

### Related Information

- “`search.h`” on page 40
- “`bsearch()` — Search Arrays” on page 137
- “`hsearch()` — Search Hash Tables” on page 647
- “`hcreate()` — Create Hash Search Tables” on page 643
- “`lsearch()` — Linear Search and Update” on page 775
- “`malloc()` — Reserve Storage Block” on page 786
- “`strcmp()` — Compare Strings” on page 1418
- “`tsearch()` — Binary Tree Search” on page 1617

## \_\_heaprpt() — Obtain Dynamic Heap Storage Report

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <stdlib.h>

typedef struct{int __uheap_size;
               int __uheap_bytes_alloc;
               int __uheap_bytes_free;
            } hreport_t;

int __heaprpt(hreport_t * heap_report_structure);
```

### General Description

\_\_heaprpt() returns to the caller the address of a structure that contains the user heap storage report. The storage report is similar in content to the user heap storage report that is generated with the RPTSTG(ON) Run-Time option.

To use this function, the calling program must obtain storage where the user's heap storage report will be stored. The address of this storage is passed as an argument to \_\_heaprpt().

### Returned Value

Upon successful completion, the struct hreport\_t will be filled with the user's heap storage report information. If the address is invalid, \_\_heaprpt() returns -1 and errno is set to EFAULT.

### Example

```
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    hreport_t * strptr;

    strptr = (hreport_t *) malloc(sizeof(hreport_t));

    if (__heaprpt(strptr) != 0)
        perror("__heaprpt() error");

    else
    {
        printf("Total amount of user heap storage      : %d\n",
              strptr->__uheap_size);
        printf("Amount of user heap storage in use       : %d\n",
              strptr->__uheap_bytes_alloc);
        printf("Amount of available user heap storage: %d\n",
              strptr->__uheap_bytes_free);
    }
}
```

`__heaprpt()`

### **Related Information**

- “stdlib.h” on page 45

## hsearch() — Search Hash Tables

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <search.h>
ENTRY *hsearch(ENTRY item, ACTION action);
```

### General Description

The `hsearch()` function is a hash-table search routine. It returns a pointer into a hash table indicating the location at which an entry can be found. The *item* argument is a structure of type `ENTRY` (defined in the `<search.h>` header) containing two pointers: *item.key* points to the comparison key (a `char *`), and *item.data* (a `void *`) points to any other data to be associated with that key. The comparison function used by `hsearch()` is `strcmp()`. The *action* argument is a member of an enumeration type `ACTION` indicating the disposition of the entry if it cannot be found in the table. *ENTER* indicates that the item should be inserted in the table at an appropriate point. *FIND* indicates that no entry should be made.

Threading Behavior: The `hcreate()` function allocates a piece of storage for use as the hash table. This storage is not exposed to the user, and is referred to by all threads. In other words, these functions operate on one hash table global to the process. The library serializes access to the table and attendant data across threads using an internal mutex.

### Returned Value

The `hsearch()` function returns a null pointer if either the action is *FIND* and the item could not be found or the action is *ENTER* and the table is full.

The following are the possible values of `errno`:

**ENOMEM** Insufficient storage space is available.

### Related Information

- “`search.h`” on page 40
- “`bsearch()` — Search Arrays” on page 137
- “`hcreate()` — Create Hash Search Tables” on page 643
- “`hdestroy()` — Destroy Hash Search Tables” on page 644
- “`lsearch()` — Linear Search and Update” on page 775
- “`malloc()` — Reserve Storage Block” on page 786
- “`strcmp()` — Compare Strings” on page 1418
- “`tsearch()` — Binary Tree Search” on page 1617

## htonl() — Translate Address Host to Network Long

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1

#include <arpa/inet.h>
in_addr_t htonl (in_addr_t hostlong);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <arpa/inet.h>
#include <netinet/in.h>

unsigned long htonl(unsigned long a);
```

### General Description

The `htonl()` call translates a long integer from host byte order to network byte order.

#### Parameter

#### Description

*a*                      The unsigned long integer to be put into network byte order.

`in_addr_t hostlong`      Is typed to the unsigned long integer to be put into network byte order.

#### Notes:

1. For OS/390, host byte order and network byte order are the same.
2. Since this function is implemented as a macro, you need one of the feature test macros and the `inet` header file.

### Returned Value

`htonl()` returns the translated long integer.

### Related Information

- “`htons()` — Translate an Unsigned Short Integer into Network Byte Order” on page 649
- “`ntohl()` — Translate a Long Integer into Host Byte Order” on page 870
- “`ntohs()` — Translate an Unsigned Short Integer into Host Byte Order” on page 871

## htons() — Translate an Unsigned Short Integer into Network Byte Order

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>
```

```
in_port_t htons(in_port_t hostshort);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
unsigned short htons(unsigned short a);
```

### General Description

The `htons()` call translates a short integer from host byte order to network byte order.

#### Parameter

#### Description

*a*

The unsigned short integer to be put into network byte order.

`in_port_t hostshort`

Is typed to the unsigned short integer to be put into network byte order.

#### Notes:

1. For OS/390, host byte order and network byte order are the same.
2. Since this function is implemented as a macro, you need one of the feature test macros and the `inet` header file.

### Returned Value

`htons()` returns the translated short integer.

### Related Information

- “`htonl()` — Translate Address Host to Network Long” on page 648
- “`ntohl()` — Translate a Long Integer into Host Byte Order” on page 870
- “`ntohs()` — Translate an Unsigned Short Integer into Host Byte Order” on page 871

## hypot() — Calculate Hypotenuse

### Standards

Standards / Extensions	C or C++	Dependencies
SAA XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double hypot(double side1, double side2);
```

### Compiler Option

LANGVL(SAA), LANGVL(SAAL2), or LANGVL(EXTENDED)

### General Description

Calculates the length of the hypotenuse of a right-angled triangle based on the lengths of two sides *side1* and *side2*. A call to hypot() is equal to:

```
sqrt(side1* side1 + side2 * side2);
```

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

The hypot() function returns the calculated length of the hypotenuse. If the correct value is outside the range of representable values, plus or minus HUGE\_VAL is returned, according to the sign of the value. The value of the macro ERANGE is stored in errno, to show the calculated value is out of range. If the correct value would cause an underflow, zero is returned and the value of the macro ERANGE is stored in errno.

### Example

#### CBC3BH01

```
/* CBC3BH01
   This example calculates the hypotenuse of a right-angled triangle with
   sides of 3.0 and 4.0.
*/
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, y, z;

    x = 3.0;
    y = 4.0;
    z = hypot(x,y);
    printf("The hypotenuse of the triangle with sides %lf and %lf"
           " is %lf\n", x, y, z);
}
```

### Output



The hypotenuse of the triangle with sides 3.000000 and 4.000000 is 5.000000

**Related Information**

- “math.h” on page 35
- “sqrt() — Calculate Square Root” on page 1396

## ibmsflush() — Flush the Application-side Datagram Queue

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#define _OPEN_SYS_SOCKET_EXT
#include <sys/socket.h>
```

```
int ibmsflush(int s);
```

### General Description

Bulk mode is supported only for receive-type socket calls. Currently, send-type socket calls are not supported for bulk mode. Until bulk mode send is supported, `ibmsflush()` simply returns to the calling program with a zero return code.

For outbound sockets, the application-side datagram queue is flushed (transferred to the TCP/IP address space) if any one of the following occur:

- An `ibmsflush()` is issued on the socket.
- The queue is full and another send-type socket call is issued.
- The socket is closed.
- Another `setibmssockopt()` is issued.

### Parameter Description

`s`            The socket descriptor.

### Returned Value

The value 0 indicates success. The value -1 indicates an error. The value of `errno` indicates the specific error.

### Errno Code      Description

EBADF            The `s` parameter is not a valid socket descriptor.

### Example

The following is an example of the `ibmsflush()` call.

```
char buffer[1000];
int rc, sizeofbuf;
struct ibm_bulkmode_struct mybulkstr;

/* Create, bind, etc done for socket sd */
.
.
.
mybulkstr.b_onoff = 1;
mybulkstr.b_max_receive_queue_size = 0;
mybulkstr.b_max_send_queue_size = 2100;
mybulkstr.b_move_data = 1;
rc = setibmssockopt(sd, SOL_SOCKET, SO_BULKMODE,
                    (char *)&mybulkstr, sizeof(mybulkstr));

strcpy( buffer, "Buffer info that fills up to 1000" );
sizeofbuf = 1000;
```

```

write(sd, buffer, sizeofbuf);

strcpy( buffer, "More buffer info that fills up to 1000" );
sizeofbuf = 1000;
write(sd, buffer, sizeofbuf);

strcpy( buffer, "Even more buffer info that fills up to 1000" );
sizeofbuf = 1000;
write(sd, buffer, sizeofbuf);

/* Issue ibmsflush() to make sure everything in buffer has been sent.*/
rc = ibmsflush(sd);
.
.
.

```

### Related Information

- “getibmssockopt() — Get the Options Associated with a Bulk Mode Socket” on page 543
- “setibmssockopt() — Set Options Associated with a Bulk Mode Socket” on page 1227

## iconv() — Code Conversion

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#include <iconv.h>
```

```
size_t iconv(iconv_t cd, char **inbuf, size_t *inbytesleft, char **outbuf,
             size_t *outbytesleft);
```

### General Description

Converts a sequence of characters from one encoded character set, and stores it into the array indirectly pointed to by *inbuf*, into a sequence of corresponding characters in another encoded character set, in the array indirectly pointed to by *outbuf*. The encoded character sets are those specified in the `iconv_open()` call that returned the conversion descriptor, *cd*. If the descriptor refers to the state-dependent encoding, then before it is first used, the *cd* descriptor is in its initial shift state.

The *inbuf* argument points to a variable that points to the first character in the input buffer. *inbytesleft* indicates the number of bytes to the end of the buffer to be converted. The *outbuf* argument points to a variable that points to the first character in the output buffer. *outbytesleft* indicates the number of available bytes to the end of the buffer.

If the output character set refers to the state-dependent encoding—if it contains the multibyte characters with shift-states—the conversion descriptor *cd* is placed in its initial state by a call for which *inbuf* is a null pointer, or for which *inbuf* points to a null pointer. When `iconv()` is called in this way, and if *outbuf* is not a null pointer or a pointer to a null pointer, and *outbytesleft* points to a positive value, `iconv()` places in the output buffer the byte sequence to change the output buffer to the initial shift state. If the output buffer is not large enough to hold the entire reset sequence, `iconv()` fails, and sets `errno` to `[E2BIG]`. Subsequent calls with *inbuf* as other than a null pointer or a pointer to a null pointer cause conversion from the current state of the conversion descriptor.

If a sequence of input bytes does not form a valid character in the specified encoded character set, conversion stops after the previous successfully converted character, and the `errno` is set to `EILSEQ`. If the input buffer ends with an incomplete character or shift sequence, conversion stops after the previous successfully converted bytes, and `errno` is set to `EINVAL`. If the output buffer is not large enough to hold the entire converted input, conversion stops just prior to the input bytes that would cause the output buffer to overflow.

The variable pointed to by *inbuf* is updated to point to the byte following the last byte of a successfully converted character. The value pointed to by *inbytesleft* is decremented to reflect the number of bytes still not converted in the input buffer. The variable pointed to by *outbuf* is updated to point to the byte following the last byte of converted output data. The value pointed to by *outbytesleft* is decremented to reflect the number of bytes still available in the output buffer. For state-

dependent encoding, the conversion descriptor is updated to reflect the shift state in effect at the end of the last successfully converted byte sequence.

If `iconv()` encounters a character in the input buffer that is valid, but for which a conversion is not defined in the conversion descriptor, *cd*, then `iconv()` performs a nonidentical conversion on this character. The conversion is implementation defined.

The `<iconv.h>` header file declares the `iconv_t` type that is a pointer to the object capable of storing the information about the converters used to convert characters in one coded character set to another. For state-dependent encoding, the object must be capable of storing the encoded information about the current shift state.

### Special Behavior for POSIX C

In the POSIX environment, a conversion descriptor returned from a successful `iconv_open` may be used safely within a single thread. In addition, it may be opened on one thread, used on a second thread (`iconv()`), and closed (`iconv_close()`) on a third thread. However, you must ensure correct cross-thread sequencing and synchronization (that is: `iconv_open()`, followed by optional `iconv()` calls, followed by `iconv_close()`). The use of a shared conversion descriptor by `iconv()` across multiple threads may result in undefined behavior.

See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information about using POSIX support.

### Returned Value

The `iconv()` function updates the variables pointed to by the arguments to reflect the extent of the conversion and returns the number of nonidentical conversions performed.

If the entire string in the input buffer is converted, the value pointed to by *inbytesleft* will be 0. If the input conversion is stopped because of any conditions mentioned above, the value pointed to by *inbytesleft* will be nonzero and `errno` is set to indicate the condition. If an error occurs, `iconv()` returns `size_t() - 1` and sets `errno` to indicate the error. If *cd* is not a valid descriptor, an error occurs and `errno` is set to `EBADF`.

### Example CBC3BI01

```
/* CBC3BI01
   This example converts an array of characters coded in encoded character
   set IBM-1047 to an array of characters coded in encoded character set
   IBM-037. Input is in inbuf, output will be in outbuf.
*/
#include <iconv.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

main ()
{
    char    *inptr; /* Pointer used for input buffer */
    char    *outptr; /* Pointer used for output buffer */
    char    inbuf[20] =
        "ABCDEFGH!@#$1234";
```

```

/* input buffer */
unsigned char  outbuf[20]; /* output buffer */
iconv_t        cd; /* conversion descriptor */
size_t         inleft; /* number of bytes left in inbuf */
size_t         outleft; /* number of bytes left in outbuf */
int            rc; /* return code of iconv() */

if ((cd = iconv_open("IBM-037", "IBM-1047")) == (iconv_t)(-1)) {
    fprintf(stderr, "Cannot open converter from %s to %s\n",
            "IBM-1047", "IBM-037");
    exit(8);
}

inleft = 20;
outleft = 20;
inptr = inbuf;
outptr = outbuf;

rc = iconv(cd, &inptr, &inleft, &outptr, &outleft);
if (rc == -1) {
    fprintf(stderr, "Error in converting characters\n");
    exit(8);
}
iconv_close(cd);
}

```

## Related Information

- “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*
- “locale.h” on page 33
- “iconv\_close() — Deallocate Code Conversion Descriptor” on page 657
- “iconv\_open() — Allocate Code Conversion Descriptor” on page 658
- “setlocale() — Set Locale” on page 1241

## iconv\_close() — Deallocate Code Conversion Descriptor

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#include <iconv.h>
```

```
int iconv_close(iconv_t cd);
```

### General Description

Deallocates the conversion descriptor *cd* and all other associated resources allocated by the `iconv_open()` function. For an illustration of using `iconv_open()`, see “Example” on page 655.

### Returned Value

Upon successful completion, a value of 0 is returned. Otherwise a value of –1 is returned and `errno` is set to indicate the error. If *cd* is not a valid descriptor, then an error occurs and `errno` is set to `EBADF`.

### Related Information

- “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*
- “locale.h” on page 33
- “iconv() — Code Conversion” on page 654
- “iconv\_open() — Allocate Code Conversion Descriptor” on page 658
- “setlocale() — Set Locale” on page 1241

## iconv\_open() — Allocate Code Conversion Descriptor

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#include <iconv.h>
```

```
iconv_t iconv_open(const char *tocode, const char *fromcode);
```

### General Description

Performs all the initialization needed to convert characters from the encoded character set specified in the array pointed to by the *fromcode* argument to the encoded character set specified in the array pointed to by the *tocode* argument.

The conversion descriptor relates the two encoded character sets.

For state-dependent encodings, the conversion descriptor will be in an encoded-character-set-dependent initial shift state, ready for immediate use with `iconv()`. The conversion descriptor remains valid until it is closed with `iconv_close()`.

Settings of *fromcode*, *tocode*, and their permitted combinations are implementation-dependent.

`iconv()` uses the following environment variables.

`_ICONV_UCS2` Tells `iconv_open(Y, X)` what type of conversion method to setup when there is a choice between "direct" conversion from X to Y and "indirect" X to UCS-2 to Y.

`_ICONV_UCS2_PREFIX` Tells `iconv_open()` what OS/390 dataset name prefix to use to find UCS-2 tables if they cannot be found in the HFS.

For illustration of using `iconv_close()`, see "Example" on page 655.

### Returned Value

The `iconv_open()` returns a conversion descriptor if successful; otherwise, it returns `(iconv_t)-1`, and it sets `errno` to indicate the error. If the conversion between encoded character sets specified is not supported, an error occurs, and `errno` is set to `EINVAL`, to show that the result of the conversion was invalid.

### Related Information

- "Internationalization: Locales and Character Sets" in the *OS/390 C/C++ Programming Guide*
- "locale.h" on page 33
- "iconv() — Code Conversion" on page 654
- "iconv\_close() — Deallocate Code Conversion Descriptor" on page 657
- "setlocale() — Set Locale" on page 1241



## ilogb() — Integer Unbiased Exponent

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>
```

```
int ilogb(double x);
```

### General Description

The `ilogb()` function returns the unbiased exponent of its argument `x` as an integer.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

If it succeeds, `ilogb()` returns the unbiased exponent of `x` as an integer. If `x` is equal to 0.0, `ilogb()` will return `INT_MIN`.

### Related Information

- “math.h” on page 35
- “logb() — Unbiased Exponent” on page 763

## index() — Search for Character

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <strings.h>
```

```
char *index(const char *string, int c);
```

### General Description

The `index()` function locates the first occurrence of `c` (converted to an unsigned char) in the string pointed to by `string`. The character `c` can be the null character (`\0`); the ending null is included in the search.

The string argument to the function must contain a null character (`\0`) marking the end of the string.

The `index()` function is identical to “`strchr()` — Search for Character” on page 1416.

### Returned Value

The `index()` function returns a pointer to the first occurrence of `c` (converted to an unsigned character) in the string pointed to by `string`. The function returns a null pointer if `c` was not found.

There are no `errno` values defined for `index()`.

### Related Information

- “`strings.h`” on page 46
- “`memchr()` — Search Buffer” on page 809
- “`rindex()` — Search for Character” on page 1145
- “`strchr()` — Search for Character” on page 1416
- “`strrchr()` — Find Last Occurrence of Character in String” on page 1449
- “`strspn()` — Search String” on page 1451
- “`strstr()` — Locate Substring” on page 1453

## inet\_addr() — Translate an Internet Address into Network Byte Order

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>

in_addr_t inet_addr(const char *cp);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr(char *cp);
```

### General Description

The `inet_addr()` call interprets character strings representing host addresses expressed in standard dotted-decimal notation and returns host addresses suitable for use as an Internet address.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Parameter Description

*cp*            A character string in standard dotted-decimal (.) notation.

Values specified in standard dotted-decimal notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When a 4-part address is specified, each part is interpreted as a byte of data and assigned, from left to right, to one of the 4 bytes of an Internet address.

When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the two rightmost bytes of the network address. This makes the three-part address format convenient for specifying class-B network addresses as **128.net.host**.

When a two-part address is specified, the last part is interpreted as a 24-bit quantity and placed in the three rightmost bytes of the network address. This makes the two-part address format convenient for specifying class-A network addresses as **net.host**.

When a one-part address is specified, the value is stored directly in the network address space without any rearrangement of its bytes.

Numbers supplied as address parts in standard dotted-decimal notation can be decimal, hexadecimal, or octal. Numbers are interpreted in C language syntax. A leading 0x implies hexadecimal; a leading 0 implies octal. A number without a leading 0 implies decimal.

### **Special Behavior for C++**

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### **Returned Value**

The Internet address is returned in network byte order. If the Internet address is returned in error—for example, not in the correct format—`INADDR_NONE` is the returned value. `INADDR_NONE` is defined in the **netinet/in.h** include file.

### **Related Information**

- “inet\_makeaddr() — Create an Internet Host Address” on page 664
- “inet\_netof() — Get the Network Number from the Internet Host Address” on page 665
- “inet\_network() — Get the Network Number from the Decimal Host Address” on page 666
- “inet\_ntoa() — Get the Decimal Internet Host Address” on page 667

## inet\_lnaof() — Translate a Local Network Address into Host Byte Order

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>

in_addr_t inet_lnaof(struct in_addr in);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_lnaof(struct in_addr in);
```

### General Description

The `inet_lnaof()` call breaks apart the Internet host address and returns the local network address portion.

#### Parameter Description

*in*            The host Internet address.

#### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

The local network address is returned in host byte order.

### Related Information

- “`inet_makeaddr()` — Create an Internet Host Address” on page 664
- “`inet_netof()` — Get the Network Number from the Internet Host Address” on page 665
- “`inet_network()` — Get the Network Number from the Decimal Host Address” on page 666
- “`inet_ntoa()` — Get the Decimal Internet Host Address” on page 667

## inet\_makeaddr() — Create an Internet Host Address

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>
```

```
struct in_addr inet_makeaddr(in_addr_t net, in_addr_t lna);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
struct in_addr inet_makeaddr(unsigned long net, unsigned long lna);
```

### General Description

The `inet_makeaddr()` call takes a network number and a local network address and constructs an Internet address.

#### Parameter Description

*net*            The network number.

*lna*            The local network address.

#### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

The Internet address is returned in network byte order.

### Related Information

- “`inet_lnaof()` — Translate a Local Network Address into Host Byte Order” on page 663
- “`inet_netof()` — Get the Network Number from the Internet Host Address” on page 665
- “`inet_network()` — Get the Network Number from the Decimal Host Address” on page 666
- “`inet_ntoa()` — Get the Decimal Internet Host Address” on page 667

## inet\_netof() — Get the Network Number from the Internet Host Address

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>

in_addr_t inet_netof(struct in_addr in);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_netof(struct addr_in in);
```

### General Description

The `inet_netof()` call breaks apart the Internet host address and returns the network number portion.

#### Parameter Description

*in*            The Internet address in network byte order.

#### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

The network number is returned in host byte order.

### Related Information

- “`inet_lnaof()` — Translate a Local Network Address into Host Byte Order” on page 663
- “`inet_makeaddr()` — Create an Internet Host Address” on page 664
- “`inet_ntoa()` — Get the Decimal Internet Host Address” on page 667

## inet\_network() — Get the Network Number from the Decimal Host Address

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>
```

```
in_addr_t inet_network(const char *cp);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
unsigned long inet_network(char cp);
```

### General Description

The `inet_network()` call interprets character strings representing addresses expressed in standard dotted-decimal notation and returns numbers suitable for use as a network number.

### Parameter Description

*cp*            A character string in standard, dotted decimal (.) notation.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

The network number is returned in host byte order.

### Related Information

- “`inet_lnaof()` — Translate a Local Network Address into Host Byte Order” on page 663
- “`inet_makeaddr()` — Create an Internet Host Address” on page 664
- “`inet_ntoa()` — Get the Decimal Internet Host Address” on page 667



## inet\_ntoa() — Get the Decimal Internet Host Address

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>

char *inet_ntoa(struct in_addr in);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char *inet_ntoa(struct in_addr in);
```

### General Description

The `inet_ntoa()` call returns a pointer to a string expressed in the dotted-decimal notation. `inet_ntoa()` accepts an Internet address expressed as a 32-bit quantity in network byte order and returns a string expressed in dotted-decimal notation.

Parameter	Description
<i>in</i>	The host Internet address.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

Returns a pointer to the Internet address expressed in dotted-decimal notation. The storage pointed to exists on a per-thread basis and is overwritten by subsequent calls.

### Related Information

- “`inet_addr()` — Translate an Internet Address into Network Byte Order” on page 661
- “`inet_lnaof()` — Translate a Local Network Address into Host Byte Order” on page 663
- “`inet_makeaddr()` — Create an Internet Host Address” on page 664
- “`inet_netof()` — Get the Network Number from the Internet Host Address” on page 665

- “inet\_network() — Get the Network Number from the Decimal Host Address” on page 666

## initgroups() — Initialize the Supplementary Group ID List for the Process

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 5.2.2

### Format

```
#define _OPEN_SYS
#include <sys/types.h>
#include <grp.h>
```

```
int initgroups(const char *user, const gid_t basegid);
```

### General Description

The `initgroups()` function obtains the supplementary group membership of *user*, and sets the current process supplementary group IDs to that list. The *basegid* is also included in the supplementary group IDs list.

The caller of this function must be a superuser or must specify the password of the target user name specified on the `initgroups()` call - issue the `password()` function prior to `initgroups()`.

### Returned Value

If successful, `initgroups()` returns 0.

If unsuccessful, `initgroups()` returns -1, and returns the error value in `errno`. The following are the possible values of `errno`:

EINVAL	The number of supplementary groups for the specified user plus the <i>basegid</i> group exceeds the maximum number of groups allowed, or an invalid <i>user</i> is specified.
EMVSERR	An MVS environmental or internal error occurred.
EMVSSAF2ERR	The System authorization facility (SAF) had an error.
EPERM	The caller is not authorized, only authorized users are allowed to alter the supplementary group IDs list.

### Related Information

- “`getgroupsbyname()` — Get Supplementary Group IDs by User Name” on page 529
- “`setgroups()` — Set the Supplementary Group ID List for the Process” on page 1223

## initstate() — Initialize Generator for Random()

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>
```

```
char *initstate(unsigned seed, char *state,
                size_t size);
```

### General Description

The `initstate()` function allows a state array, pointed to by the *state* argument, to be initialized for future use in calls to the `random()` functions by the calling thread. The *size* argument, which specifies the size in bytes of the state array, is used by the `initstate()` function to decide how sophisticated a random-number generator to use; the larger the state array, the more random the numbers. Values for the amount of state information are 8, 32, 64, 128, and 256 bytes. Amounts less than 8 bytes generate an error, while other amounts are rounded down to the nearest known value. The *seed* argument specifies a starting point for the random-number sequence and provides for restarting at the same point. The `initstate()` function returns a pointer to the previous state information array.

### Returned Value

If successful, `initstate()` returns a pointer to the previous state array. Otherwise, a null pointer is returned. The function fails and writes a message to standard error if it is called with *size* less than 8.

### Related Information

- “`stdlib.h`” on page 45
- “`random()` — A Better Random-Number Generator” on page 1079
- “`setstate()` — Change Generator for `random()`” on page 1275
- “`srandom()` — Use Seed to Initialize Generator for `random()`” on page 1400
- “`drand48()` — Pseudo-random Number Generator” on page 286
- “`random()` — A Better Random-Number Generator” on page 1079

## insque() — Insert an Element into a Doubly-linked List

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <search.h>
```

```
void insque(void *element, void *pred);
```

### General Description

The `insque()` function inserts the element pointed to by *element* into a doubly-linked list immediately after the element pointed to by *pred*. The function operates on pointers to structures which have a pointer to their successor in the list as their first element, and a pointer to their predecessor as the second. The application is free to define the remaining contents of the structure, and manages all storage itself. To insert the first element into a linear (non-circular) list, an application would call `insque(element, NULL)`; To insert the first element into a circular list, the application would set the element's forward and back pointers to point to the element.

### Returned Value

The `insque()` function returns no value.

### Related Information

- “search.h” on page 40
- “remque() — Remove an Element from a Doubly-linked List” on page 1135

## ioctl() — Control Device

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

#### Terminals

```
#include <sys/ioctl.h>
int ioctl(int fildev, int cmd, ... /* arg */);
```

#### Sockets

```
#define _XOPEN_SOURCE_EXTENDED 1

/**      OR      **/

#define _OE_SOCKETS

#include <sys/ioctl.h>
#include <net/rtroute.h>
#include <net/if.h>
int ioctl(int fildev, int cmd, ... /* arg */);
```

#### STREAMS

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stropts.h>
int ioctl(int fildev, int cmd, ... /* arg */);
```

### General Description

ioctl() performs a variety of control functions on devices. The *cmd* argument and an optional third argument (with varying type) are passed to and interpreted by the device associated with *fildev*.

The *cmd* argument selects the control function to be performed and will depend on the device being addressed.

The *arg* argument represents additional information that is needed by this specific device to perform the requested function. The type of *arg* depends upon the particular control request, but it is either an integer or a pointer to a device-specific data structure.

ioctl() information is divided into the following sections:

- Terminals
- Sockets
- STREAMS

## Terminals

The following ioctl() commands are used with terminals:

Command	Description
TIOCSWINSZ	Set window size. Used as the second operand in an ioctl() against a terminal. The window size information pointed to by the third operand is copied into an area in the kernel associated with the terminal, and a SIGWINCH signal is generated against the foreground process group.
TIOCGWINSZ	Get window size. Used as the second operand in an ioctl() against a terminal. The current window size is returned in the area pointed to by the third operand - a winsize structure.

The winsize structure is the third operand in an ioctl() call when you use TIOCSWINSZ or TIOCGWINSZ. The structure contains four unsigned short integers:

Field	Description
ws_row	Number of rows in the window, in characters.
ws_col	Number of columns in the window, in characters. This assumes single byte characters. Multibyte characters may take more room.
ws_xpixel	Horizontal size of the window, in pixels.
ws_ypixel	Vertical size of the window, in pixels.

## Sockets

The following ioctl() commands are used with sockets:

Command	Description
FIONBIO	Sets or clears nonblocking I/O for a socket. <i>arg</i> is a pointer to an integer. If the integer is 0, nonblocking I/O on the socket is cleared. Otherwise, the socket is set for nonblocking I/O.
FIONREAD	Gets the number of immediately readable bytes for the socket. <i>arg</i> is a pointer to an integer. Sets the value of the integer to the number of immediately readable characters for the socket.
FIOGETOWN	Returns the PID that has been set that designates the recipient of signals.
FIOSETOWN	Sets the PID to be used when sending signals  FIOGETOWN and FIOSETOWN are equivalent to the F_GETOWN and F_SETOWN commands of fctl(). For information on the values for pid, refer to that function. This function is only valid for AF_INET stream sockets.
SECIGET	Gets the peer socket's security identity values for an AF_UNIX connected stream socket. The MVS user ID, effective UID, and effective GID of the peer process are returned in the <b>seci</b> structure, which is mapped by BPXYSECI. This option is valid only for the AF_UNIX domain.

SIOCADDRT	Adds a routing table entry. <i>arg</i> is a pointer to a <b>rtentry</b> structure, as defined in <code>&lt;net/rtroute.h&gt;</code> . The routing table entry, passed as an argument, is added to the routing tables. This option is valid only for the AF_INET domain.
SIOCATMARK	Queries whether the current location in the data input is pointing to out-of-band data. <i>arg</i> is a pointer to an integer. SIOCATMARK sets the argument to 1 if the socket points to a mark in the data stream for out-of-band data; otherwise, it sets the argument to 0. Refer to <code>recv()</code> , <code>recvfrom()</code> and <code>recvmsg()</code> for more information on receiving out-of-band data.
SIOCDELRT	Deletes a routing table entry. <i>arg</i> is a pointer to a <b>rtentry</b> structure, as defined in <code>&lt;net/rtroute.h&gt;</code> . If it exists, the routing table entry passed as an argument is deleted from the routing tables. This option is valid only for the AF_INET domain.
SIOCGIFADDR	Gets the network interface address. <i>arg</i> is a pointer to an <b>ifreq</b> structure, as defined in <code>&lt;net/if.h&gt;</code> . The interface address is returned in the argument. This option is valid only for the AF_INET domain.
SIOCGIFBRDADDR	Gets the network interface broadcast address. <i>arg</i> is a pointer to an <b>ifreq</b> structure, as defined in <code>&lt;net/if.h&gt;</code> . The interface broadcast address is returned in the argument. This option is valid only for the AF_INET domain.
SIOCGIFCONF	Gets the network interface configuration. <i>arg</i> is a pointer to an <b>ifconf</b> structure, as defined in <code>&lt;net/if.h&gt;</code> . The interface configuration is returned in the buffer pointed to by the <b>ifconf</b> structure. The returned data's length is returned in the field that had originally contained the length of the buffer. This option is valid only for the AF_INET domain.
SIOCGIFDSTADDR	Gets the network interface destination address. <i>arg</i> is a pointer to an <b>ifreq</b> structure, as defined in <code>&lt;net/if.h&gt;</code> . The interface destination (point-to-point) address is returned in the argument. This option is valid only for the AF_INET domain.
SIOCGIFFLAGS	Gets the network interface flags. <i>arg</i> is a pointer to an <b>ifreq</b> structure, as defined in <code>&lt;net/if.h&gt;</code> . The interface flags are returned in the argument. This option is valid only for the AF_INET domain.
SIOCGIFMETRIC	Gets the network interface routing metric. <i>arg</i> is a pointer to an <b>ifreq</b> structure, as defined in <code>&lt;net/if.h&gt;</code> . The interface routing metric is returned in the argument. This option is valid only for the AF_INET domain.
SIOCGIFNETMASK	Gets the network interface network mask. <i>arg</i> is a pointer to an <b>ifreq</b> structure, as defined in <code>&lt;net/if.h&gt;</code> . The interface network mask is returned in the argument. This option is valid only for the AF_INET domain.



## SIOCGSPLXFQDN

Gets the fully qualified domain name for a given server and domain name in a sysplex. This is a special purpose command to support applications that have registered with WorkLoad Manager(WLM) for connection optimization services via the Domain Name System(DNS). 'arg' is a pointer to sysplexFqDn structure, as defined in <ezbzdnc.h>. sysplexFqDn contains pointer to sysplexFqDnData structure, as defined in <ezbzdnc.h>.

sysplexFqDnData structure contains server name(input), group name(input) and fully qualified domain name(output).

The ioctl() function with the SIOCGSPLXFQDN command will fail if:

- EFAULT Write user storage failed
- EINVAL One of the following:
  - Group name required
  - Buffer length invalid
  - Socket call parameter error
- ENXIO One of the following:
  - Sysplex address not found
  - Res not found In DNS
  - Time out
  - Time Unexpected Error

**Example**

The following is an example of the ioctl() call used with SIOCGSPLXFQDN.

```
#include <ezbzdnc.h>
sysplexFqDn      splxFqDn;
sysplexFqDnData  splxData;
int              rc;

splxFqDn.splxVersion = splxDataVersion;
splxFqDn.splxBufLen  = sizeof(sysplexFqDnData);
splxFqDn.splxBufAddr = &splxData;

/* Assign values to splxData.groupName, */
/* splxData.serverName if required      */
.
.

/* Get the fully qualified domain name */
rc = ioctl(s,SIOCGSPLXFQDN, (char *) &splxFqDn);

/* splxData.domainName contains the fully*/
/* qualified domain name.                  */
```

## SIOCSIFMETRIC

Sets the network interface routing metric. *arg* is a pointer to an **ifreq** structure, as defined in <net/if.h>. SIOCSIFMETRIC sets the interface routing metric to the value passed in the argument. This option is valid only for the AF\_INET domain.

**SIOCSVIPA** Defines or deletes a dynamic VIPA. *arg* is a pointer to a **dvreq** structure, as defined in <ezbzdvp.h>. This option is valid only for the AF\_INET domain.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Terminal and Sockets Returned Value

The value 0 indicates success; the value -1 indicates an error. The value of the error code indicates the specific error.

#### Error Code Description

EBADF	The <i>files</i> parameter is not a valid socket descriptor.
EINVAL	The request is invalid or not supported.
EIO	The process group of the process issuing the function is an orphaned, background process group, and the process issuing the function is not ignoring or blocking SIGTTOU.
EMVSPARM	Incorrect parameters were passed to the service.
ENODEV	The device is incorrect. The function is not supported by the device driver.
ENOTTY	An incorrect file descriptor was specified. The file type was not character special.

### Example

The following is an example of the `ioctl()` call.

```
int s;
int dontblock;
int rc;
:
/* Place the socket into nonblocking mode */
dontblock = 1;
rc = ioctl(s, FIONBIO, (char *) &dontblock);
:
```

### STREAMS

The following `ioctl()` commands are used with STREAMS:

**L\_PUSH** Pushes the module whose name is pointed to by *arg* onto the top of the current STREAM, just below the STREAM head. It then calls the *open()* function of the newly-pushed module.

The `ioctl()` function with the `L_PUSH` command will fail if:

EINVAL	Invalid module name.
ENXIO	Open function of new module failed.
ENXIO	Hang-up received on <i>files</i>

L_POP	<p>Removes the module just below the STREAM pointed to by <i>fildev</i>. The <i>arg</i> argument should be 0 in an L_POP request.</p> <p>The ioctl() function with the L_POP command will fail if:</p> <p>EINVAL No module present in the STREAM.</p> <p>ENXIO Hang-up received on <i>fildev</i>.</p>
L_LOOK	<p>Retrieves the name of the module just below the STREAM head of the STREAM pointed to by <i>fildev</i> and places it in a character string pointed to by <i>arg</i>. The buffer pointed to by <i>arg</i> should be at least FMNAMESZ+1 bytes long, where FMNAMESZ is defined in &lt;stropts.h&gt;.</p> <p>The ioctl() function with the L_LOOK command will fail if:</p> <p>EINVAL No module present in the STREAM.</p>
L_FLUSH	<p>This request flushes read and/or write queues, depending on the value of <i>arg</i>. Valid <i>arg</i> values are:</p> <p>FLUSHR Flush all read queues.</p> <p>FLUSHW Flush all write queues.</p> <p>FLUSHRW Flush all read and all write queues.</p> <p>The ioctl() function with the L_FLUSH command will fail if:</p> <p>EINVAL Invalid <i>arg</i> value.</p> <p>EAGAIN or ENOSR Unable to allocate buffers for flush message.</p> <p>ENXIO Hangup received on <i>fildev</i>.</p>
I_FLUSHBAND	<p>Flushes a particular band of messages. The <i>arg</i> argument points to a bandinfo structure. The bi_flag member may be one of FLUSHR, FLUSHW, or FLUSHRW as described above. The bi_pri member determines the priority band to be flushed.</p>
I_SETSIG	<p>Requests that the STREAMS implementation send the SIGPOLL signal to the calling process when a particular event has occurred on the STREAM associated with <i>fildev</i>. I_SETSIG supports an asynchronous processing capability in STREAMS. The value of <i>arg</i> is a bitmask that specifies the events for which the process should be signaled. It is the bitwise-OR of any combination of the following constants:</p> <p>S_RDNORM A normal (priority band set to 0) message has arrived at the head of a STREAM head read queue. A signal will be generated even if the message is of zero length.</p> <p>S_RDBAND A message with a nonzero priority band has arrived at the head of a STREAM head read queue. A signal will be generated even if the message is of zero length.</p>

S_INPUT	A message, other than a high-priority message, has arrived at the head of a STREAM head read queue. A signal will be generated even if the message is of zero length.
S_HIPRI	A high-priority message is present on a STREAM head read queue. A signal will be generated even if the message is of zero length.
S_OUTPUT	The write queue for normal data (priority band 0) just below the STREAM head is no longer full. This notifies the process that there is room on the queue for sending (or writing) normal data downstream.
S_WRNORM	Same as S_OUTPUT.
S_WRBAND	The write queue for a nonzero priority band just below the STREAM head is no longer full. This notifies the process that there is no room on the queue for sending (or writing) priority data downstream.
S_MSG	A STREAMS signal message that contains the SIGPOLL signal has reached the front of the STREAM head read queue.
S_ERROR	Notification of an error condition has reached the STREAM head.
S_HANGUP	When used in conjunction with S_RDBAND, SIGURG is generated instead of SIGPOLL when a priority message reaches the front of the STREAM head read queue.

If *arg* is 0, the calling process will be unregistered and will not receive further SIGPOLL signals for the STREAM associated with *fildev*.

Processes that wish to receive SIGPOLL signals must explicitly register to receive them using I\_SETSIG. If several processes register to receive this signal for the same event on the same STREAM, each process will be signaled when the event occurs.

The ioctl() function with the I\_SETSIG command will fail if:

EINVAL	The value of <i>arg</i> is invalid.
EINVAL	The value of <i>arg</i> is 0 and the calling process is not registered to receive the SIGPOLL signal.
EAGAIN	There were insufficient resources to store the signal request.

I\_GETSIG Returns the events for which the calling process is currently registered to be sent a SIGPOLL signal. The events are returned as a bitmask in an int pointed to by *arg*, where the events are those specified in the description of I\_SETSIG above.

The ioctl() function with the I\_GETSIG command will fail if:

	EINVAL	Process is not registered to receive the SIGPOLL signal.
I_FIND		<p>This request compares the names of all modules currently present in the STREAM to the name pointed to by <i>arg</i>, and returns 1 if the name module is present in the STREAM, or returns 0 if the named module is not present.</p> <p>The ioctl() function with the I_FIND command will fail if:</p> <p>EINVAL <i>arg</i> does not contain a valid module name.</p>
I_PEEK		<p>This request allows a process to retrieve the information in the first message on the STREAM head read queue without taking the message off the queue. It is analogous to <i>getmsg()</i> except that this command does not remove the message from the queue. The <i>arg</i> argument points to a <i>strpeek</i> structure.</p> <p>The <i>maxlen</i> member in the <i>ctlbuf</i> and <i>databuf</i> <i>strbuf</i> structure must be set to the number of bytes of control information and/or data information, respectively, to retrieve. The <i>flags</i> member may be marked RS_HIPRI or 0, as described by <i>getmsg()</i> - <i>getpmsg()</i>. If the process sets <i>flags</i> to RS_HIPRI, for example, I_PEEK will only look for a high-priority message on the STREAM head read queue.</p> <p>I_PEEK returns 1 if a message was retrieved, and returns 0 if no message was found on the STREAM head read queue, or if the RS_HIPRI flag was set in <i>flags</i> and a high-priority message was not present on the STREAM head read queue. It does not wait for a message to arrive. On return, <i>ctlbuf</i> specifies information in the control buffer, <i>databuf</i> specifies information in the data buffer, and <i>flags</i> contains the value RS_HIPRI or 0.</p>
I_SRDOPT		<p>Sets the read mode using the value of the argument <i>arg</i>. Read modes are described in <i>read()</i>. Valid <i>arg</i> flags are:</p> <p>RNORM Byte-stream mode, the default.</p> <p>RMSGD Message-discard mode.</p> <p>RMSGN Message-nondiscarded mode.</p> <p>The bitwise inclusive OR of RMSGD and RMSGN will return EINVAL. The bitwise inclusive OR of RNORM and either RMSGD or RMSGN will result in the other flag overriding RNORM which is the default.</p> <p>In addition, treatment of control messages by the STREAM head may be changed by setting any of the following flag in <i>arg</i>:</p> <p>RPROTNORM</p> <p style="padding-left: 40px;">Fail <i>read()</i> with EBADMSG if a message containing a control part is at the front of the STREAM head read queue.</p> <p>RPROTDAT Deliver the control part of a message as data when a process issues a <i>read()</i>.</p> <p>RPROTDIS Discard the control part of a message, delivery any data portion, when a process issues a <i>read()</i>.</p> <p>The ioctl() function with the I_SRDOPT command will fail if:</p> <p>EINVAL The <i>arg</i> argument is not valid.</p>

I_GRDOPT	Returns the current read mode setting, as described above, in an int pointed to by the argument <i>arg</i> . Read modes are described in <i>read()</i> .
I_NREAD	Counts the number of data bytes in the data part of the first message on the STREAM head read queue and places this value in the int pointed to by <i>arg</i> . The return value for the command is the number of messages on the STREAM head read queue. For example, if 0 is returned in <i>arg</i> , but the <i>ioctl()</i> return value is greater than 0, this indicates that a zero-length message is next on the queue.
I_FDINSERT	Creates a message from specified buffer(s), adds information about another STREAM, and sends the message downstream. The message contains a control part and an optional data part. The data and control parts to be sent are distinguished by placement in separate buffers, as described below. The <i>arg</i> argument points to a <b>strfdinsert</b> structure.

The **len** member in the *ctlbuf* *strbuf* structure must be set to the size of a pointer plus the number of bytes of control information to be sent with the message. The *fildev* member specifies the file descriptor of the other STREAM, and the *offset* member, which must be suitably aligned for use as a pointer, specifies the offset from the start of the control buffer where I\_FDINSERT will store a pointer whose interpretation is specific to the STREAM end. The **len** member in the *databuf* *strbuf* structure must be set to the number of bytes of data information to be sent with the message, or 0 if no data part is to be sent.

The **flags** member specifies the type of message to be created. A normal message is created if **flags** is set to 0, and a high-priority message is created if **flags** is set to RS\_HIPRI. For non-priority messages, I\_FDINSERT will block if the STREAM write queue is full due to internal flow control conditions. For priority messages, I\_FDINSERT does not block on this condition. For non-priority messages, I\_FDINSERT does not block when the write queue is full and O\_NONBLOCK is set. Instead, it fails and sets *errno* to EAGAIN.

I\_FDINSERT also blocks, unless prevented by lack of internal resources, waiting for the availability of message blocks in the STREAM, regardless of priority or whether O\_NONBLOCK has been specified. No partial message is sent.

The *ioctl()* function with the I\_FDINSERT command will fail if:

EAGAIN	A non-priority message is specified, the O_NONBLOCK flag is set, and the STREAM write queue is full due to internal flow control conditions.
EAGAIN or ENOSR	Buffers can not be allocated for the message that is to be created.
EINVAL	One of the following: <ul style="list-style-type: none"> <li>The <i>fd</i> member of the <b>strfdinsert</b> structure is not a valid, open STREAM file descriptor.</li> </ul>

- The size of a pointer plus *offset* is greater than the *len* member for the buffer specified through *ctlptr*
- the *offset* member does not specify a properly-aligned location in the data buffer.
- An undefined value is stored in **flags**

ENXIO Hang-up received for *fd* or *filides*.

ERANGE The *len* member for the buffer specified through *databuf* does not fall within the range specified by the maximum and minimum packet sizes of the topmost STREAM module or the *len* member for the buffer specified through *databuf* is larger than the maximum configured size of the data part of a message; or the *len* member for the buffer specified through *ctlbuf* is larger than the maximum configured size of the control part of a message.

## I\_STR

Constructs an internal STREAMS ioctl() message from the data pointed to by *arg*, and sends that message downstream.

This mechanism is provided to send ioctl() requests to downstream modules and drivers. It allows information to be sent with ioctl(), and returns to the process any information sent upstream by the downstream recipient. I\_STR blocks until the system responds with either a positive or negative acknowledgement message, or until the request "times out" after some period of time. If the request times out, it fails with *errno* set to ETIME.

At most, one I\_STR can be active on a STREAM. Further I\_STR calls will block until the active I\_STR completes at the STREAM head. The default timeout interval for these requests is 15 seconds. The O\_NONBLOCK flag has no effect on this call.

To send requests downstream, *arg* must point to a **strioc** structure.

The **ic\_cmd** member is the internal ioctl() command intended for a downstream module or driver and **ic\_timeout** is the number of seconds (-1 = infinite, 0 = use implementation-dependent timeout interval, >0 = as specified) an I\_STR request will wait for acknowledgement before timing out. **ic\_len** member has two uses: on input, it contains the length of the data argument passed in, and on return from the command, it contains the number of bytes being returned to the process (the *buffer* pointed to by **ic\_dp** should be large enough to contain the maximum amount of data that any module or the driver in the STREAM can return.)

The STREAM head will convert the information pointed to by the **strioc** structure to an internal ioctl() command message and send it downstream.

The ioctl() function with the I\_STR command will fail if:

EAGAIN or ENOSR

Unable to allocate buffers for the ioctl() message.

- EINVAL This *ic\_len* member is less than 0 or larger than the maximum configured size of the data part of a message, or *ic\_timeout* is less than -1.
- ENXIO Hang-up received on *fildev*.
- ETIME A downstream ioctl() timed out before acknowledgement was received.

An I\_STR can also fail while waiting for an acknowledgement if a message indicating an error or a hang-up is received at the STREAM head. In addition, an error code can be returned in the positive or negative acknowledgement message, in the event the ioctl() command sent downstream fails. For these cases, I\_STR fails with *errno* set to the value in the message.

- I\_SWROPT Sets the write mode using the value of the argument *arg*. Valid bit settings for *arg* are:
  - SNDZERO Send a zero-length message downstream when a *write()* of 0 bytes occurs. To not send a zero-length message when a *write()* of 0 bytes occurs, this bit must not be set in *arg* (for example, *arg* would be set to 0).

The ioctl() function with the I\_SWROPT command will fail if:

- EINVAL *arg* is not the above value.

- I\_GWROPT Returns the current write mode setting as described above, in the int that is pointed to by the argument *arg*.

- I\_SENDFD I\_SENDFD creates a new reference to the open file description associated with the file descriptor *arg* and writes a message on the STREAMS-based pipes *fildev* containing the reference, together with the user ID and group ID of the calling process.

The ioctl() function with the I\_SENDFD command will fail if:

- EAGAIN The sending STREAM is unable to allocate a message block to contain the file pointer; or the read queues of the receiving STREAM head is full and cannot accept the message sent by I\_SENDFD.
- EBADF The *arg* argument is not a valid, open file descriptor.
- EINVAL The *fildev* argument is not connected to a STREAM pipe.
- ENXIO Hangup received on *fildev*.

- I\_RECVFD Retrieves the reference to an open file description from a message within a STREAMS-based pipe using the I\_SENDFD command, and allocates a new file descriptor in the calling process that refers to this open file description. The *arg* argument is a pointer to an **strrecvfd** data structure as defined in <stropts.h>.

The **fd** member is a file descriptor. The **uid** and **gid** members are the effective user ID and effective group ID, respectively, of the sending process.

If O\_NONBLOCK is not set I\_RECVFD blocks until a message is present at the STREAM head. If O\_NONBLOCK is set, I\_RECVFD



fails with *errno* set to EAGAIN if no message is present at the STREAM head.

If the message at the STREAM head is a message sent by an I\_SENDFD, a new file descriptor is allocated for the open file descriptor referenced in the message. The new file descriptor is placed in the **fd** member of the **strrecvfd** structure pointed to by *arg*.

The ioctl() function with the I\_RECVFD command will fail if:

- EAGAIN      A message is not present at the STREAM head read queue and the O\_NONBLOCK flag is set.
- EBADMSG    The message at the STREAM head read queue is not a message containing a passed file descriptor.
- EMFILE      The process has the maximum number of file descriptors currently open that is allowed.
- ENXIO      Hang-up received on *fildev*.

## I\_LIST

This request allows the process to list all the module names on the STREAM, up to and including the topmost driver names. If *arg* is a null pointer, the return value is the number of modules, including the driver, that are on the STREAM pointed to by *fildev*. This lets the process allocate enough space for the module names. Otherwise, it should point to an **str\_list** structure.

The **sl\_nmods** member indicates the number of entries the process has allocated in the array. Upon return, the **sl\_modlist** member of the **str\_list** structure contains the list of module names. The number of entries that have been filled into the **sl\_modlist** array is found in the **sl\_nmode** member (the number includes the number of module including the driver). The return value from ioctl() is 0. The entries are filled in starting at the top of the STREAM and continuing downstream until either the end of the STREAM is reached, or the number of requested modules (**sl\_nmods**) is satisfied.

The ioctl() function with the I\_LIST command will fail if:

- EINVAL      The **sl\_nmods** member is less than 1.
- EAGAIN or ENOSR      Unable to allocate buffers.

## I\_ATMARK

This request allows the process to see if the message at the head of the STREAM head read queue is marked by some module downstream. The *arg* argument determines how the checking is done when there may be multiple marked messages on the STREAM head read queue. It may take on the following values:

ANYMARK Check if the message is marked.

LASTMARK Check if the message is the last one marked on the queue.

The bitwise inclusive OR of the flags ANYMARK and LASTMARK is permitted.

The return value is 1 if the mark condition is satisfied and 0 otherwise.

The `ioctl()` function with the `I_ATMARK` command will fail if:

`EINVAL` Invalid *arg* value.

**I\_CKBAND** Check if the message of a given priority band exists on the STREAM head read queue. This returns 1 if a message of the given priority exists, 0 if no message exists, or -1 on error. *arg* should be of type `int`.

The `ioctl()` function with the `I_CKBAND` command will fail if :

`EINVAL` Invalid *arg* value.

**I\_GETBAND** Return the priority band of the first message on the STREAM head read queue in the integer referenced by *arg*.

The `ioctl()` function with the `I_GETBAND` command will fail if:

`ENODATA` No message on the STREAM head read queue.

**I\_CANPUT** Check if a certain band is writable. *arg* is set to the priority band in question. The return value is 0 if the band is flow-controlled, 1 if the band is writable, or -1 on error.

The `ioctl()` function with the `I_CANPUT` command will fail if:

`EINVAL` Invalid *arg* value.

**I\_SETCLTIME** This request allows the process to set the time the STREAM head will delay when a STREAM is closing and there is data on the write queues. Before closing each module or driver, if there is a data on its write queue, the STREAM head will delay for the specified amount of time to allow the data to drain. If, after the delay, data is still present, it will be flushed. The *arg* argument is a pointer to an integer specifying the number of milliseconds to delay, rounded up to the nearest valid value. If `I_SETCLTIME` is not performed on a STREAM, an implementation-dependent default timeout interval is used.

The `ioctl()` function with the `I_SETCLTIME` command will fail if:

`EINVAL` Invalid *arg* value.

**I\_GETCLTIME** This request returns the close time delay in the integer pointed to by *arg*

### Multiplexed STREAMS Configurations.

The following four commands are used for connecting and disconnecting multiplexed STREAMS configurations. These commands use an implementation-dependent default timeout interval.

**I\_LINK** Connects two STREAMS, where *fildev* is the file descriptor of the STREAM connected to the multiplexing driver, and *arg* is the file descriptor of the STREAM connected to another driver. The STREAM designated by *arg* gets connected below the multiplexing driver. `I_LINK` requires the multiplexing driver to send an acknowledgement message to the STREAM head regarding the connection. This call returns a multiplexer ID number (an identifier used to disconnect the multiplexer; see `(I_UNLINK)` on success, and -1 on failure.

The `ioctl()` function with the `I_LINK` command will fail if:

- `ENXIO` Hang-up received on *fildes*.
- `ETIME` Time out before acknowledgement message was received at STREAM head.
- `EAGAIN` or `ENOSR`  
Unable to allocate STREAMS storage to perform the `I_LINK`.
- `EBADF` The *arg* argument is not a valid, open file descriptor.
- `EINVAL` The *fildes* does not support multiplexing; or *arg* is not a STREAM or is already connected downstream from a multiplexer, or the specified `I_LINK` operation would connect the STREAM head in more than one place in the multiplexed STREAM.

An `I_LINK` can also fail while waiting for the multiplexing driver to acknowledge the request, if a message indicating an error or a hang-up is received at the STREAM head of *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, `I_LINK` fails with *errno* set to the value in the message.

**`I_UNLINK`** Disconnects the two STREAMs specified by *fildes* and *arg*. *fildes* is the file descriptor of the STREAM connected to the multiplexing driver. The *arg* argument is the multiplexer ID number that was returned by the `I_LINK` `ioctl()` command when a STREAM was connected downstream from the multiplexing driver. If *arg* is `MUXID_ALL`, then all STREAMs that were connected to *fildes* are disconnected. As in `I_LINK`, this command requires acknowledgement.

The `ioctl()` function with the `I_UNLINK` command will fail if:

- `ENXIO` Hang-up received on *fildes*.
- `ETIME` Time out before acknowledgement message was received at STREAM head.
- `EAGAIN` or `ENOSR`  
Unable to allocate buffers for the acknowledgement message.
- `EINVAL` Invalid multiplexer ID number.

An `I_UNLINK` can also fail while waiting for the multiplexing driver to acknowledge the request if a message indicating an error or a hang-up is received at the STREAM head of *fildes*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, `I_UNLINK` fails with *errno* set to the value in the message.

**`I_PLINK`** Creates a *persistent connection* between two STREAMs, where *fildes* is the file descriptor of the STREAM connected to another driver. This call creates a persistent connection which can exist even if the file descriptor *fildes* associated with the upper STREAM to the multiplexing driver is closed. The STREAM designated by *arg* gets connected via a persistent connection below the multiplexing

driver. `I_PLINK` requires the multiplexing driver to send an acknowledgement message to the STREAM head. This call returns a multiplexer ID number (an identifier that may be used to disconnect the multiplexer, see `I_PUNLINK`) on success, and -1 on failure.

The `ioctl()` function with the `I_PLINK` command will fail if:

- `ENXIO` Hang-up received on *fildev*.
- `ETIME` Time out before acknowledgement message was received at STREAM head.
- `EAGAIN` or `ENOSR`  
Unable to allocate STREAMS storage to perform the `I_PLINK`.
- `EBADF` The *arg* argument is not valid, open file descriptor.
- `EINVAL` The *fildev* argument does not support multiplexing; or *arg* is not a STREAM or is already connected downstream from a multiplexer; or the specified `I_PLINK` operation would connect the STREAM head in more than one place in the multiplexed STREAM.

An `I_PLINK` can also fail while waiting for the multiplexing driver to acknowledge the request, if a message indicating an error or a hang-up is received at the STREAM head of *fildev*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, `I_PLINK` fails with *errno* set to the value in the message.

**`I_PUNLINK`** Disconnects the two STREAMs specified by *fildev* and *arg* from a persistent connection. The *fildev* argument is the file descriptor of the STREAM connected to the multiplexing driver. The *arg* argument is the multiplexer ID number that was returned by the `I_PLINK` `ioctl()` command when a STREAM was connected downstream from the multiplexing driver. If *arg* is `MUXID_ALL` then all STREAMs which are persistent connections to *fildev* are disconnected. As in `I_PLINK`, this command requires the multiplexing driver to the acknowledge the request.

The `ioctl()` function with the `I_PUNLINK` command will fail if:

- `ENXIO` Hang-up received on *fildev*.
- `ETIME` Time out before acknowledgement message was received at STREAM head.
- `EAGAIN` or `ENOSR`  
Unable to allocate buffers for the acknowledgement message.
- `EINVAL` Invalid multiplexer ID number.

An `I_PUNLINK` can also fail while waiting for the multiplexing driver to acknowledge the request if a message indicating an error or a hang-up is received at the STREAM head of *fildev*. In addition, an error code can be returned in the positive or negative acknowledgement message. For these cases, `I_PUNLINK` fails with *errno* set to the value in the message.

## STREAMS Returned Value

Upon successful completion, `ioctl()` returns a value other than -1 that depends upon the STREAMS device control function. Otherwise, it returns -1 and sets `errno` to indicate the error.

Under the following general conditions, `ioctl()` will fail if:

**Note:** It is impossible for `ioctl()` to perform any STREAMS type commands successfully, since OS/390 UNIX services do not provide any STREAMS-based files. The function will always return -1 with `errno` set to indicate the failure. See “`open()` — Open a File” on page 872 for more information.

- EBADF    The *fildev* argument is not a valid open file descriptor.
- EINTR    A signal was caught during the `ioctl()` operation.
- EINVAL   The STREAM or multiplexer referenced by *fildev* is linked (directly or indirectly) downstream from a multiplexer.

If an underlying device driver detects an error, the `ioctl()` will fail if:

- EINVAL   The *cmd* or *arg* argument is not valid for this device.
- EIO       Some physical I/O error has occurred.
- ENOTTY   The *fildev* argument is not associated with a STREAMS device that accepts control functions.
- ENXIO    The *cmd* or *arg* argument is not valid for this device driver, but the service requested can not be performed on this particular sub-device.
- ENODEV   The *fildev* argument refers to a valid STREAMS device, but the corresponding device driver does not support the `ioctl()` function.

If a STREAM is connected downstream from a multiplexer, any `ioctl()` command except `I_UNLINK` and `I_PUNLINK` will set `errno` to `EINVAL`.

## Related Information

- “`close()` — Close a File” on page 191
- “`fcntl()` — Control Open File Descriptors” on page 350
- “`getmsg()` - `getpmsg()` — Receive Next Message from a STREAMS File” on page 555
- “`open()` — Open a File” on page 872
- “`pipe()` — Create an Unnamed Pipe” on page 907
- “`poll()` — Monitor Activity on File Descriptors and Message Queues” on page 910
- “`putmsg()` - `putpmsg()` — Send a Message on a STREAM” on page 1056
- “`read()` — Read From a File or Socket” on page 1080
- “`sigaction()` — Examine or Change a Signal Action” on page 1295
- “`ezbzdnc.h`” on page 28
- “`write()` — Write Data on a File or Socket” on page 1780

## \_\_ipdbcs() — Retrieve the List of Requested DBCS Tables to Load

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#include <__ftp.h>
```

```
struct __ipdbcss *__ipdbcs(void);
```

### General Description

The `__ipdbcs()` function determines the values that IP address resolution initialization found in the resolver configuration data set for the keywords `LoadDBCSTables`. If the `LoadDBCSTables` keywords are not found in the resolver configuration data set, the structure returned has a count of zero and each element in the structure list points to a null string.

### Returned Value

If successful, `__ipdbcs()` returns the complete structure `__ipdbcss` with each entry in `__ip_dbcs_list[]` initialized either to a valid name or to a null string. The number of valid names, up to the maximum of 8, is placed in `__ipdbcsnum`. If no table names are specified then `__ipdbcsnum` is set to zero.

If unsuccessful, `__ipdbcs()` returns a null value. The `__ipdbcs()` function is only unsuccessful if IP Address Resolution initialization fails to complete.

### Related Information

- “`__ftp.h`” on page 29
- “`__ipdspc()` — Retrieve the Data Set Prefix Specified” on page 689
- “`__ipmsgc()` — Determine the Case to Use for FTP Messages” on page 691

## \_\_ipdspx() — Retrieve the Data Set Prefix Specified

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#include <__ftp.h>
```

```
char *__ipdspx(void);
```

### General Description

The \_\_ipdspx() function determines the value that IP address resolution initialization found in the resolver configuration data set for the keyword DataSetPrefix. If no DataSetPrefix keyword is found in the resolver configuration data set, then the default value is returned.

### Returned Value

If successful, \_\_ipdspx() return the null terminated string that was supplied in the configuration data set. If the configuration data set did not supply a value for the keyword DataSetPrefix then \_\_ipdspx() returns the string TCPIP.

If unsuccessful, \_\_ipdspx() returns null value. The \_\_ipdspx() function is only unsuccessful if IP Address Resolution initialization fails to complete.

### Related Information

- “\_\_ftp.h” on page 29
- “\_\_ipmsgc() — Determine the Case to Use for FTP Messages” on page 691
- “\_\_ipdbcs() — Retrieve the List of Requested DBCS Tables to Load” on page 688

## \_\_iphost() — Retrieve the Resolver Supplied Hostname

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R3

### Format

```
#include <__ftp.h>
```

```
char *__iphost(void);
```

### General Description

The \_\_iphost() function lets an application determine the values that IP address resolution initialization found in the resolver configuration data set for the keyword HOSTname. If the keyword is not found in the resolver configuration data set, the char string returned will be a null string.

### Returned Value

If successful, \_\_iphost() returns the null terminated character string, which is the name supplied on the HOSTname keyword found in the resolver configuration file.

If unsuccessful, \_\_iphost() returns a null value. The \_\_iphost() function is only unsuccessful if IP Address Resolution initialization fails to complete.

### Related Information

- “\_\_ftp.h” on page 29
- “\_\_ipdbcs() — Retrieve the List of Requested DBCS Tables to Load” on page 688
- “\_\_ipdspx() — Retrieve the Data Set Prefix Specified” on page 689
- “\_\_ipmsgc() — Determine the Case to Use for FTP Messages” on page 691
- “\_\_ipnode() — Retrieve the Resolver Supplied Node Name” on page 692
- “\_\_iptcpn() — Retrieve the Resolver Supplied Jobname or Userid” on page 693



## \_\_ipmsgc() — Determine the Case to Use for FTP Messages

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
int __ipmsgc(void);
```

### General Description

The \_\_ipmsgc() function determines the value that IP address resolution initialization found in the resolver configuration data set for the keyword MessageCase. If no MessageCase keyword is found in the resolver configuration data set, then the default value is returned.

The *init* argument returned is one of the following set of symbols defined in the \_\_ftp.h header file, each one stands for a message case selection.

\_\_MIXED Represents mixed case value selected for the messages FTP will send.

\_\_UPPER

Represents upper case value selected for the messages FTP will send.

### Returned Value

\_\_ipmsgc() is always successful and returns either the value of the \_\_MIXED or the value of \_\_UPPER for all requests. \_\_MIXED is the default value.

### Related Information

- “\_\_ftp.h” on page 29
- “\_\_ipdbcs() — Retrieve the List of Requested DBCS Tables to Load” on page 688
- “\_\_ipdspx() — Retrieve the Data Set Prefix Specified” on page 689

## \_\_ipnode() — Retrieve the Resolver Supplied Node Name

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R3

### Format

```
#include <__ftp.h>
```

```
char *__ipnode(void);
```

### General Description

The \_\_ipnode() function lets an application determine the values that IP address resolution initialization found as the NodeID name used by the VMCF platform. If the VMCF nodeID name is not found, the char string returned will be a null string.

### Returned Value

If successful, \_\_ipnode() returns the null terminated character string, which is the name found for the VMCF platform.

If unsuccessful, \_\_ipnode() returns a null value. The \_\_ipnode() function is only unsuccessful if IP Address Resolution initialization fails to complete.

### Related Information

- “\_\_ftp.h” on page 29
- “\_\_ipdbcs() — Retrieve the List of Requested DBCS Tables to Load” on page 688
- “\_\_ipdspx() — Retrieve the Data Set Prefix Specified” on page 689
- “\_\_iphost() — Retrieve the Resolver Supplied Hostname” on page 690
- “\_\_ipmsgc() — Determine the Case to Use for FTP Messages” on page 691
- “\_\_iptcpn() — Retrieve the Resolver Supplied Jobname or Userid” on page 693

## \_\_iptcpn() — Retrieve the Resolver Supplied Jobname or Userid

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#include <__ftp.h>
```

```
char *__iptcpn(void);
```

### General Description

The \_\_iptcpn() function lets an application determine the values that IP address resolution initialization found in the resolver configuration data set for either of the keywords TCPIPuserid or TCPIPjobname. Whichever is the last one read. If neither keyword is found in the resolver configuration data set, the char string returned will be a null string.

### Returned Value

If successful, \_\_iptcpn() returns the null terminated character string which is the name supplied on the TCPIPuserid or TCPIPjobname keyword found in the resolver configuration file.

If unsuccessful, \_\_iptcpn() returns a null value. The \_\_iptcpn() function is only unsuccessful if IP Address Resolution initialization fails to complete.

### Related Information

- “\_\_ftp.h” on page 29
- “setibmopt() — Set IBM TCP/IP Image” on page 1225

## isalnum() to isxdigit() — Test Integer Value

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <ctype.h>
```

```
int isalnum(int c);
int isalpha(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
```

### General Description

The functions listed above, which are all declared in `ctype.h`, test a given integer value. The valid integer values for *c* are those representable as an *unsigned char* or EOF.

The functions in `ctype.h` are also available as macros. For better performance, the macro forms are recommended over the functional forms.

However, to get the functional forms, do one or more of the following:

- For C only: do *not* include `ctype.h`.
- Specify `#undef`, for example, `#undef islower`
- Surround the call statement by parentheses, for example, `(islower)('a')`

Here are descriptions of each function in this group.

- `isalnum()` Test for an upper- or lowercase letter, or a decimal digit, as defined in the `alnum` locale source file and in the `alnum` class of the `LC_CTYPE` category of the current locale.
- `isalpha()` Test for an alphabetic character, as defined in the `alpha` locale source file and in the `alpha` class of the `LC_CTYPE` category of the current locale.
- `iscntrl()` Test for any control character, as defined in the `cntrl` locale source file and in the `cntrl` class of the `LC_CTYPE` category of the current locale.
- `isdigit()` Test for a decimal digit, as defined in the `digit` locale source file and in the `digit` class of the `LC_CTYPE` category of the current locale.

- isgraph() Test for a printable character excluding space, as defined in the graph locale source file and in the graph class of the LC\_CTYPE category of the current locale.
- islower() Test for a lowercase character, as defined in the lower locale source file and in the lower class of the LC\_CTYPE category of the current locale.
- isprint() Test for a printable character including space, as defined in the print locale source file and in the print class of the LC\_CTYPE category of the current locale.
- ispunct() Test for any nonalphanumeric printable character, excluding space, as defined in the punct locale source file and in the punct class of the LC\_CTYPE category of the current locale.
- isspace() Test for a white-space character, as defined in the space locale source file and in the space class of the LC\_CTYPE category of the current locale.
- isupper() Test for an uppercase character, as defined in the upper locale source file and in the upper class of the LC\_CTYPE category of the current locale.
- isxdigit() Test for a hexadecimal digit, as defined in the xdigit locale source file and in the xdigit class of the LC\_CTYPE category of the current locale.

The space, uppercase, and lowercase characters can be redefined by their respective class of the LC\_CTYPE in the current locale. The LC\_CTYPE category is discussed in the “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*.

To provide an ASCII input/output format for applications using these functions, define feature test macro `__LIBASCII` as described on page 22.

## Returned Value

These functions return a nonzero value if the integer satisfies the test condition, or a 0 value if it does not.

## Example

### CBC3BI02

```

/* CBC3BI02
   This example analyzes all characters between code 0x0 and code UPPER_LIMIT.
   The output of this example is a 256-line table showing the characters
   from 0 to 255, and a notification of whether they have the attributes tested.
*/
#include <stdio.h>
#include <ctype.h>

#define UPPER_LIMIT 0xFF

int main(void)
{
    int ch;

    for ( ch = 0; ch <= UPPER_LIMIT; ++ch )
    {
        printf("%3d ", ch);
        printf("%#04x ", ch);
        printf(" %c", isprint(ch) ? ch : ' ');
        printf("%3s ", isalnum(ch) ? "Alphanumeric" : " ");
    }
}

```

```
printf("%2s ", isalpha(ch) ? "Alphabetic" : " ");
printf("%2s", iscntrl(ch) ? "Control" : " ");
printf("%2s", isdigit(ch) ? "Digit" : " ");
printf("%2s", isgraph(ch) ? "Graphic" : " ");
printf("%2s ", islower(ch) ? "Lower" : " ");
printf("%3s", ispunct(ch) ? "Punctuation" : " ");
printf("%2s", isspace(ch) ? "Space" : " ");
printf("%3s", isprint(ch) ? "Printable" : " ");
printf("%2s ", isupper(ch) ? "Upper" : " ");
printf("%2s ", isxdigit(ch) ? "Hex" : " ");

    putchar('\n');
}
```

### Related Information

- “ctype.h” on page 24
- “isblank() — Test for Blank Character Classification” on page 706
- “iswalnum() to iswxdigit() — Test Wide Integer Value” on page 715
- “setlocale() — Set Locale” on page 1241
- “tolower() - toupper() — Convert Character Case” on page 1592

## isascii() — Test for 7-bit US-ASCII Character

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	POSIX(ON)

### Format

#### **\_XOPEN\_SOURCE**

```
#define _XOPEN_SOURCE
#include <ctype.h>

int isascii(int c);
```

#### **\_ALL\_SOURCE**

```
#define _ALL_SOURCE
#include <ctype.h>

int isascii(int c);
```

### General Description

#### **Special Behavior for \_XOPEN\_SOURCE**

The `isascii()` function tests whether `c` is a 7-bit US-ASCII character code. The `isascii()` function is defined on all integer values.

#### **Special Behavior for \_ALL\_SOURCE**

The `isascii()` function tests whether the character with EBCDIC encoding `c` in the current locale is a member of the set of POSIX Portable Characters and POSIX Control Characters shown below.

### Returned Value

#### **Special Behavior for \_XOPEN\_SOURCE**

The `isascii()` function returns nonzero if `c` is a 7-bit US-ASCII character code between 0 and hexadecimal 007F inclusive; otherwise it returns 0.

#### **Special Behavior for \_ALL\_SOURCE**

The `isascii()` function returns nonzero if `c` is the EBCDIC encoding in the current locale for a character in the set of POSIX Portable Characters and Control Characters; otherwise it returns 0.

Following is a list of the symbolic names, IBM-1047 EBCDIC code page encodings, and ISO-8859-1 ASCII encodings for the set of POSIX Portable Characters and POSIX Control Characters. Cases where EBCDIC character encodings vary across EBCDIC Country Extended Code Pages (CECPs) are noted.

Table 28 (Page 1 of 4). Characters for which isascii() returns nonzero

Character (Symbolic Name)	IBM-1047 Encoding (Hex)	ISO 8859-1 Encoding (Hex)
<NUL>	00	00
<SOH>	01	01
<STX>	02	02
<ETX>	03	03
<EOT>	37	04
<ENQ>	2D	05
<ACK>	2E	06
<BEL> <alert>	2F	07
<BS> <backspace>	16	08
<HT> <tab>	05	09
<NL> <newline>	15	0A
<VT> <vertical-tab>	0B	0B
<FF> <form-feed>	0C	0C
<CR> <carriage-return>	0D	0D
<SO>	0E	0E
<SI>	0F	0F
<DLE>	10	10
<DC1>	11	11
<DC2>	12	12
<DC3>	13	13
<DC4>	3C	14
<NAK>	3D	15
<SYN>	32	16
<ETB>	26	17
<CAN>	18	18
<EM>	19	19
<SUB>	3F	1A
<ESC>	27	1B
<IFS/IS4>	1C	1C
<IGS/IS3>	1D	1D
<IRS/IS2>	1E	1E
<IUS/ITB/IS1>	1F	1F
<space>	40	20
<exclamation-mark>	5A (cecp variant)	21
<quotation-mark>	7F	22
<number-sign>	7B (cecp variant)	23
<dollar-sign>	5B (cecp variant)	24
<percent-sign>	6C	25



Table 28 (Page 2 of 4). Characters for which *isascii()* returns nonzero

Character (Symbolic Name)	IBM-1047 Encoding (Hex)	ISO 8859-1 Encoding (Hex)
<ampersand>	50	26
<apostrophe>	7D	27
<left-parenthesis>	4D	28
<right-parenthesis>	5D	29
<asterisk>	5C	2A
<plus-sign>	4E	2B
<comma>	6B	2C
<hyphen>	60	2D
<period>	4B	2E
<slash>	61	2F
<zero>	F0	30
<one>	F1	31
<two>	F2	32
<three>	F3	33
<four>	F4	34
<five>	F5	35
<six>	F6	36
<seven>	F7	37
<eight>	F8	38
<nine>	F9	39
<colon>	7A	3A
<semicolon>	5E	3B
<less-than-sign>	4C	3C
<equals-sign>	7E	3D
<greater-than-sign>	6E	3E
<question-mark>	6F	3F
<commercial-at>	7C (cecp variant)	40
<A>	C1	41
<B>	C2	42
<C>	C3	43
<D>	C4	44
<E>	C5	45
<F>	C6	46
<G>	C7	47
<H>	C8	48
<I>	C9	49
<J>	D1	4A
<K>	D2	4B

Table 28 (Page 3 of 4). Characters for which isascii() returns nonzero

Character (Symbolic Name)	IBM-1047 Encoding (Hex)	ISO 8859-1 Encoding (Hex)
<L>	D3	4C
<M>	D4	4D
<N>	D5	4E
<O>	D6	4F
<P>	D7	50
<Q>	D8	51
<R>	D9	52
<S>	E2	53
<T>	E3	54
<U>	E4	55
<V>	E5	56
<W>	E6	57
<X>	E7	58
<Y>	E8	59
<Z>	E9	5A
<left-square-bracket>	AD (cecp variant)	5B
<backslash>	E0 (cecp variant)	5C
<right-square-bracket>	BD (cecp variant)	5D
<circumflex>	5F (cecp variant)	5E
<underscore>	6D	5F
<grave-accent>	79 (cecp variant)	60
<a>	81	61
<b>	82	62
<c>	83	63
<d>	84	64
<e>	85	65
<f>	86	66
<g>	87	67
<h>	88	68
<i>	89	69
<j>	91	6A
<k>	92	6B
<l>	93	6C
<m>	94	6D
<n>	95	6E
<o>	96	6F
<p>	97	70
<q>	98	71

Table 28 (Page 4 of 4). Characters for which isascii() returns nonzero

Character (Symbolic Name)	IBM-1047 Encoding (Hex)	ISO 8859-1 Encoding (Hex)
<r>	99	72
<s>	A2	73
<t>	A3	74
<u>	A4	75
<v>	A5	76
<w>	A6	77
<x>	A7	78
<y>	A8	79
<z>	A9	7A
<left-brace>	C0 (cecp variant)	7B
<vertical-line>	4F (cecp variant)	7C
<right-brace>	D0 (cecp variant)	7D
<tilde>	A1 (cecp variant)	7E
<DEL>	07	7F

### Related Information

- “toascii() — Translate Integer to a 7-bit ASCII Character” on page 1586

## isastream() — Test a File Descriptor

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stropts.h>
```

```
int isastream(int fildes);
```

### General Description

The *isastream()* function tests whether *fildes*, an open file descriptor, is associated with a STREAMS-based file.

### Returned Value

If successful, *isastream()* returns 1 if *fildes* refers to a STREAMS-based file and 0 if not. Otherwise, *isastream()* returns -1 and sets *errno* to indicate the error.

**Note:** OS/390 UNIX services do not supply any STREAMS devices or pseudodevices. It is impossible for *isastream()* to return 1 since there are no STREAMS-based file descriptors. It will return 0 unless *fildes* is not a valid open file descriptor, in which case it will return -1 with *errno* set to indicate the failure. See “*open()* — Open a File” on page 872 for more information.

EBADF     The *fildes* argument is not a valid open file descriptor.

### Related Information

- “*stropts.h*” on page 46

## isatty() — Test if Descriptor Represents a Terminal

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
int isatty(int fildes);
```

### General Description

Determines if a file descriptor, *fildes*, is associated with a terminal.

### Returned Value

isatty() returns 1 if the given file descriptor is a terminal, or 0 otherwise.

### Special Behavior for XPG4

isatty() returns 1 if the given file descriptor is a terminal, or 0 otherwise and sets `errno` to indicate the error.

`EBADF` The *fildes* argument is not a valid open file descriptor.

`ENOTTY` The *fildes* argument is not associated with a terminal.

### Example

#### CBC3BI03

```
/* CBC3BI03
   This example determines if a file descriptor is associated with a terminal.
*/
#define _POSIX_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

void check_fd(int fd) {
    printf("fd %d is ", fd);
    if (!isatty(fd))
        printf("NOT ");
    puts("a tty");
}

main() {
    int p[2], fd;
    char fn[]="temp.file";

    if (pipe(p) != 0)
        perror("pipe() error");
    else {
        if ((fd = creat(fn, S_IWUSR)) < 0)
            perror("creat() error");
```

```
    else {  
        check_fd(0);  
        check_fd(fileno(stderr));  
        check_fd(p[1]);  
        check_fd(fd);  
        close(fd);  
        unlink(fn);  
    }  
    close(p[0]);  
    close(p[1]);  
}
```

**Output**

```
fd 0 is a tty  
fd 2 is a tty  
fd 4 is NOT a tty  
fd 5 is NOT a tty
```

**Related Information**

- “unistd.h” on page 53
- “ttyname() — Get the Name of a Terminal” on page 1632

## \_\_isBFP() — Determine Application Floating-Point Format

### Standards

Standards / Extensions	C or C++	Dependencies
	both	OS/390 V2R6

### Format

```
#include <_Ieee754.h>
int __isBFP(void);
```

### General Description

The \_\_isBFP() function determines the application floating-point mode.

### Returned Value

\_\_isBFP() returns 1 if the floating-point mode of the caller is IEEE, and returns 0 if the floating-point mode of the caller is hexadecimal.

### Related Information

- “fp\_read\_rnd() — Determine Rounding Mode” on page 435
- “fp\_swap\_rnd() — Swap Rounding Mode” on page 446

## isblank() — Test for Blank Character Classification

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <ctype.h>
```

```
int isblank(int c);
```

### General Description

Tests whether the current LC\_CTYPE locale category assigns *c* the blank character attribute. The *tab* and *space* characters have the blank attribute in the POSIX locale (with name “POSIX” or “C”).

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

### Returned Value

isblank() returns a nonzero value if the current LC\_CTYPE locale category assigns *c* the blank character attribute. Otherwise, it returns the value 0.

### Example

```
/* This example tests if c is a blank type. */
#include <stdio.h>
#include <ctype.h>
#include <locale.h>

void check(char c) {
    if ((c != ' ') && (isprint(c)))
        printf(" %c is ", c);
    else
        printf("x%02x is ", c);
    if (!isblank(c))
        printf("not ");
    puts("a blank type character");
}

main() {
    printf("\nIn LC_CTYPE category of locale \ with name \"%s\":\n",
        setlocale(LC_CTYPE, NULL));
    check('a');
    check(' ');
    check(0x00);
    check('\n');
    check('\t');
}
```



**Output**

```
In LC_CTYPE category of locale with name “.....”;  
  a  is not a blank type character  
x40 is a blank type character  
x00 is not a blank type character  
x15 is not a blank type character  
x05 is a blank type character
```

**Related Information**

- “ctype.h” on page 24
- “iswalnum() to iswxdigit() — Test Wide Integer Value” on page 715
- “iswblank() — Test for Blank Character Classification” on page 718
- “setlocale() — Set Locale” on page 1241

## iscics() — Verify Whether CICS is Running

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <cics.h>
```

```
int iscics(void);
```

### General Description

Determines whether the program is running under CICS.

### Returned Value

The iscics() function returns a nonzero integer if your program is currently running under CICS, and returns the value 0 otherwise.

### Example

#### CBC3BI04

```
/* CBC3BI04
   This example tests to see if the program is running under CICS.
   If not, it calls a subroutine ABCPGM; otherwise, it uses a CICS EXEC
   statement to invoke ABCPGM.
*/
#ifdef __cplusplus
extern "OS" void ABCPGM(char *);
#else
#pragma linkage(ABCPGM, OS)
void ABCPGM(char *);
#endif

#include <stdio.h>
#include <cics.h>

int main(void)
{
    char mydata[123];

    if (iscics() == 0) {          /* not a CICS environment */
        ABCPGM(mydata);
    }
    else {                       /* this is a CICS environment */
        EXEC CICS LINK PROGRAM("ABCPGM ") COMMAREA(mydata);
    }
}
```

### Related Information

- “cics.h” on page 23

**iscntrl() — Test for Control Classification**

The information for this function is included in “isalnum() to isxdigit() — Test Integer Value” on page 694.

**isdigit() — Test for Hexadecimal-Digit Classification**

The information for this function is included in “isalnum() to isxdigit() — Test Integer Value” on page 694.

**isgraph() — Test for Graphic Classification**

The information for this function is included in “isalnum() to isxdigit() — Test Integer Value” on page 694.

**islower() — Test for Lowercase**

The information for this function is included in “isalnum() to isxdigit() — Test Integer Value” on page 694.

## ismccollet() — Identify a Multi-Character Collating Element

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <collate.h>
```

```
int ismccollet(collet_t c);
```

### General Description

Determines whether a character is a multicharacter collating element. A collating element is a glyph, usually a character, that has a value used to define its order in a collating sequence. A multicharacter collating element is a sequence of two or more characters that are to be collated as one entity.

### Returned Value

The ismccollet() function returns:

- 1            if collet\_t represents a multicharacter collating element
- 0            if collet\_t represents a single-character collating element
- 1           if collet\_t is out of range, or otherwise invalid

### Example CBC3BI05

```
/* CBC3BI05
   This example prints all of the collating elements in the collating
   sequence, by using the ismccollet() function to determine if the collating
   element is a multi-character collating element.
*/
#include <collate.h>
#include <locale.h>
#include <stdio.h>
#include <wchar.h>
#include <wctype.h>

main(int argc, char *argv[]) {
    collet_t e, *rp;
    int i;

    setlocale(LC_ALL, "");
    i = collorder(&rp);
    for (; i-- > 0; rp++) {
        if (ismccollet(*rp))
            printf('%s' ", colltostr(*rp));
        else if (iswprint(*rp))
            printf('%lc' ", *rp);
        else
            printf('%x' ", *rp);
    }
}
```

## Related Information

- “collate.h” on page 23
- “cclass() — Return Characters in a Character Class” on page 149
- “collequiv() — Return a List of Equivalent Collating Elements” on page 200
- “collorder() — Return List of Collating Elements” on page 202
- “collrange() — Calculate the Range List of Collating Elements” on page 204
- “colltostr() — Return a String for a Collating Element” on page 206
- “getmccoll() — Get Next Collating Element from String” on page 554
- “getwmccoll() — Get Next Collating Element from Wide String” on page 632
- “maxcoll() — Return Maximum Collating Element” on page 788
- “strtocoll() — Return Collating Element for String” on page 1454

## isnan() — Test for NaN

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <math.h>
```

```
int isnan( double x);
```

### General Description

The `isnan()` function tests whether `x` is NaN (not a number). On MVS, `isnan()` always returns 0.

`isnan()` is available as a macro. For better performance, the macro form is recommended over the functional form. To use the functional form, do one of the following:

- Do not include `math.h`.
- Specify `#undef isnan` after the inclusion of `math.h`.
- Enclose the call statement in parentheses.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

The `isnan()` function always returns 0.

There are no `errno` values defined for `isnan()`.

### Related Information

- “`math.h`” on page 35

## \_\_isPosixOn() — Test for Posix Runtime Option

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#include <unistd.h>
```

```
int __isPosixOn( void );
```

### General Description

The \_\_isPosixOn() function returns 1 if the kernel is active and the Posix runtime option is in effect for the calling process.

### Returned Value

The \_\_isPosixOn() function returns 1 if the kernel is active and the Posix runtime option is in effect for the calling process, and returns 0 otherwise.

No errors are defined for the \_\_isPosixOn() function.

### Related Information

- “unistd.h” on page 53
- “\_\_openMvsRel() — Get the OS/390 UNIX Release Number” on page 884

**isupper**

### **isprint() — Test for Printable Character Classification**

The information for this function is included in “isalnum() to isxdigit() — Test Integer Value” on page 694.

### **ispunct() — Test for Punctuation Classification**

The information for this function is included in “isalnum() to isxdigit() — Test Integer Value” on page 694.

### **isspace() — Test for Space Character Classification**

The information for this function is included in “isalnum() to isxdigit() — Test Integer Value” on page 694.

### **isupper() — Test for Uppercase Letter Classification**

The information for this function is included in “isalnum() to isxdigit() — Test Integer Value” on page 694.



## iswalnum() to iswxdigit() — Test Wide Integer Value

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wctype.h>
```

```
int iswalnum(wint_t wc);
int iswalpha(wint_t wc);
int iswcntrl(wint_t wc);
int iswdigit(wint_t wc);
int iswgraph(wint_t wc);
int iswlower(wint_t wc);
int iswprint(wint_t wc);
int iswpunct(wint_t wc);
int iswspace(wint_t wc);
int iswupper(wint_t wc);
int iswxdigit(wint_t wc);
```

### General Description

The functions listed above, which are all declared in `wctype.h`, test a given wide integer value. These functions are sensitive to locale. For locale descriptions, see “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*. Here are descriptions of each function in this group.

<code>iswalnum()</code>	Test for a wide alphanumeric character, as defined in the <code>alnum</code> locale source file and in the <code>alnum</code> class of the <code>LC_CTYPE</code> category of the current locale.
<code>iswalpha()</code>	Test for a wide alphabetic character, as defined in the <code>alpha</code> locale source file and in the <code>alpha</code> class of the <code>LC_CTYPE</code> category of the current locale.
<code>iswcntrl()</code>	Test for a wide control character, as defined in the <code>cntrl</code> locale source file and in the <code>cntrl</code> class of the <code>LC_CTYPE</code> category of the current locale.
<code>iswdigit()</code>	Test for a wide decimal-digit character: 0 through 9, as defined in the <code>digit</code> locale source file and in the <code>digit</code> class of the <code>LC_CTYPE</code> category of the current locale.
<code>iswgraph()</code>	Test for a wide printing character, not a space. as defined in the <code>graph</code> locale source file and in the <code>graph</code> class of the <code>LC_CTYPE</code> category of the current locale.
<code>iswlower()</code>	Test for a wide lowercase letter, as defined in the <code>lower</code> locale source file and in the <code>lower</code> class of the <code>LC_CTYPE</code> category of the current locale.
<code>iswprint()</code>	Test for any wide printing character, as defined in the <code>print</code> locale source file and in the <code>print</code> class of the <code>LC_CTYPE</code> category of the current locale.

iswpunct()	Test for a wide nonalphanumeric, nonspace character, as defined in the <code>punct</code> locale source file and in the <code>punct</code> class of the <code>LC_CTYPE</code> category of the current locale.
iswspace()	Test for a wide white-space character, as defined in the <code>space</code> locale source file and in the <code>space</code> class of the <code>LC_CTYPE</code> category of the current locale.
iswupper()	Test for a wide uppercase letter, as defined in the <code>upper</code> locale source file and in the <code>upper</code> class of the <code>LC_CTYPE</code> category of the current locale.
iswxdigit()	Test for a wide hexadecimal digit 0 through 9, a through f, or A through F, as defined in the <code>xdigit</code> locale source file and in the <code>xdigit</code> class of the <code>LC_CTYPE</code> category of the current locale.

The behavior of these wide-character function are affected by the `LC_CTYPE` category of the current locale. The space, uppercase, and lowercase characters can be redefined by their respective class of the `LC_CTYPE` in the current locale. If you change the category, undefined results can occur.

## Returned Value

These functions return a nonzero value if the wide integer satisfies the test value, or a 0 value if it does not. The value for `wc` must be representable as a wide unsigned character. `WEOF` is a valid input value.

## Example

### CBC3BI06

```

/* CBC3BI06
   This example tests for various wide integer values and prints a result.
*/
#include <stdio.h>
#include <wctype.h>

int main(void)
{
    wint_t wc;

    for (wc=0; wc <= 0xFF; wc++) {
        printf("%3d", wc);
        printf(" %#4x ", wc);
        printf("%3s", iswalnum(wc) ? "AN" : " ");
        printf("%2s", iswalpha(wc) ? "A" : " ");
        printf("%2s", iswcntrl(wc) ? "C" : " ");
        printf("%2s", iswdigit(wc) ? "D" : " ");
        printf("%2s", iswgraph(wc) ? "G" : " ");
        printf("%2s", iswlower(wc) ? "L" : " ");
        printf(" %c", iswprint(wc) ? wc : ' ');
        printf("%3s", iswpunct(wc) ? "PU" : " ");
        printf("%2s", iswspace(wc) ? "S" : " ");
        printf("%3s", iswprint(wc) ? "PR" : " ");
        printf("%2s", iswupper(wc) ? "U" : " ");
        printf("%2s", iswxdigit(wc) ? "X" : " ");

        putchar('\n');
    }
}

```

## Related Information

- “wctype.h” on page 56

## iswblank() — Test for Blank Character Classification

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <wctype.h>
```

```
int iswblank(wint_t wc);
```

### General Description

Tests for a wide blank character.

The space, uppercase, and lowercase characters can be redefined by their respective classes of the LC\_CTYPE in the current locale.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

### Returned Value

The function returns a nonzero value if the wide integer satisfies the test value, or a 0 value if it does not.

The value for `wc` must be representable as a wide unsigned char. WEOF is a valid input value.

The behavior of this wide-character function is affected by the LC\_CTYPE category of the current locale. If you change the category, undefined results can occur.

### Related Information

- “wctype.h” on page 56
- “isblank() — Test for Blank Character Classification” on page 706
- “iswalnum() to iswxdigit() — Test Wide Integer Value” on page 715

## **iswcntrl() — Test for Control Classification**

The information for this function is included in “iswalnum() to iswxdigit() — Test Wide Integer Value” on page 715.

## iswctype() — Test for Character Property

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wctype.h>
```

```
int iswctype(wint_t wc, wctype_t wc_prop);
```

### General Description

Determines whether the wide character *wc* has the property *wc\_prop*. If the value of *wc* is neither WEOF nor any value of the wide character that corresponds to a multibyte character, the behavior is undefined. If the value of *wc\_prop* is invalid (that is, not obtained by a previous call to `wctype()`, or *wc\_prop* has been invalidated by a subsequent call to `setlocale()` that has affected category `LC_CTYPE`), the behavior is undefined.

These eleven strings are reserved for the standard (basic) character classes: `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, and `xdigit`.

The functions are shown below with their equivalent `isw*()` function:

```
iswctype(wc, wctype("alnum")); - iswalnum(wc);
iswctype(wc, wctype("alpha")); - iswalpha(wc);
iswctype(wc, wctype("blank")); - iswblank(wc);
iswctype(wc, wctype("cntrl")); - iswcntrl(wc);
iswctype(wc, wctype("digit")); - iswdigit(wc);
iswctype(wc, wctype("graph")); - iswgraph(wc);
iswctype(wc, wctype("lower")); - iswlower(wc);
iswctype(wc, wctype("print")); - iswprint(wc);
iswctype(wc, wctype("punct")); - iswpunct(wc);
iswctype(wc, wctype("space")); - iswspace(wc);
iswctype(wc, wctype("upper")); - iswupper(wc);
iswctype(wc, wctype("xdigit")); - iswxdigit(wc);
```

### Returned Value

The `iswctype()` function returns nonzero (true) if the wide character *wc* has the property *wc\_prop*.

### Example CBC3BI07

```
/* CBC3BI07
   This example tests various wide characters for certain properties and
   prints the result.
*/
#include <stdio.h>
#include <wchar.h>
#include <wctype.h>

int main(void)
{
```

```

int wc;

for (wc=0; wc <= 0xFF; wc++) {
    printf("%3d", wc);
    printf(" %#4x ", wc);
    printf("%3s", iswctype(wc, wctype("alnum")) ? "AN" : " ");
    printf("%2s", iswctype(wc, wctype("alpha")) ? "A" : " ");
    printf("%2s", iswctype(wc, wctype("cntrl")) ? "C" : " ");
    printf("%2s", iswctype(wc, wctype("digit")) ? "D" : " ");
    printf("%2s", iswctype(wc, wctype("graph")) ? "G" : " ");
    printf("%2s", iswctype(wc, wctype("lower")) ? "L" : " ");
    printf(" %c", iswctype(wc, wctype("print")) ? wc : ' ');
    printf("%3s", iswctype(wc, wctype("punct")) ? "PU" : " ");
    printf("%2s", iswctype(wc, wctype("space")) ? "S" : " ");
    printf("%3s", iswctype(wc, wctype("print")) ? "PR" : " ");
    printf("%2s", iswctype(wc, wctype("upper")) ? "U" : " ");
    printf("%2s", iswctype(wc, wctype("xdigit")) ? "X" : " ");

    putchar('\n');
}

```

## Related Information

- “wctype.h” on page 56
- “wctype() — Obtain Handle for Character Property Classification” on page 1753

## **iswdigit() — Test for Hexadecimal-Digit Classification**

The information for this function is included in “iswalnum() to iswxdigit() — Test Wide Integer Value” on page 715.

## **iswgraph() — Test for Graphic Classification**

The information for this function is included in “iswalnum() to iswxdigit() — Test Wide Integer Value” on page 715.

## **iswlower() — Test for Lowercase**

The information for this function is included in “iswalnum() to iswxdigit() — Test Wide Integer Value” on page 715.

## **iswprint() — Test for Printable Character Classification**

The information for this function is included in “iswalnum() to iswxdigit() — Test Wide Integer Value” on page 715.

## **iswpunct() — Test for Punctuation Classification**

The information for this function is included in “iswalnum() to iswxdigit() — Test Wide Integer Value” on page 715.

## **iswspace() — Test for Space Character Classification**

The information for this function is included in “iswalnum() to iswxdigit() — Test Wide Integer Value” on page 715.

## **iswupper() — Test for Uppercase Letter Classification**

The information for this function is included in “iswalnum() to iswxdigit() — Test Wide Integer Value” on page 715.

## **iswxdigit() — Test for Hexadecimal-Digit Classification**

The information for this function is included in “iswalnum() to iswxdigit() — Test Wide Integer Value” on page 715.

## **isxdigit() — Test for Hexadecimal-Digit Classification**

The information for this function is included in “iswalnum() to iswxdigit() — Test Wide Integer Value” on page 715.



## JoinWorkUnit() — Join a WLM Work Unit

### Standards

Standards / Extensions	C or C++	Dependencies
	both	

### Format

```
#include <sys/_wlm.h>
int JoinWorkUnit(wlmetok_t *enclavetoken);
```

### General Description

The JoinWorkUnit function provides the ability for an application to join a WLM work unit.

*\*enclavetoken* Points to a work unit enclave token that was returned from a call to either CreateWorkUnit() or ContinueWorkUnit().

### Returned Value

Upon successful completion JoinWorkUnit() returns a zero. If the function is unsuccessful, -1 is returned and errno is set on to one of the following values:

- EFAULT** An argument of this function contained an address that was not accessible to the caller.
- EINVAL** An argument of this function contained an incorrect value.
- EMVSWLMERROR**  
The WLM join enclave failed. Use \_\_errno2() to obtain the WLM service reason code for the failure.
- EPERM** The calling thread's address space is not permitted to the BPX.WLMSEVER Facility class. The caller's address space must be permitted to the BPX.WLMSEVER Facility class if it is defined. If BPX.WLMSEVER is not defined, the calling process is not defined as a superuser (UID=0).
- EMVSSAF2ERR**  
An error occurred in the security product.

### Related Information

- “sys/\_wlm.h” on page 51
- “CheckSchEnv() — Check WLM Scheduling Environment” on page 172
- “ConnectServer() — Connect to WLM as a Server Manager” on page 219
- “ConnectWorkMgr() — Connect to WLM as a Work Manager” on page 221
- “ContinueWorkUnit() — Continue WLM Work Unit” on page 225
- “CreateWorkUnit() — Create WLM Work Unit” on page 236
- “DeleteWorkUnit() — Delete a WLM Work Unit” on page 272
- “DisconnectServer() — Disconnect from WLM Server” on page 276
- “LeaveWorkUnit() — Leave a WLM Work Unit” on page 742
- “QueryMetrics() — Query WLM System Information” on page 1070
- “QuerySchEnv() — Query WLM Scheduling Environment” on page 1072

## jrand48() — Pseudo-random Number Generator

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>
```

```
long int jrand48(unsigned short int x16v[3]);
```

### General Description

The drand48(), erand48(), jrand48(), lrand48(), mrand48() and nrand48() functions generate uniformly distributed pseudorandom numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The functions drand48() and erand48() return non-negative, double-precision, floating-point values, uniformly distributed over the interval [0.0,1.0).

The functions lrand48() and nrand48() return non-negative, long integers, uniformly distributed over the interval [0,2\*\*31).

The functions mrand48() and jrand48() return signed long integers, uniformly distributed over the interval [-2\*\*31,2\*\*31).

The jrand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(i), according to the linear congruential formula:

$$X(n+1) = (aX(n) + c) \bmod (2^{**}48) \quad n \geq 0$$

The jrand48() function uses storage provided by the argument array, x16v[3], to save the most recent 48-bit integer value in the sequence, X(i). The jrand48() function uses x16v[0] for the low order (rightmost) 16 bits, x16v[1] for the middle order 16 bits, and x16v[2] for the high order 16 bits of this value.

The initial values of a, and c are:

```
a   = 5deece66d (base 16)
c   = b          (base 16)
```

The values a and c, may be changed by calling the lcong48() function. The initial values of a and c are restored if either the seed48() or srand48() function is called.

### Special Behavior for OS/390 UNIX Services

You can make the jrand48() function and other functions in the drand48 family thread specific by setting the environment variable \_RAND48 to the value THREAD before calling any function in the drand48 family.

If you do not request thread specific behavior for the drand48 family, C/370 serializes access to the storage for X(n), a and c by functions in the drand48 family when they are called by a multithreaded application.

If thread specific behavior is requested and the `jrand48()` function is called from thread `t`, the `jrand48()` function generates the next 48-bit integer value in a sequence of 48-bit integer values,  $X(t,i)$ , for the thread according to the linear congruential formula:

$$X(t,n+1) = (a(t)X(t,n) + c(t)) \bmod (2^{**}48) \quad n \geq 0$$

The `jrand48()` function uses storage provided by the argument array, `x16v[3]`, to save the most recent 48-bit integer value in the sequence,  $X(t,i)$ . The `jrand48()` function uses `x16v[0]` for the low order (rightmost) 16 bits, `x16v[1]` for the middle order 16 bits, and `x16v[2]` for the high order 16 bits of this value.

The initial values of  $a(t)$  and  $c(t)$  on the thread `t` are:

```
a(t)   = 5deece66d (base 16)
c(t)   = b          (base 16)
```

The values  $a(t)$  and  $c(t)$  may be changed by calling the `lcong48()` function from the thread `t`. The initial values of  $a(t)$  and  $c(t)$  are restored if either the `seed48()` or `srand48()` function is called from the thread.

## Returned Value

The `jrand48()` function saves the generated 48-bit value,  $X(n+1)$ , in storage provided by the argument array, `x16v[3]`. The `jrand48()` function transforms the generated 48-bit value to a signed long integer value on the interval  $[-2^{**}31, 2^{**}31)$  and returns this transformed value.

## Special Behavior for OS/390 UNIX Services

If thread specific behavior is requested for the `drand48` family and the `jrand48()` function is called on thread `t`, the `jrand48()` function saves the generated 48-bit value,  $X(t,n+1)$ , in storage provided by the argument array, `x16v[3]`. The `jrand48()` function transforms the generated 48-bit value to a signed long integer value on the interval  $[-2^{**}31, 2^{**}31)$  and returns this transformed value.

## Related Information

- “`stdlib.h`” on page 45
- “`drand48()` — Pseudo-random Number Generator” on page 286
- “`erand48()` — Pseudo-random Number Generator” on page 314
- “`lcong48()` — Pseudo-random Number Initializer” on page 736
- “`lrand48()` — Pseudo-random Number Generator” on page 773
- “`mrnd48()` — Pseudo-random Number Generator” on page 843
- “`nrnd48()` — Pseudo-random Number Generator” on page 868
- “`seed48()` — Pseudo-random Number Initializer” on page 1156
- “`srand48()` — Pseudo-random Number Initializer” on page 1401

## j0() - j1() - jn() — Bessel Functions of the First Kind

### Standards

Standards / Extensions	C or C++	Dependencies
SAA XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double j0(double x);
```

```
double j1(double x);
```

```
double jn(int n, double x);
```

### Compiler Option

LANGVL(SAA), LANGVL(SAAL2), or LANGVL(EXTENDED)

### General Description

The j0(), j1(), and jn() functions are Bessel functions of the *first kind*, for orders 0, 1, and *n*, respectively. Bessel functions are solutions to certain types of differential equations. The argument *x* must be positive. The argument *n* should be greater than or equal to 0. If *n* is less than 0, there will be a negative exponent in the result.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

For j0(), j1(), y0(), or y1(), if the absolute value of *x* is too large, the function sets *errno* to ERANGE to indicate a value that is out of range, and returns a value of 0. The calculated value is returned otherwise.

### Example

#### CBC3BJ01

```
/* CBC3B01
   This example computes y to be the order 0 Bessel function of the first
   kind for x, and z to be the order 3 Bessel function of the second kind
   for x.
*/
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, y, z;
    x = 4.27;

    y = j0(x);      /* y = -0.3660 is the order 0 bessel */
                   /* function of the first kind for x */
    z = yn(3,x);    /* z = -0.0875 is the order 3 bessel */
}
```

```
/* function of the second kind for x */  
printf("x = %f\n y = %f\n z = %f\n", x, y, z);  
}
```

### Related Information

- “math.h” on page 35
- “erf() - erfc() — Calculate Error and Complementary Error Functions” on page 316
- “gamma() — Calculate Gamma Function” on page 497
- “y0() - y1() - yn() — Bessel Functions of the Second Kind” on page 1793

## kill() — Send a Signal to a Process

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	POSIX(ON) MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <signal.h>
```

```
int kill(pid_t pid, int sig);
```

### General Description

Sends a signal to a process or process group. A process has permission to send a signal if the real or effective user ID of the sender is the same as the real or effective user ID of the intended recipient. A process can also send signals if it has appropriate privileges. If `_POSIX_SAVED_IDS` is defined in the `unistd.h` header file, the saved set user ID of the intended recipient is checked instead of its effective user ID.

Regardless of user ID, a process can always send a `SIGCONT` signal to a process that is a member of the same session (same session ID) as the sender.

You can use either `signal()` or `sigaction()` to specify how a signal will be handled when `kill()` is invoked.

A process can use `kill()` to send a signal to itself. If the signal is not blocked or ignored, at least one pending unblocked signal is delivered to the sender before `kill()` returns. If there are no other pending unblocked signals, the delivered signal is *sig*.

*pid* can be used to specify these processes:

`pid_t pid;`

Specifies the processes that the caller wants to send a signal to:

- If *pid* is greater than 0, `kill()` sends its signal to the process whose ID is equal to *pid*.
- If *pid* is equal to 0, `kill()` sends its signal to all processes whose process group ID is equal to that of the sender, except for those that the sender does not have appropriate privileges to send a signal to.
- If *pid* is `-1`, `kill()` returns `-1`.
- **Special Behavior for XPG4.2:** If *pid* is `-1`, `kill()` sends the signal, *sig*, to all processes, except for those to which the sender does not have appropriate privileges to send a signal.
- If *pid* is less than `-1`, `kill()` sends its signal to all processes whose process group ID is equal to the absolute value of *pid*, except for those that the sender does not have appropriate privileges to send a signal to.

`int sig;`      The signal that should be sent to the processes specified by *pid*. (For a list of signals, see Table 32 on page 1295.) This must be 0 or one of the signals defined in the `signal.h` header file. If *sig* is 0, `kill()` performs error checking but does not send a signal. You can code *sig* as 0 to check whether the *pid* argument is valid.

This function is supported only in a POSIX program. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information. You can use it to pass `SIGIOERR`, but only with MVS 5.1 or higher.

## Returned Value

`kill()` returns 0 if it has permission to send *sig* to any of the processes specified by *pid*. If `kill()` fails to send a signal, it returns the value `-1` and sets `errno` to one of the following:

`EINVAL`    The value of *sig* is incorrect or is not the number of a supported signal.

`EPERM`     The caller does not have permission to send the signal to any process specified by *pid*.

`ESRCH`     There are no processes or process groups corresponding to *pid*.

## Example

### CBC3BK01

```
/* CBC3BK01 */
#define _POSIX_SOURCE
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <wait.h>

main() {
    sigset_t sigset;
    int    p[2], status;
    char   c='z';
    pid_t  pid;

    if (pipe(p) != 0)
        perror("pipe() error");
    else {
        if ((pid = fork()) == 0) {
            sigemptyset(&sigset);
            puts("child is letting parent know he's ready for signal");
            write(p[1], &c, 1);
            puts("child is waiting for signal");
            sigsuspend(&sigset);
            exit(0);
        }

        puts("parent is waiting for child to say he's ready for signal");
        read(p[0], &c, 1);
        puts("child has told parent he's ready for signal");

        kill(pid, SIGTERM);

        wait(&status);
        if (WIFSIGNALED(status))
            if (WTERMSIG(status) == SIGTERM)
                puts("child was ended with a SIGTERM");
        else
```

```

        printf("child was ended with a %d signal\n", WTERMSIG(status));
        else puts("child was not ended with a signal");

        close(p[0]);
        close(p[1]);
    }
}

```

**Output**

```

parent is waiting for child to say he's ready for signal
child is letting parent know he's ready for signal
child is waiting for signal
child has told parent he's ready for signal
child was ended with a SIGTERM

```

**Related Information**

- “signal.h” on page 41
- “unistd.h” on page 53
- “bsd\_signal() — BSD Version of signal()” on page 135
- “killpg() — Send a Signal to a Process Group” on page 731
- “pthread\_kill() — Send a Signal to a Thread” on page 984
- “raise() — Raise Signal” on page 1074
- “getpid() — Get the Process ID” on page 573
- “setsid() — Create Session, Set Process Group ID” on page 1268
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sighold() — Add a Signal to a Thread” on page 1322
- “sigignore() — Set Disposition to Ignore a Signal” on page 1323
- “signal() — Handle Interrupts” on page 1330
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “sigrelse() — Remove a Signal from a Thread” on page 1342
- “sigset() — Change a Signal Action and/or a Thread” on page 1343



## killpg() — Send a Signal to a Process Group

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON) MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>
```

```
int killpg(pid_t pgrp, int sig);
```

### General Description

The killpg() function sends a signal to a process group.

A process has permission to send a signal if the real or effective user ID of the sender is the same as the real or effective user ID of the intended recipient. A process can also send signals if it has appropriate privileges. If \_POSIX\_SAVED\_IDS is defined in the <unistd.h> include file, the saved set user ID of the intended recipient is checked instead of its effective user ID.

Regardless of user ID, a process can always send a SIGCONT signal to a process group that is a member of the same session (same session ID) as the sender.

*pid\_t pgrp*; Specifies the process group that the caller wants to send a signal to:

- If *pgrp* is greater than one, killpg() sends the signal, *sig*, to the process whose process group ID is equal to *pgrp* and which the sender has appropriate privileges to send a signal.
- If *pgrp* is equal to or less than one, killpg() returns a -1 and sets *errno* to **EINVAL**.

*int sig*; The signal that should be sent to the processes specified by *pid*. (For a list of signals, see Table 32 on page 1295.) This must be zero, or one of the signals defined in the <signal.h> include file. If *sig* is zero, killpg() performs error checking but doesn't really send a signal. You can code *sig* as zero to check whether the *pid* argument is valid.

This function is supported only in a POSIX program. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information.

### Returned Value

If successful, killpg() returns zero if it has permission to send *sig* to any of the processes in the process group ID specified by *pgrp*.

If unsuccessful, killpg() returns -1, and returns the error value in *errno*. The following are the possible values of *errno*:

- |               |  |
|---------------|--|
| <b>EINVAL</b> | The value of <i>sig</i> is incorrect or is not the number of a supported signal, or the value of <i>pgrp</i> is less than or equal to one. |
| <b>EPERM</b>  | The caller does not have permission to send the signal to any process in the process group ID specified by <i>pgrp</i> .                   |

ESRCH      There are no process groups corresponding to *pgid*.

**Related Information**

- “signal.h” on page 41
- “bsd\_signal() — BSD Version of signal()” on page 135
- “kill() — Send a Signal to a Process” on page 728
- “getpgid() — Get Process Group ID” on page 570
- “getpid() — Get the Process ID” on page 573
- “raise() — Raise Signal” on page 1074
- “setsid() — Create Session, Set Process Group ID” on page 1268
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sighold() — Add a Signal to a Thread” on page 1322
- “sigignore() — Set Disposition to Ignore a Signal” on page 1323
- “signal() — Handle Interrupts” on page 1330
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “sigrelse() — Remove a Signal from a Thread” on page 1342
- “sigset() — Change a Signal Action and/or a Thread” on page 1343

## labs() — Calculate Long Absolute Value

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
long int labs(long int n);
```

### General Description

Calculates the absolute value of its long integer argument *n*. The result is undefined when the argument is equal to LONG\_MIN, the smallest available long integer (-2 147 483 648). The value LONG\_MIN is defined in the limits.h header file.

### Returned Value

Returns the absolute value of the long integer argument *n*.

### Example

#### CBC3BL01

```
/* CBC3BL01
   This example computes y as the absolute value of the long
   integer -41567.
*/
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    long x, y;

    x = -41567L;
    y = labs(x);

    printf("The absolute value of %ld is %ld\n", x, y);
}
```

### Output

The absolute value of -41567 is 41567

### Related Information

- “stdlib.h” on page 45
- “abs() — Calculate Integer Absolute Value” on page 73
- “fabs() — Calculate Floating-Point Absolute Value” on page 338

## lchown() — Change Owner and Group of a File

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
int lchown(const char *path, uid_t owner, gid_t group);
```

### General Description

The `lchown()` function has the same effect as `chown()` except in the case where the named file is a symbolic link. In this case `lchown()` changes the ownership of the symbolic link file itself, while `chown()` changes the ownership of the file or directory to which the symbolic link refers.

### Returned Value

Upon successful completion, `lchown()` returns 0. Otherwise, it returns -1 and sets `errno` to indicate the error.

The `lchown()` function will fail if:

**EACCES** Search permission is denied on a component of the path prefix of *path*.

**EINVAL** The owner or group id is not a value supported by the implementation.

**ENAMETOOLONG**

The length of a pathname exceeds `PATH_MAX` or a pathname component is longer than `NAME_MAX`.

**ENOENT** A component of *path* does not name an existing file or *path* is an empty string.

**ENOTDIR** A component of the path prefix of *path* is not a directory.

**EOPNOTSUPP**

The *path* argument names a symbolic link and the implementation does not support setting the owner or group of a symbolic link.

**ELOOP** Too many symbolic links were encountered in resolving *path*

**EPERM** The effective user ID does not match the owner of the file and the process does not have appropriate privileges.

**EROFS** The file resides on a read-only file system.

The `lchown()` function may fail if:

**EIO** An I/O error occurred while reading or writing to the file system.

**EINTR** A signal was caught during execution of the function.

**ENAMETOOLONG**

Path name resolution of a symbolic link produced an intermediate result whose length exceeds `PATH_MAX`.

**Related Information**

- “`chown()` — Change the Owner or Group of a File or Directory” on page 177
- “`symlink()` — Create a Symbolic Link to a Path Name” on page 1479

## lcong48() — Pseudo-random Number Initializer

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>
```

```
void lcong48(unsigned short int param[7]);
```

### General Description

The drand48(), erand48(), jrand48(), lrand48(), mrand48() and nrand48() functions generate uniformly distributed pseudorandom numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The lcong48(), seed48(), and srand48() functions are initialization functions, one of which should be invoked before either the drand48(), lrand48() or mrand48() function is called.

The drand48(), lrand48() and mrand48() functions generate a sequence of 48-bit integer values,  $X(i)$ , according to the linear congruential formula:

$$X(n+1) = (aX(n) + c) \bmod (2^{**}48) \quad n \geq 0$$

The initial values of  $X$ ,  $a$ , and  $c$  are:

```
X(0) = 1
a    = 5deece66d (base 16)
c    = b          (base 16)
```

C/370 provides storage to save the most recent 48-bit integer value of the sequence,  $X(i)$ . This storage is shared by the drand48(), lrand48() and mrand48() functions. The lcong48() function is used to reinitialize the most recent 48-bit value in this storage. The lcong48() function replaces the low order (rightmost) 16 bits of this storage with *param*[0], the middle order 16 bits with *param*[1], and the high order 16 bits with *param*[2].

The values  $a$  and  $c$ , may also be changed by calling the lcong48() function. The lcong48() function replaces the low order (rightmost) 16 bits of  $a$  with *param*[3], the middle order 16 bits with *param*[4], and the high order 16 bits with *param*[5]. The lcong48() function replaces  $c$  with *param*[6].

### Special Behavior for OS/390 UNIX Services

You can make the lcong48() function and other functions in the drand48 family thread specific by setting the environment variable `_RAND48` to the value `THREAD` before calling any function in the drand48 family.

If you do not request thread specific behavior for the drand48 family, C/370 serializes access to the storage for  $X(n)$ ,  $a$  and  $c$  by functions in the drand48 family when they are called by a multithreaded application.

If thread specific behavior is requested, calls to the `drand48()`, `lrand48()` and `mrnd48()` functions from thread `t` generate a sequence of 48-bit integer values,  $X(t,i)$ , according to the linear congruential formula:

$$X(t,n+1) = (a(t)X(t,n) + c(t)) \bmod (2^{**}48) \quad n \geq 0$$

C/370 provides thread specific storage to save the most recent 48-bit integer value of the sequence,  $X(t,i)$ . When the `lcong48()` function is called from thread `t`, it reinitializes the most recent 48-bit value in this storage. The `lcong48()` function replaces the low order (rightmost) 16 bits of this storage with `param[0]`, the middle order 16 bits with `param[1]`, and the high order 16 bits with `param[2]`.

The `lcong48()` function may also be used to change values of `a(t)` and `c(t)` for the thread `t`. The `lcong48()` function replaces the low order (rightmost) 16 bits of `a(t)` with `param[3]`, the middle order 16 bits with `param[4]`, and the high order 16 bits with `param[5]`. The `lcong48()` function replaces `c(t)` with `param[6]`.

### Returned Value

After the `lcong48()` function has used values from the argument array, `param[7]`, to change the values of `a` and `c` and to reinitialized storage for the most recent 48-bit integer value in the sequence,  $X(i)$ , it returns.

### Special Behavior for OS/390 UNIX Services

If thread specific behavior is requested for the `drand48` family and the `lcong48()` function is called on thread `t`, it uses the argument array, `param[7]`, to change the values of `a(t)` and `c(t)` and to reinitialize storage for the most recent 48-bit integer value in the sequence,  $X(t,i)$ , for the thread. Then it returns.

### Related Information

- “`stdlib.h`” on page 45
- “`drand48()` — Pseudo-random Number Generator” on page 286
- “`erand48()` — Pseudo-random Number Generator” on page 314
- “`jrand48()` — Pseudo-random Number Generator” on page 724
- “`lrnd48()` — Pseudo-random Number Generator” on page 773
- “`mrnd48()` — Pseudo-random Number Generator” on page 843
- “`nrnd48()` — Pseudo-random Number Generator” on page 868
- “`seed48()` — Pseudo-random Number Initializer” on page 1156
- “`srand48()` — Pseudo-random Number Initializer” on page 1401

## ldexp() — Multiply by a Power of Two

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double ldexp(double x, int exp);
```

### General Description

Calculates the value of  $x \cdot 2^{\text{exp}}$ .

### Returned Value

Returns the calculated value. Otherwise, if the correct calculated value is outside the range of representable values, plus or minus HUGE\_VAL is returned, according to the sign of the value. The value of the macro ERANGE is stored in errno to indicate that the result was out of range.

### Special Behavior for XPG4.2

ERANGE The result underflowed. ldexp() returns 0.0.

### Example

#### CBC3BL02

```
/* CBC3BL02
   This example computes y = 1.5 * (2**5).
*/
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, y;
    int p;

    x = 1.5;
    p = 5;
    y = ldexp(x,p);

    printf("%lf times 2 to the power of %d is %Lf\n", x, p, y);
}
```

### Output

```
1.500000 times 2 to the power of 5 is 48.000000
```



**Related Information**

- “math.h” on page 35
- “frexp() — Extract Mantissa and Exponent of the Floating-Point Value” on page 462
- “modf() — Extract Fractional and Integral Parts of Floating-Point Value” on page 837

## ldiv() — Compute Quotient and Remainder of Integral Division

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
ldiv_t ldiv(long int numerator, long int denominator);
```

### General Description

Calculates the quotient and remainder of the division of *numerator* by *denominator*.

### Returned Value

Returns a structure of type `ldiv_t`, containing both the quotient `long int quot` and the remainder `long int rem`. If the value cannot be represented, the returned value is undefined. If *denominator* is 0, a divide by 0 exception is raised.

### Example

#### CBC3BL03

```
/* CBC3BL03
   This example uses the ldiv() function to calculate the quotients and
   remainders for a set of two dividends and two divisors.
*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    long int num[2] = {45,-45};
    long int den[2] = {7,-7};
    ldiv_t ans; /* ldiv_t is a struct type containing two long int:
                  'quot' stores quotient; 'rem' stores remainder */
    short i,j;

    printf("Results of long division:\n");
    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
        {
            ans = ldiv(num[i], den[j]);
            printf("Dividend: %6ld Divisor: %6ld", num[i], den[j]);
            printf(" Quotient: %6ld Remainder: %6ld\n", ans.quot, ans.rem);
        }
}
```

### Output

```
Results of long division:
Dividend: 45 Divisor: 7 Quotient: 6 Remainder: 3
Dividend: 45 Divisor: -7 Quotient: -6 Remainder: 3
Dividend: -45 Divisor: 7 Quotient: -6 Remainder: -3
Dividend: -45 Divisor: -7 Quotient: 6 Remainder: -3
```

**Related Information**

- “stdlib.h” on page 45
- “div() — Calculate Quotient and Remainder” on page 277

## LeaveWorkUnit() — Leave a WLM Work Unit

### Standards

Standards / Extensions	C or C++	Dependencies
	both	

### Format

```
#include <sys/_wlm.h>
int LeaveWorkUnit(wlmetok_t *enclavetoken);
```

### General Description

The `LeaveWorkUnit()` function provides the ability for an application to leave a WLM work unit.

*\*enclavetoken* Points to a work unit enclave token that was returned from a call to `CreateWorkUnit()` or `ContinueWorkUnit()`.

### Returned Value

Upon successful completion `LeaveWorkUnit()` return a zero. If the function is unsuccessful, -1 is returned and `errno` is set on to one of the following values:

**EFAULT** An argument of this function contained an address that was not accessible to the caller.

**EINVAL** An argument of this function contained an incorrect value.

**EMVSWLMERROR**  
The WLM leave enclave failed. Use `__errno2()` to obtain the WLM service reason code for the failure.

**EPERM** The calling thread's address space is not permitted to the BPX.WLMSEVER Facility class. The caller's address space must be permitted to the BPX.WLMSEVER Facility class if it is defined. If BPX.WLMSEVER is not defined, the calling process is not defined as a superuser (UID=0).

**EMVSSAF2ERR**  
An error occurred in the security product.

### Related Information

- “`sys/_wlm.h`” on page 51
- “`CheckSchEnv()` — Check WLM Scheduling Environment” on page 172
- “`ConnectServer()` — Connect to WLM as a Server Manager” on page 219
- “`ConnectWorkMgr()` — Connect to WLM as a Work Manager” on page 221
- “`ContinueWorkUnit()` — Continue WLM Work Unit” on page 225
- “`CreateWorkUnit()` — Create WLM Work Unit” on page 236
- “`DeleteWorkUnit()` — Delete a WLM Work Unit” on page 272
- “`DisconnectServer()` — Disconnect from WLM Server” on page 276
- “`JoinWorkUnit()` — Join a WLM Work Unit” on page 723
- “`QueryMetrics()` — Query WLM System Information” on page 1070
- “`QuerySchEnv()` — Query WLM Scheduling Environment” on page 1072

## lfind() — Linear Search Routine

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <search.h>
```

```
void *lfind(const void *key, const void *base, size_t *nelp,
           size_t width, int (*compar)(const void *, const void *));
```

### General Description

The `lfind()` function is the same as `lsearch()` except that if the entry is not found, it is not added to the table. Instead, a null pointer is returned.

### Special Behavior for C++

Because C and C++ linkage conventions are incompatible, `lfind()` cannot receive a C++ function pointer as the comparator argument. If you attempt to pass a C++ function pointer to `lfind()`, the compiler will flag it as an error. You can pass a C or C++ function to `lfind()` by declaring it as `extern "C"`.

### Returned Value

If the searched for entry is found, the `lfind()` function returns a pointer to it, otherwise it returns a null pointer.

No errors are defined.

### Related Information

- “`search.h`” on page 40
- “`lsearch()` — Linear Search and Update” on page 775
- “`bsearch()` — Search Arrays” on page 137
- “`hsearch()` — Search Hash Tables” on page 647
- “`tsearch()` — Binary Tree Search” on page 1617

lgamma() — Log Gamma Function

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

Format

```
#define _XOPEN_SOURCE
#include <math.h>
```

```
double lgamma(double x);
extern int signgam;
int *__signgam(void);
```

General Description

The lgamma() function computes the  $\log_e |\Gamma(x)|$  where  $\Gamma(x)$  is defined as  $\int_0^\infty e^{-t} t^{(x-1)} dt$ .

The sign of  $\Gamma(x)$  is returned in the external integer *signgam*. The argument *x* may not be a non-positive integer.

In a multithreaded process, each thread has its own instance of the *signgam* variable. Threads access their instances of the variable by calling the `__signgam()` function. See “`__signgam()` — Return signgam Reference” on page 1335. The `math.h` header (see “`math.h`” on page 35) redefines the string “*signgam*” to an invocation of the `__signham` function. The actual *signgam* external variable is used to store the *signgam* value for the IPT.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

Returned Value

If it succeeds, lgamma() returns the above function of its argument.

lgamma() will fail under the following conditions:

- If the result overflows, the function will return **HUGE\_VAL** and set `errno` to `ERANGE`.
- If *x* is a non-positive integer, lgamma() returns **HUGE\_VAL** and sets `errno` to `EDOM`.

Example

```

/*
   This example uses lgamma() to calculate  $\ln(|G(x)|)$ , where  $x = 42$ .
*/
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x=42, g_at_x;

    g_at_x = exp(lgamma(x));          /* g_at_x = 3.345253e+49 */
    printf ("The value of G(%4.2f) is %7.2e\n", x, g_at_x);
}

```

### Output

The value of G(42.00) is 3.35e+49

### Related Information

- “math.h” on page 35
- “exp() — Calculate Exponential Function” on page 334
- “isnan() — Test for NaN” on page 712
- “\_\_signgam() — Return signgam Reference” on page 1335

## \_\_librel() — Query Release Level

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <stdlib.h>
```

```
int __librel(void);
```

### General Description

Provides the release level of the OS/390 C/C++ library. To use this function, you must compile with LANGLVL(EXTENDED).

### Returned Value

Returns the OS/390 C/C++ Specific Library release level that your OS/390 C or OS/390 C++ program is using. The value is meant to be printed in a hexadecimal format. The first byte of the value returned contains the product and version, second byte the release, and the third and fourth bytes contain the modification level. For C programs running under the C/370 Specific Library (the common library version), the product designation is 0.

The following diagram shows the formats of the 32-bit int returned by the two versions of \_\_librel().

The C/370 V2R2 version of \_\_librel() returns 0x02020000

```

| | | |
| | | |--- Mod level
| | | |--- Release
| | | |--- Version

```

In this case, the high-order 8 bits are used to return the version number.



The LE 1.5 version of `__librel()` returns 0x11050000

```

| | | |
| | | |--- Mod level
| |---- Release level
|---- Version level
----- Product 1 (LE)

```

The OS/390 R2 version of `__librel()` returns 0x21020000

```

| | | |
| | | |--- Mod level
| |---- Release level
|---- Version level
----- Product 2 (OS/390)

```

The OS/390 R3 version of `__librel()` returns 0x21030000

```

| | | |
| | | |--- Mod level
| |---- Release level
|---- Version level
----- Product 2 (OS/390)

```

The OS/390 R4 version of `__librel()` returns 0x22040000

```

| | | |
| | | |--- Mod level
| |---- Release level
|---- Version level
----- Product 2 (OS/390)

```

The OS/390 R6 version of `__librel()` returns 0x22060000

```

| | | |
| | | |--- Mod level
| |---- Release level
|---- Version level
----- Product 2 (OS/390)

```

The OS/390 R7 version of `__librel()` returns 0x22070000

```

| | | |
| | | |--- Mod level
| |---- Release level
|---- Version level
----- Product 2 (OS/390)

```

In these cases, these 8 bits are divided into two fields. The first 4 bits contain the product number and the second 4 bits contain the version number.

### Example CBC3BL04

```

/* CBC3BL04
   This example demonstrates what is returned when the function __librel()
   is called when using the Version 1 Release 5 Modification Level 0, of
   Language Environment for MVS.
*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    printf("The current release of the library is: %X\n",__librel());
}

```

### Output

The current release of the library is: 11050000

### **Related Information**

- “stdlib.h” on page 45

## link() — Create a Link to a File

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
int link(const char *oldfile, const char *newname);
```

### General Description

Provides an alternative path name for the existing file, so that the file can be accessed by either the old or the new name. `link()` creates a link from the path name *newname* to an existing file, with the path name *oldfile*. The link can be stored in the same directory as the original file or in a completely different one.

Links are allowed to files only, not to directories.

This is a hard link, which ensures the existence of a file even after its original name has been removed.

If `link()` successfully creates the link, it increments the *link count* of the file. The link count tells how many links there are to the file. At the same time, `link()` updates the change time of the file, and the change time and modification time of the directory that contains *newname* (that is, the directory that holds the link). If `link()` fails, the link count is not incremented.

If *oldfile* names a symbolic link, `link()` creates a link that refers to the file that results from resolving the path name contained in the symbolic link. If *newname* names a symbolic link, `link()` fails and sets `errno` to `EEXIST`.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

If successful, `link()` returns the value 0. If unsuccessful, it returns the value `-1` and it sets `errno` to one of the following:

- EACCES** The process did not have appropriate permissions to create the link. Possible reasons include no search permission on a path name component of *oldfile* or *newname*, no write permission on the directory intended to contain the link, or no permission to access *oldfile*.
- EEXIST** Either *newname* refers to a symbolic link, or a file or directory with the name *newname* already exists.
- EINVAL** Either *oldfile* or *newname* is incorrect, because it contains a null.

- ELOOP** A loop exists in symbolic links. This error is issued if the number of symbolic links encountered during resolution of *oldfile* or *newname* is greater than `POSIX_SYMLINK_MAX`.
- EMLINK** *oldfile* already has its maximum number of links. The maximum number of links to a file is given by `LINK_MAX`, which you can determine by using `pathconf()` or `fpathconf()`.
- ENAMETOOLONG** *oldfile* or *newname* is longer than `PATH_MAX`, or a component of one of the path names is longer than `NAME_MAX` while `_POSIX_NO_TRUNC` is in effect. For symbolic links, the length of the pathname string substituted for a symbolic link in *oldfile* or *newname* exceeds `PATH_MAX`. The `PATH_MAX` and `NAME_MAX` values can be determined using `pathconf()`.
- ENOENT** A path name component of *oldfile* or *newname* does not exist, or *oldfile* itself does not exist, or one of the two arguments is an empty string.
- ENOSPC** The directory intended to contain the link cannot be extended to contain another entry.
- ENOTDIR** A path name component of one of the arguments is not a directory.
- EPERM** *oldfile* is the name of a directory, and links to directories are not supported.
- EROFS** Creating the link would require writing on a read-only file system.
- EXDEV** *oldfile* and *newname* are on different file systems.

### Example

```
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

main() {
    char fn[]="link.example.file";
    char ln[]="link.example.link";
    int fd;

    if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        close(fd);
        if (link(fn, ln) != 0) {
            perror("link() error");
            unlink(fn);
        }
        else {
            unlink(fn);
            unlink(ln);
        }
    }
}
```

**Related Information**

- “unistd.h” on page 53
- “rename() — Rename File” on page 1136
- “symlink() — Create a Symbolic Link to a Path Name” on page 1479
- “unlink() — Remove a Directory Entry” on page 1660

## listen() — Prepare the Server for Incoming Client Requests

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int listen(int socket, int backlog);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/socket.h>

int listen(int socket, int backlog);
```

### General Description

The `listen()` call applies only to stream sockets. It indicates a readiness to accept client connection requests, and creates a connection request queue of length *backlog* to queue incoming connection requests. Once full, additional connection requests are rejected.

#### Parameter Description

*socket*     The socket descriptor.

*backlog*    Defines the maximum length for the queue of pending connections.

The `listen()` call indicates a readiness to accept client connection requests. It transforms an active socket into a passive socket. Once called, *socket* can never be used as an active socket to initiate connection requests. Calling `listen()` is the third of four steps that a server performs to accept a connection. It is called after allocating a stream socket with `socket()`, and after binding a name to *socket* with `bind()`. It must be called before calling `accept()`.

If the backlog is less than 0, *backlog* is set to 0. If the backlog is greater than SOMAXCONN, as defined in **sys/socket.h**, *backlog* is set to SOMAXCONN.

For AF\_UNIX sockets, this value is variable and can be set in the application. For AF\_INET sockets, the value cannot exceed the maximum number of connections allowed by the installed TCP/IP.

#### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

## Returned Value

The value 0 indicates success; the value -1 indicates an error. The value of the error code indicates the specific error.

### Error Code      Description

EBADF            The *socket* parameter is not a valid socket descriptor.

### EDESTADDRREQ

The socket is not bound to a local address, and the protocol does not support listening on an unbound socket.

EINVAL           An invalid argument was supplied. The socket is not named (a `bind()` has not been done), or the socket is ready to accept connections (a `listen()` has already been done). The socket is already connected.

ENOBUFS        Insufficient system resources are available to complete the call.

ENOTSOCK       The descriptor is for a file, not for a socket.

### EOPNOTSUPP

The *socket* parameter is not a socket descriptor that supports the `listen()` call.

## Related Information

- “`accept()` — Accept a New Connection on a Socket” on page 75
- “`bind()` — Bind a Name to a Socket” on page 128
- “`connect()` — Connect a Socket” on page 214
- “`socket()` — Create a Socket” on page 1371

## localdtconv() — Date/Time Formatting Convention Inquiry

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <locale.h>
```

```
struct dtconv *localdtconv(void);
```

### General Description

Determines the date/time format information of the current locale.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

### Returned Value

Returns the address of the *dtconv* structure:

```
struct dtconv {
    char *abbrev_month_names[12]; /* Abbreviated month names */
    char *month_names[12];       /* full month names */
    char *abbrev_day_names[7];   /* Abbreviated day names */
    char *day_names[7];          /* full day names */
    char *date_time_format;      /* date and time format */
    char *date_format;           /* date format */
    char *time_format;           /* time format */
    char *am_string;             /* AM string */
    char *pm_string;             /* PM string */
    char *time_format_ampm;      /* long date format */
};
```

The *dtconv* structure can be overwritten by subsequent calls to *localdtconv()* and *setlocale()* with *LC\_ALL* or *LC\_TIME*.

### Related Information

- “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*
- “locale.h” on page 33
- “time.h” on page 51
- “asctime() — Convert Time to Character String” on page 106
- “ctime() — Convert Time to a Character String” on page 250
- “gmtime() — Convert Time to Broken-Down UTC Time” on page 640
- “localeconv() — Query Numeric Conventions” on page 756
- “localtime() — Convert Time and Correct for Local Time” on page 758
- “mktime() — Convert Local Time” on page 828



- “setlocale() — Set Locale” on page 1241
- “strftime() — Convert to Formatted Time” on page 1432
- “time() — Determine Current Time” on page 1570
- “tzset() — Set the Time Zone” on page 1637

## localeconv() — Query Numeric Conventions

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <locale.h>
```

```
struct lconv *localeconv(void);
```

### General Description

Sets the components of a structure having type `struct lconv` to values appropriate for the current locale. The structure may be overwritten by another call to `localeconv()` or by calling `setlocale()` and passing `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC`.

For a list of the elements in the `lconv` structure, see Table 8 on page 33.

Pointers to strings with a value of "" indicate that the value is not available in the C locale or is of 0 length. `char` types with a value of `UCHAR_MAX` indicate that the value is not available in the current locale.

The *grouping* and *non\_grouping* elements can have the following values:

#### CHAR\_MAX

No further grouping is to be performed.

0 The previous element is to be repeatedly used for the remainder of the digits.

other The integer value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

The `n_sign_posn` and `p_sign_posn` elements can have the following values:

#### Value      Meaning

0 The quantity and `currency_symbol` are enclosed in parentheses.

1 The sign precedes the quantity and `currency_symbol`.

2 The sign follows the quantity and `currency_symbol`.

3 The sign precedes the `currency_symbol`.

4 The sign follows the `currency_symbol`.

5 Use `debit_sign` or `credit_sign` for `p_sign_posn` or `n_sign_posn`.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

## Returned Value

Returns a pointer to the structure.

## Example CBC3BL06

```
/* CBC3BL06
   This example prints out the default decimal point for your locale and
   then the decimal point for the Fr_CA locale.
*/
#include <stdio.h>
#include <locale.h>

int main(void)
{
    char * string;
    struct lconv * mylocale;
    mylocale = localeconv();
    /* Display default decimal point */
    printf( "Default decimal point is a %s\n",
            mylocale->decimal_point );

    if (NULL != (string = setlocale(LC_ALL, "Fr_CA.IBM-1047" )))
    {
        mylocale = localeconv();
        /* A comma is set to be the decimal point
           when the locale is Fr_CA.IBM-1047 */
        printf( "French-speaking Canadian decimal point is a %s\n",
                mylocale->decimal_point );
    }
    else {
        printf("setlocale(LC_ALL, Fr_CA.IBM-1047) returned <NULL>\n");
    }
    return 0;
}
```

## Output

Default decimal point is a .  
 French-speaking Canadian decimal point is a ,

## Related Information

- “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*
- “locale.h” on page 33
- “localdtconv() — Date/Time Formatting Convention Inquiry” on page 754
- “localtime() — Convert Time and Correct for Local Time” on page 758
- “setlocale() — Set Locale” on page 1241

## localtime() — Convert Time and Correct for Local Time

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <time.h>
```

```
struct tm *localtime(const time_t *timeval);
```

### General Description

Converts the calendar time pointed to by *timeval* to a broken-down time expressed in local time. Calendar time is usually obtained by a call to the `time()` function.

### Returned Value

Returns a pointer to a *tm* structure containing the broken-down time, expressed as a local time, and corresponding to the calendar time pointed to by *timeval*. If the calendar time cannot be converted, `localtime()` returns a null pointer. See “`gmtime()` — Convert Time to Broken-Down UTC Time” on page 640 for a description of the fields of the *tm* structure.

### Notes:

- This function is sensitive to time zone information which is provided by:
  - The TZ environmental variable when POSIX(ON) and TZ is correctly defined, or by the \_TZ environmental variable when POSIX(OFF) and \_TZ is correctly defined.
  - The LC\_TOD category of the current locale if POSIX(OFF) or TZ is not defined.

The time zone external variables `tzname`, `timezone`, and `daylight` declarations remain feature test protected in `time.h`.

- The `ctime()`, `localtime()`, and `mktime()` functions now return coordinated universal time (UTC) unless customized locale information is made available, which includes setting the `timezone_name` variable.
- In POSIX you can supply the necessary information by using environment variables.
- In non-POSIX applications, you can supply customized locale information by setting time zone and daylight information in `LC_TOD`.
- By customizing the locale, you allow the time functions to preserve both time and date, correctly adjusting for daylight time on a given date.
- The `gmtime()` and `localtime()` functions may use a common, statically allocated structure for the conversion. Each call to one of these functions will alter the result of the previous call.
- Calendar time returned by the `time()` function begins at the epoch, which was at 00:00:00 coordinated universal time (UTC), January 1, 1970.

- The `localtime()` function converts calendar time (that is, seconds elapsed since the epoch) to broken-down time, expressed as local time, using time zone information.

Such information is provided as follows:

- For a POSIX program, time zone information is provided by the TZ environment variable or the current LC\_TOD locale category. The `localtime()` function calls the `tzset()` function to parse the TZ environment variable. If `tzset()` cannot find the TZ environment variable or cannot parse it, `tzset()` obtains time zone information for the `localtime()` function from the current LC\_TOD locale category. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information about using POSIX support.
- For all other C and C++ applications, time zone information is provided by the current LC\_TOD locale category.

See “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide* for a description of LC\_TOD, which is a nonstandard, OS/390 C/C++ proprietary locale category.

### Example CBC3BL07

```
/* CBC3BL07
   This example queries the system clock and displays the local time.
*/
#include <time.h>
#include <stdio.h>

int main(void)
{
    struct tm *newtime;
    time_t ltime;

    time(&ltime);
    newtime = localtime(&ltime);
    printf("The date and time is %s", asctime(newtime));
}
```

### Output

This output would occur if the local time is 3:00 p.m. June 16, 1995):

The date and time is Fri Jun 16 15:00:00 1995

### Related Information

- “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*
- “locale.h” on page 33
- “time.h” on page 51
- “asctime() — Convert Time to Character String” on page 106
- “ctime() — Convert Time to a Character String” on page 250
- “gmtime() — Convert Time to Broken-Down UTC Time” on page 640
- “localdtconv() — Date/Time Formatting Convention Inquiry” on page 754
- “mktime() — Convert Local Time” on page 828
- “strftime() — Convert to Formatted Time” on page 1432
- “time() — Determine Current Time” on page 1570
- “tzset() — Set the Time Zone” on page 1637

## lockf() — Record Locking on Files

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
```

```
int lockf(int filedes, int function, off_t size);
```

### General Description

The `lockf()` function allows sections of a file to be locked with advisory-mode locks. Calls to `lockf()` from other processes which attempt to lock the locked file section will either return an error value or block until the section becomes unlocked. All the locks for a process are removed when the process terminates. Record locking with `lockf()` is supported for regular files.

The *filedes* argument is an open file descriptor. The file descriptor must have been opened with a write-only permission (`O_WRONLY`) or with read/write permission (`O_RDWR`) to establish a lock with this function.

The *function* argument is a control value which specifies the action to be taken. The permissible values for *function* are defined in `<unistd.h>` as follows:

Function	Description
-----	-----
<code>F_ULOCK</code>	unlock locked sections
<code>F_LOCK</code>	lock a section for exclusive use
<code>F_TLOCK</code>	test and lock a section for exclusive use
<code>F_TEST</code>	test a section for locks by other processes

`F_TEST` detects if a lock by another process is present on the specified section; `F_LOCK` and `F_TLOCK` both lock a section of a file if the section is available; `F_ULOCK` removes locks from a section of the file.

The *size* argument is the number of contiguous bytes to be locked or unlocked. The section to be locked or unlocked starts at the current offset in the file and extends forward for a positive size or backward for a negative size (the preceding bytes up to but not including the current offset). If *size* is 0, the section from the current offset through the largest possible file offset is locked (that is, from the current offset through the present or any future end-of-file). An area need not be allocated to the file to be locked because locks may exist past the end-of-file.

The sections locked with `F_LOCK` or `F_TLOCK` may, in whole or in part, contain or be contained by a previously locked section for the same process. When this occurs, or if adjacent locked sections would occur, the sections are combined into a single locked section. If the request would cause the number of locks to exceed a system imposed limit, the request will fail.

`F_LOCK` and `F_TLOCK` requests differ only by the action taken if the section is not available. `F_LOCK` blocks the calling process until the section is available.

F\_TLOCK makes the function fail if the section is already locked by another process.

File locks are released on first close by the locking process of any file descriptor for the file.

F\_ULOCK requests may release (wholly or in part) one or more locked sections controlled by the process. Locked sections will be unlocked starting at the current file offset through *size* bytes or to the end of file if *size* is (off\_t)0. When all of a locked section is not released (that is, when the beginning or end of the area to be unlocked falls within a locked section), the remaining portions of that section are still locked by the process. Releasing the center portion of a locked section will cause the remaining locked beginning and end portions to become two separate locked sections. If the request would cause the number of locks in the system to exceed a system imposed limit, the request will fail.

A potential for deadlock occurs if a process controlling a locked section is blocked by accessing another process' locked section. If the system detects that a deadlock would occur, lockf() will fail with an EDEADLK error.

Locks obtained by lockf() are controlled by the same facility controlling locks obtained by fcntl().

The interaction between fcntl() and lockf() locks is unspecified.

Blocking on a section is interrupted by any signal.

## Returned Value

If successful, lockf() returns 0. Otherwise, it returns -1, sets errno to one of the following to indicate an error, and existing locks are not changed:

The lockf() function will fail if:

**EBADF**     The *filedes* argument is not a valid open file descriptor; or *function* is F\_LOCK or F\_TLOCK and *filedes* is not a valid file descriptor open for writing.

**EACCES or EAGAIN**     The *function* argument is F\_TLOCK or F\_TEST and the section is already locked by another process

**EDEADLK**     The *function* argument is F\_LOCK and a deadlock is detected.

**EINTR**     A signal was caught during execution of the function.

## Related Information

- “unistd.h” on page 53

## log() — Calculate Natural Logarithm

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double log(double x);
```

### General Description

Calculates the natural logarithm (base e) of  $x$ , for  $x$  greater than 0.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

Returns the computed value. If  $x$  is negative, it sets `errno` to `EDOM` and returns the value `—HUGE_VAL`. If  $x$  is 0.0, `log()` returns the value `—HUGE_VAL` and sets `errno` to `ERANGE`. If the correct value would cause an underflow, 0 is returned and the value of the macro `ERANGE` is stored in `errno`.

### Example

#### CBC3BL08

```
/* CBC3BL08
   This example calculates the natural logarithm of 1000.0.
*/
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x = 1000.0, y;

    y = log(x);

    printf("The natural logarithm of %lf is %lf\n", x, y);
}
```

### Output

The natural logarithm of 1000.000000 is 6.907755

### Related Information

- “math.h” on page 35
- “exp() — Calculate Exponential Function” on page 334
- “log10() — Calculate Base 10 Logarithm” on page 767
- “pow() — Raise to Power” on page 916



## logb() — Unbiased Exponent

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>
```

```
double logb(double x);
```

### General Description

The logb() function returns the radix-16 exponent of its argument *x*, which is the integral part of

$\log_{\text{base } r} (\text{abs}(x))$

as a double float.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

If it succeeds, logb() returns the radix-16 exponent of *x*.

logb() will fail under the following condition: If *x* is equal to 0.0, logb() will return -HUGE\_VAL and set errno to EDOM.

### Related Information

- “math.h” on page 35
- “ilogb() — Integer Unbiased Exponent” on page 659

## \_\_login() — Create a New Security Environment for Process

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R6

### Format

```
#define _OPEN_SYS

#include <unistd.h>

int __login(int    function_code,
            int    identity_type,
            int    identity_length,
            void   *identity,
            int    pass_length,
            char   *pass,
            int    certificate_length,
            char   *certificate,
            int    option_flags);
```

### General Description

The \_\_login() function provides a way for a process to change its identity so as to be different than the address space identity and create a new security environment for the process. Once changed the process should not revert back to a previous identity and security environment. The following rules apply:

- Any single threaded process can issue a \_\_login to change its security environment.
- If the process is in a multiproc/multiuser environment and there is no task level security associated with the process, then the new security environment will be associated with the process.
- If the process is in a multiproc/multiuser environment and there is task level security associated with the process, then the old security environment will be replaced by the new security environment.

The function has the following parameters:

Parameter	Description
<i>function_code</i>	Specifies the function. Specify __LOGIN_CREATE, as defined in the unistd.h header file, to create a process level security environment for the caller's process.
<i>identity_type</i>	Specifies the format of the the user identity being provided in <i>*identity</i> . Specify __LOGIN_USERID, as defined in the unistd.h header file. The userid identity is in the format of a 1 to 8 character userid and is passed as input.
<i>identity_length</i>	Specifies the length of the <i>identity</i> as defined by <i>identity_type</i> .
<i>*identity</i>	Specifies the user identity as defined by <i>identity_type</i> .
<i>pass_length</i>	Specifies the length of the password defined by <i>pass</i> .

<i>*pass</i>	Specifies a user password or pass ticket.
<i>certificate_length</i>	Is not used presently and must be set to zero.
<i>certificate</i>	Is not used presently and must point to void.
<i>option_flags</i>	Specifies options used to tailor request. Must be set to 0.

### Usage Notes:

1. The intent of the \_\_login() service is to provide a way for a process to change its identity so as to be different than the address space identity. The process should either terminate or select a new userid, but should not try to revert back to the original identity. The user could issue the \_\_login() again with the original user identity, but the task would retain its own security environment and not share the the security environment at the address space level.
2. A security manager supporting multiproc/multiuser environment must be installed and operational.

### Returned Value

The \_\_login function returns 0 if the request is successful. If unsuccessful, \_\_login() returns -1 and sets errno to one of the following:

<b>errno</b>	<b>Meaning</b>
EINVAL	A parameter is invalid.
EPERM	The operation was not permitted. Calling process may not be authorized in BPX.DAEMON facility class. The function is not supported in an address space where a load was done from an uncontrolled library. A required password was not specified.
ESRCH	The USERID cannot become an OMVS process. The userid provided is not defined to the security manager or doesn't have an OMVS segment defined.
EMVSSAF2ERR	An error occurred in the security product. The userid has been revoked or is unable to use the application.
EMVSERR	An MVS environmental error or internal occurred.
ENOSYS	The function is not implemented or installed.
EACCES	Permission is denied.
EMVSEXPIRE	The password for the specified resource has expired.

### Related Information

- “unistd.h” on page 53

## log1p() — Natural Log of $x + 1$

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>
```

```
double log1p(double x);
```

### General Description

The `log1p()` function computes  $\log_e(1.0 + x)$ . The value of  $x$  must be greater than  $-1.0$ .

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

If it succeeds, `log1p()` returns the value of the above function of  $x$ .

`log1p()` will fail under the following conditions:

- If  $x$  is less than  $-1.0$ , `log1p()` will return `-HUGE_VAL` and set `errno` to `EDOM`.
- If  $x$  is equal to  $-1.0$ , `log1p()` will return `-HUGE_VAL` and set `errno` to `ERANGE`.

### Related Information

- “`math.h`” on page 35
- “`log()` — Calculate Natural Logarithm” on page 762

## log10() — Calculate Base 10 Logarithm

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double log10(double x);
```

### General Description

Calculates the base 10 logarithm of the positive value of *x*.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

Returns the computed value. If *x* is negative, it sets *errno* to *EDOM* and returns the value `—HUGE_VAL`. If *x* is 0, `log10()` returns the value `—HUGE_VAL`, and sets *errno* to *ERANGE*. If the correct value would cause an underflow, 0 is returned and the value of the macro *ERANGE* is stored in *errno*.

### Example

#### CBC3BL09

```
/* CBC3BL09
   This example calculates the base 10 logarithm of 1000.0.
*/
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x = 1000.0, y;

    y = log10(x);

    printf("The base 10 logarithm of %lf is %lf\n", x, y);
}
```

### Output

The base 10 logarithm of 1000.000000 is 3.000000

### Related Information

- “math.h” on page 35
- “exp() — Calculate Exponential Function” on page 334
- “log() — Calculate Natural Logarithm” on page 762
- “pow() — Raise to Power” on page 916

## longjmp() — Restore Stack Environment

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <setjmp.h>
```

```
void longjmp(jmp_buf env, int value);
```

### General Description

Restores a stack environment previously saved in *env* by `setjmp()`. The `setjmp()` and `longjmp()` functions provide a way to perform a nonlocal goto. They are often used in signal handlers.

A call to `setjmp()` causes the current stack environment to be saved in *env*. A subsequent call to `longjmp()` restores the saved environment, and returns control to a point in the program corresponding to the `setjmp()` call. Execution resumes as if the `setjmp()` call had just returned the given *value* of the *value* argument. All variables that are accessible to the function that receives control contain the values they had when `longjmp()` was called. The values of register variables are unpredictable. Non-volatile *auto* variables that are changed between calls to `setjmp()` and `longjmp()` are also unpredictable.

**Note:** Ensure that the function that calls `setjmp()` does not return before you call the corresponding `longjmp()` function. Calling `longjmp()` after the function calling `setjmp()` returns causes unpredictable program behavior.

The *value* argument passed to `longjmp()` must be nonzero. If you give a 0 argument for *value*, `longjmp()` substitutes a 1 in its place.

### Special Behavior for POSIX

In a POSIX program, the signal mask is *not* saved. Thus, to save and restore a stack environment that includes the current signal mask, use `sigsetjmp()` and `siglongjmp()` instead of `setjmp()` and `longjmp()`. The `sigsetjmp()`—`siglongjmp()` pair, the `setjmp()`—`longjmp()` pair, the `_setjmp()`—`_longjmp()` pair, and the `getcontext()`—`setcontext()` pair cannot be intermixed. A stack environment saved by `setjmp()` can be restored only by `longjmp()`. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information about using POSIX support.

### Special Behavior for C++

If `setjmp()` and `longjmp()` are used to transfer control in a OS/390 C++ program, the behavior in terms of the destruction of automatic objects is undefined. This applies to both OS/390 C++ and C/C++ OS/390 ILC modules. The use of `setjmp()` and `longjmp()` in conjunction with `try()`, `catch()`, and `throw()` is also undefined.

### Special Behavior for XPG4.2

In a program that was compiled with the feature test macro, `_XOPEN_SOURCE_EXTENDED`, defined, another pair of functions, `_setjmp()`—`_longjmp()` are available. These functions are, on this implementation, functionally identical to `setjmp()`—`longjmp()`. Therefore it is possible, but not recommended, to intermix the `setjmp()`—`longjmp()` pair with the `_setjmp()`—`_longjmp()` pair.

## Returned Value

The `longjmp()` function does not use the normal function call and return mechanisms; it has no returned value.

## Example

This example provides for saving the stack environment at this statement:  
`if(setjmp(mark) != 0) ...`

When the system first performs the `if` statement, it saves the environment in *mark* and sets the condition to FALSE because `setjmp()` returns a 0 when it saves the environment. The program prints the message: `setjmp has been called`

The subsequent call to function *p* tests for a local error condition, which can cause it to perform the `longjmp()` function. Then, control returns to the original `setjmp()` function using the environment saved in *mark*. This time the condition is TRUE because -1 is the returned value from the `longjmp()` function. The example then performs the statements in the block and prints: `longjmp has been called`

It then performs the *recover* function and leaves the program.

```
/* Illustration of longjmp(). */
#include <stdio.h>
#include <setjmp.h>

jmp_buf mark;

void p(void);
void recover(void);

int main(void)
{
    if (setjmp(mark) != 0)
    {
        printf("longjmp has been called\n");
        recover();
        exit(1);
    }
    printf("setjmp has been called\n");
    :
    p();
    :
}

void p(void)
{
    int error = 0;
    :
    error = 9;
    :
    if (error != 0)
        longjmp(mark, -1);
    :
}
```

```
}  
  
void recover(void)  
{  
    :  
}
```

### Related Information

- “setjmp.h” on page 40
- “getcontext() — Get User Context” on page 505
- “\_longjmp() — Non-Local Goto” on page 771
- “setcontext() — Restore User Context” on page 1210
- “\_setjmp() — Set Jump Point for a Non-Local Goto” on page 1237
- “setjmp() — Preserve Stack Environment” on page 1234
- “siglongjmp() — Restore the Stack Environment and Signal Mask” on page 1327
- “sigsetjmp() — Save Stack Environment and Signal Mask” on page 1346
- “swapcontext() — Save and Restore User Context” on page 1472



## `_longjmp()` — Non-Local Goto

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <setjmp.h>
```

```
void _longjmp(jmp_buf env, int value);
```

### General Description

The `_longjmp()` function restores a stack environment previously saved in *env* by `_setjmp()`. The `_setjmp()` and `_longjmp()` functions provide a way to perform a non-local *goto*. They are often used in signal handlers.

A call to `_setjmp()` causes the current stack environment to be saved in *env*.

A subsequent call to `_longjmp()` restores the saved environment and returns control to a point in the program corresponding to the `_setjmp()` call. Execution resumes as if the `_setjmp()` call had just returned the given *value* of the *value* argument. All variables that are accessible to the function that receives control contain the values they had when `_longjmp()` was called. The values of register variables are unpredictable. Nonvolatile *auto* variables that are changed between calls to `_setjmp()` and `_longjmp()` are also unpredictable.

The X/Open standard states that `_longjmp()` and `_setjmp()` are functionally identical to `longjmp()` and `setjmp()`, respectively, with the addition restriction that `_longjmp()` and `_setjmp()` do not manipulate the signal mask. However, on this implementation `longjmp()` and `setjmp()` do not manipulate the signal mask. So on this implementation `_longjmp()` and `_setjmp()` are literally identical to `longjmp()` and `setjmp()`, respectively.

To save and restore a stack environment, including the current signal mask, use `sigsetjmp()` and `siglongjmp()` instead of `_setjmp()` and `_longjmp()`, or `setjmp()` and `longjmp()`.

The `_setjmp()`—`_longjmp()` pair, the `setjmp()`—`longjmp()` pair, the `sigsetjmp()`—`siglongjmp()` pair, and the `getcontext()`—`setcontext()` pair cannot be intermixed. A stack environment saved by `_setjmp()` can be restored only by `_longjmp()`.

### Notes:

1. However, on this implementation, since the `_setjmp()`—`_longjmp()` pair are functionally identical to the `setjmp()`—`longjmp()` pair it is possible to intermix them, but it is not recommended.
2. Ensure that the function that calls `_setjmp()` does not return before you call the corresponding `_longjmp()` function. Calling `_longjmp()` after the function calling `_setjmp()` returns causes unpredictable program behavior.

The *value* argument passed to `_longjmp()` must be nonzero. If you give a zero argument for *value*, `_longjmp()` substitutes a 1 in its place.

*env*            An address for a `jmp_buf` structure

*value*          The return value from `_setjmp()`

### **Special Behavior for C++**

If `_setjmp()` and `_longjmp()` used to transfer control in a OS/390 C++ program, the behavior in terms of the destruction of automatic objects is undefined. This applies both to C OS/390 and C/C++ OS/390 ILC modules. The use of `_setjmp()` and `_longjmp()` in conjunction with `try()`, `catch()`, and `throw()` is also undefined.

### **Returned Value**

The `_longjmp()` function does not use the normal function call and return mechanisms; it has no returned value. When `_longjmp()` completes, program execution continues as if the corresponding invocation of `_setjmp()` had just returned the value specified by *value*.

### **Related Information**

- “`setjmp.h`” on page 40
- “`getcontext()` — Get User Context” on page 505
- “`longjmp()` — Restore Stack Environment” on page 768
- “`setcontext()` — Restore User Context” on page 1210
- “`_setjmp()` — Set Jump Point for a Non-Local Goto” on page 1237
- “`setjmp()` — Preserve Stack Environment” on page 1234
- “`siglongjmp()` — Restore the Stack Environment and Signal Mask” on page 1327
- “`sigsetjmp()` — Save Stack Environment and Signal Mask” on page 1346
- “`swapcontext()` — Save and Restore User Context” on page 1472

## lrand48() — Pseudo-random Number Generator

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>
```

```
long int lrand48(void);
```

### General Description

The drand48(), erand48(), jrand48(), lrand48(), mrand48() and nrand48() functions generate uniformly distributed pseudorandom numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The functions drand48() and erand48() return non-negative, double-precision, floating-point values, uniformly distributed over the interval [0.0,1.0).

The functions lrand48() and nrand48() return non-negative, long integers, uniformly distributed over the interval [0,2\*\*31).

The functions mrand48() and jrand48() return signed long integers, uniformly distributed over the interval [-2\*\*31,2\*\*31).

The lrand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(i), according to the linear congruential formula:

$$X(n+1) = (aX(n) + c) \bmod (2^{**}48) \quad n \geq 0$$

The initial values of X, a, and c are:

```
X(0) = 1
a    = 5deece66d (base 16)
c    = b          (base 16)
```

C/370 provides storage to save the most recent 48-bit integer value of the sequence, X(i). This storage is shared by the drand48(), lrand48() and mrand48() functions. The value, X(n), in this storage may be reinitialized by calling the lcong48(), seed48() or srand48() function. Likewise, the values of a and c, may be changed by calling the lcong48() function. Thereafter, whenever the seed48() or srand48() function is called to change X(n), the initial values of a and c are also reestablished.

### Special Behavior for OS/390 UNIX Services

You can make the lrand48() function and other functions in the drand48 family thread specific by setting the environment variable \_RAND48 to the value THREAD before calling any function in the drand48 family.

If you do not request thread specific behavior for the drand48 family, C/370 serializes access to the storage for X(n), a and c by functions in the drand48 family when they are called by a multithreaded application.

If thread specific behavior is requested, and the `lrand48()` function is called from thread `t`, the `lrand48()` function generates the next 48-bit integer value in a sequence of 48-bit integer values,  $X(t,i)$ , for the thread `t`. The sequence of values for a thread is generated according to the linear congruential formula:

$$X(t,n+1) = (a(t)X(t,n) + c(t)) \bmod (2^{**}48) \quad n \geq 0$$

The initial values of  $X(t)$ ,  $a(t)$  and  $c(t)$  for the thread `t` are:

```
X(t,0) = 1
a(t)   = 5deece66d (base 16)
c(t)   = b         (base 16)
```

C/370 provides storage which is specific to the thread `t` to save the most recent 48-bit integer value of the sequence,  $X(t,i)$ , generated by the `drand48()`, `lrand48()` or `mrnd48()` function. The value,  $X(t,n)$ , in this storage may be reinitialized by calling the `lcong48()`, `seed48()` or `srand48()` function from the thread `t`. Likewise, the values of  $a(t)$  and  $c(t)$  for thread `t` may be changed by calling the `lcong48()` function from the thread. Thereafter, whenever the `seed48()` or `srand48()` function is called from the thread `t` to change  $X(t,n)$ , the initial values of  $a(t)$  and  $c(t)$  are also reestablished.

## Returned Value

The `lrand48()` function transforms the generated 48-bit value,  $X(n+1)$ , to a non-negative, long integer value on the interval  $[0, 2^{**}31)$  and returns this transformed value.

## Special Behavior for OS/390 UNIX Services

If thread specific behavior is requested for the `drand48` family and the `lrand48()` function is called on thread `t`, the `lrand48()` function transforms the generated 48-bit value,  $X(t,n+1)$ , to a non-negative, long integer value on the interval  $[0, 2^{**}31)$  and returns this transformed value.

## Related Information

- “`stdlib.h`” on page 45
- “`drand48()` — Pseudo-random Number Generator” on page 286
- “`erand48()` — Pseudo-random Number Generator” on page 314
- “`jrand48()` — Pseudo-random Number Generator” on page 724
- “`lcong48()` — Pseudo-random Number Initializer” on page 736
- “`mrnd48()` — Pseudo-random Number Generator” on page 843
- “`nrnd48()` — Pseudo-random Number Generator” on page 868
- “`seed48()` — Pseudo-random Number Initializer” on page 1156
- “`srand48()` — Pseudo-random Number Initializer” on page 1401

## lsearch() — Linear Search and Update

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <search.h>

void *lsearch(const void *key, void *base, size_t
*nelp, size_t width,
               int (*compar)(const void *, const void *));
```

### General Description

The `lsearch()` function is a linear search routine. It returns a pointer into a table indicating where an entry may be found. If the entry does not occur, it is added at the end of the table. The *key* argument points to the entry to be sought in the table. The *base* argument points to the first element in the table. The *width* argument is the size of an element in bytes. The *nelp* argument points to an integer containing the current number of elements in the table. The integer to which *nelp* points is incremented if the entry is added to the table. The *compar* argument points to a comparison function which the user must supply ( `strcmp()`, for example). It is called with two arguments that point to the elements being compared. The function must return 0 if the elements are equal and nonzero otherwise.

### Special Behavior for C++

Because C and C++ linkage conventions are incompatible, `lsearch()` cannot receive a C++ function pointer as the comparator argument. If you attempt to pass a C++ function pointer to `lsearch()`, the compiler will flag it as an error. You can pass a C or C++ function to `lsearch()` by declaring it as `extern "C"`.

### Returned Value

If the searched for entry is found, the `lsearch()` function returns a pointer to it, otherwise it returns a pointer to the newly added element. A null pointer is returned in case of error.

No errors are defined.

### Related Information

- “`search.h`” on page 40
- “`lfind()` — Linear Search Routine” on page 743
- “`bsearch()` — Search Arrays” on page 137
- “`hsearch()` — Search Hash Tables” on page 647
- “`tsearch()` — Binary Tree Search” on page 1617

## lseek() — Change the Offset of a File

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
off_t lseek(int filides, off_t offset, int pos);
```

### General Description

Changes the current file offset to a new position in an HFS file. The new position is the given byte *offset* from the position specified by *pos*. After you have used `lseek()` to seek to a new location, the next I/O operation on the file begins at that location.

`lseek()` lets you specify new file offsets past the current end of the file. If data is written at such a point, read operations in the gap between this data and the old end of the file will return bytes containing zeros. (In other words, the gap is assumed to be filled with zeros.)

Seeking past the end of a file, however, does not automatically extend the length of the file. There must be a write operation before the file is actually extended.

### Special Behavior for POSIX C

For character special files, `lseek()` sets the file offset to the specified value. OS/390 UNIX services ignore the file offset value during the read/write processing to character special files.

<code>int <i>filides</i>;</code>	The file whose current file offset you want to change.
<code>off_t <i>offset</i>;</code>	The amount (positive or negative) the byte offset is to be changed. The sign indicates whether the offset is to be moved forward (positive) or backward (negative).
<code>int <i>pos</i>;</code>	One of the following symbols (defined in the <code>unistd.h</code> header file):
	SEEK_SET The start of the file
	SEEK_CUR The current file offset in the file
	SEEK_END The end of the file

See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information about using POSIX support.

## Returned Value

If successful, `lseek()` returns the new file offset, measured in bytes from the beginning of the file. If unsuccessful, it returns the value `-1`.

If `lseek()` fails, it sets `errno` to one of the following:

- `EBADF`    *fil*des is not a valid open file descriptor.
- `EINVAL`    *pos* contained something other than one of the three options, or the combination of the *pos* values would have placed the file offset before the beginning of the file.
- `ESPIPE`    *fil*des is associated with a pipe or FIFO special file.

## Example

This fragment positions a file (that has at least 10 bytes) to an offset of 10 bytes before the end of the file.

```
lseek(fil
```

## Related Information

- “`unistd.h`” on page 53
- “`fseek()` — Change File Position” on page 474
- “`fsetpos()` — Set File Position” on page 477
- “`creat()` — Create a New File or Rewrite an Existing One” on page 233
- “`dup()` — Duplicate an Open File Descriptor” on page 288
- “`fcntl()` — Control Open File Descriptors” on page 350
- “`open()` — Open a File” on page 872
- “`read()` — Read From a File or Socket” on page 1080
- “`sigaction()` — Examine or Change a Signal Action” on page 1295
- “`write()` — Write Data on a File or Socket” on page 1780

## lstat() — Get Status of File or Symbolic Link

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX1_SOURCE 2
#include <sys/stat.h>
```

```
int lstat(const char *pathname, struct stat *buf);
```

### General Description

Gets status information about a specified file and places it in the area of memory pointed to by the *buf* argument. You do not need permissions on the file itself, but you must have search permission on all directory components of the *pathname*.

If the named file is a symbolic link, lstat() returns information about the symbolic link itself.

The information is returned in the following stat structure, defined in the sys/stat.h header file.

Table 29. Elements of the stat Structure

Structure	Description
mode_t st_mode	A bit string indicating the permissions and privileges of the file. Symbols are defined in the sys/stat.h header file to refer to bits in a mode_t value; these symbols are listed in “chmod() — Change the Mode of a File or Directory” on page 174.
ino_t st_ino	The serial number of the file.
dev_t st_dev	The numeric ID of the device containing the file.
nlink_t st_nlink	The number of links to the file.
uid_t st_uid	The numeric user ID of the file's owner.
gid_t st_gid	The numeric group ID of the file's group.
off_t st_size	For regular files, the file's size in bytes. For symbolic links, the length of the pathname contained therein not counting the trailing null. For other kinds of files, the value of this field is unspecified.
time_t st_atime	The most recent time the file was accessed.
time_t st_ctime	The most recent time the status of the file was changed.
time_t st_mtime	The most recent time the contents of the file were changed.

Values for time\_t are given in terms of seconds that have elapsed since epoch.

If the named file is a symbolic link, lstat() updates the time-related fields before putting information in the stat structure.



You can examine properties of a `mode_t` value from the `st_mode` field by using a collection of macros defined in the `sys/modes.h` header file. If *mode* is a `mode_t` value, and *genvalue* is an unsigned `int` value from the `stat` structure, then:

`S_ISBLK(mode)`

Is nonzero for block special files.

`S_ISCHR(mode)`

Is nonzero for character special files.

`S_ISDIR(mode)`

Is nonzero for directories.

`S_ISEXTL(mode,genvalue)`

Is nonzero for external links. (External links are introduced in MVS 5.1; they are not supported in MVS 4.3 or previous releases, where they return 0).

`S_ISFIFO(mode)`

Is nonzero for pipes and FIFO special files.

`S_ISLNK(mode)`

Is nonzero for symbolic links.

`S_ISREG(mode)`

Is nonzero for regular files.

`S_ISSOCK(mode)`

Is nonzero for sockets. (Sockets are introduced in MVS 5.1; they are not supported in MVS 4.3 or previous releases, where they return 0).

If `Istat()` successfully determines all this information, it stores it in the area indicated by the *buf* argument.

## Returned Value

If successful, `Istat()` returns the value 0. If unsuccessful, it returns the value -1.

If `Istat()` fails, it sets `errno` to one of the following:

- |              |   |
|--------------|---|
| EACCES       | The process does not have search permission on some component of the <i>pathname</i> prefix.  |
| EINVAL       | <i>buf</i> contains a null.   |
| ELOOP        | A loop exists in symbolic links. This error is issued if the number of symbolic links encountered during resolution of the <i>pathname</i> argument is greater than <code>POSIX_SYMLINK_MAX</code> .  |
| ENAMETOOLONG | <i>pathname</i> is longer than <code>PATH_MAX</code> characters or some component of <i>pathname</i> is longer than <code>NAME_MAX</code> characters while <code>_POSIX_NO_TRUNC</code> is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds <code>PATH_MAX</code> . The <code>PATH_MAX</code> and <code>NAME_MAX</code> values can be determined through <code>pathconf()</code> . |
| ENOENT       | There is no file named <i>pathname</i> , or <i>pathname</i> is an empty string.   |
| ENOTDIR      | A component of the <i>pathname</i> prefix is not a directory.   |

## Special Behavior for XPG4.2

The following new errno:

**EIO** An I/O error occurred while reading from the file system.

**EOVERFLOW**

The file size exceeded the storage reserved for `st_size` in the **stat** structure.

### Example CBC3BL12

```
/* CBC3BL12
   This example provides status information for a file.
*/
#define _OPEN_SYS
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

main() {
    char fn[]="temp.file", ln[]="temp.link";
    struct stat info;
    int fd;

    if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        close(fd);
        if (link(fn, ln) != 0)
            perror("link() error");
        else {
            if (lstat(ln, &info) != 0)
                perror("lstat() error");
            else {
                puts("lstat() returned:");
                printf(" inode:  %d\n", (int) info.st_ino);
                printf(" dev id:  %d\n", (int) info.st_dev);
                printf(" mode:   %08x\n", info.st_mode);
                printf(" links:  %d\n", info.st_nlink);
                printf(" uid:   %d\n", (int) info.st_uid);
                printf(" gid:   %d\n", (int) info.st_gid);
                printf("created:  %s", ctime(&info.st_createtime));
            }
            unlink(ln);
        }
        unlink(fn);
    }
}
```

### Output

```
lstat() returned:
inode:  3022
dev id:  1
mode:   03000080
links:  2
uid:    25
gid:    500
created:  Fri Jun 16 15:00:00 1995
```

## Related Information

- “sys/stat.h” on page 48
- “sys/types.h” on page 49
- “chmod() — Change the Mode of a File or Directory” on page 174
- “chown() — Change the Owner or Group of a File or Directory” on page 177
- “creat() — Create a New File or Rewrite an Existing One” on page 233
- “dup() — Duplicate an Open File Descriptor” on page 288
- “extlink\_np() — Create an External Symbolic Link” on page 336
- “fcntl() — Control Open File Descriptors” on page 350
- “fstat() — Get Status Information about a File” on page 479
- “link() — Create a Link to a File” on page 749
- “mkdir() — Make a Directory” on page 817
- “mkfifo() — Make a FIFO Special File” on page 820
- “open() — Open a File” on page 872
- “pipe() — Create an Unnamed Pipe” on page 907
- “read() — Read From a File or Socket” on page 1080
- “readlink() — Read the Value of a Symbolic Link” on page 1091
- “remove() — Delete File” on page 1133
- “stat() — Get File Information” on page 1404
- “symlink() — Create a Symbolic Link to a Path Name” on page 1479
- “unlink() — Remove a Directory Entry” on page 1660
- “utime() — Set File Access and Modification Times” on page 1664
- “write() — Write Data on a File or Socket” on page 1780

## l64a() — Convert Long to Base-64 String Representation

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>
```

```
char *l64a (long value);
```

### General Description

The l64a() function converts a long integer into its corresponding base-64 character representation. In this notation, long integers are represented by up to 6 characters, each character representing a digit in base-64 notation. The following characters are used to represent digits:

Character	Digit represented
.	0
/	1
0-9	2-11
A-Z	12-37
a-z	38-63

### Returned Value

The l64a() function returns a pointer to the base-64 representation of *value*. If *value* is zero, l64a() returns a pointer to a null string.

The l64a() function returns a pointer to a static buffer, which will be overwritten by subsequent calls. Buffers are allocated on a per-thread basis.

There are no errno values defined for l64a().

### Related Information

- “stdlib.h” on page 45
- “strtoul() — Convert String to Unsigned Integer” on page 1462
- “a64l() — Convert Base-64 String Representation to Long Integ” on page 124

## makecontext() — Modify User Context

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON)

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ucontext.h>
```

```
void makecontext(ucontext_t *ucp, void (*func) (), int argc, ...);
```

### General Description

The `makecontext()` function modifies the context specified by `ucp`, which has been initialized using `getcontext()`. When this context is resumed using `setcontext()` or `swapcontext()`, program execution continues by calling `func()`, passing it the arguments that follow `argc` in the `makecontext()` call.

The value of `argc` must match the number of integer arguments passed to `func()`, otherwise the behavior is undefined.

The `uc_link` member of `ucontext_t` is used to determine the context that will be resumed when the context being modified by `makecontext()` returns. If the `uc_link` member is not equal to `0`, the process continues as if after a call to `setcontext()` with the context pointed to by the `uc_link` member. If the `uc_link` member is equal to `0`, the process exits as if `exit()` were called. The `uc_link` member should be initialized prior to the call to `makecontext()`.

This function is supported only in a POSIX program.

The `<ucontext.h>` header file defines the `ucontext_t` type as a structure that includes the following members:

<code>mcontext_t</code>	<code>uc_mcontext</code>	A machine-specific representation of the saved context.
<code>ucontext_t</code>	<code>*uc_link</code>	Pointer to the context that will be resumed when this context returns.
<code>sigset_t</code>	<code>uc_sigmask</code>	The set of signals that are blocked when this context is active.
<code>stack_t</code>	<code>uc_stack</code>	The stack used by this context.

### Special Behavior for C++

Because C and C++ linkage conventions are incompatible, `makecontext()` cannot receive C++ function pointers. If you attempt to pass a C++ function pointer to `makecontext()`, the compiler will flag it as an error. To use the C++ `makecontext()` function, you must ensure that all functions registered for `makecontext()` have C linkage by declaring them as extern “C”. For example:

```

C:   void func(int, int);
      :
      makecontext(&context, func, 2, arg1, arg2);

C++: extern "C" void func();
      :
      makecontext(&context, func, 2, arg1, arg2);

```

## Returned Value

makecontext() does not return a value.

The following are the possible values of errno:

**EINVAL** The context being modified is using an alternate stack, and the target function entry point is not a valid Language Environment or C entry point.

The *argc* argument specifies a value less than 0.

**ENOMEM** The *ucp* argument does not have enough stack left to complete the operation. Or more than 15 arguments are passed to the target function, and there is not enough storage to hold all of the arguments.

**Note:** If the target function is in a DLL that has not yet been loaded, then makecontext() cannot determine the size requirement and assumes that the size required is MINSIGSTKSZ. Therefore, in this case, the stack must be at least the size indicated by MINSIGSTKSZ. If the size required by the target function is more than MINSIGSTKSZ, then you must load the DLL before invoking makecontext().

## Example

This example creates a context in main with the *getcontext()* statement, then modifies the context to have its own stack and to invoke the function *func*. It invokes the function with the *setcontext()* statement. Since the **uc\_link** member is set to 0, the process exits when the function returns.

```

/* This example shows the usage of makecontext(). */

#define _XOPEN_SOURCE_EXTENDED 1
#include <stdio.h>
#include <ucontext.h>

#define STACK_SIZE 16384

void func(int);

ucontext_t context, *cp = &context;

int main(void) {
    int value = 1;

    getcontext(cp);
    context.uc_link = 0;
    if ((context.uc_stack.ss_sp = (char *) malloc(STACK_SIZE)) != NULL) {
        context.uc_stack.ss_size = STACK_SIZE;
        context.uc_stack.ss_flags = 0;
        makecontext(cp, func, 1, value);
    }
    else {

```

```

        perror("not enough storage for stack");
        abort();
    }
    printf("context has been built\n");
    setcontext(cp);
    perror("returned from setcontext");
    abort();
}

void func(int arg) {

    printf("function called with value %d\n",arg);
    printf("process will exit when function returns\n");
    return();
}

```

### Output

```

context has been built
function called with value 1
process will exit when function returns

```

### Related Information

- “ucontext.h” on page 52
- “getcontext() — Get User Context” on page 505
- “setcontext() — Restore User Context” on page 1210
- “swapcontext() — Save and Restore User Context” on page 1472

## malloc() — Reserve Storage Block

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
void *malloc(size_t size);
```

### General Description

Reserves a block of storage of *size* bytes. Unlike the `calloc()` function, the content of the storage allocated is indeterminate. The storage to which the returned value points is always aligned for storage of any type of object. Under OS/390 C only, if 4K alignment is required, use the `__4kmalc()` function. (This function is available to C applications in stand-alone Systems Programming Facility applications.) The library functions specific to the System Programming C environment are described in the *OS/390 C/C++ Programming Guide*.

### Special Behavior for C++

The C++ keywords `new` and `delete` are not interoperable with `calloc()`, `free()`, `malloc()`, or `realloc()`.

### Returned Value

Returns a pointer to the reserved space. The storage space to which the returned value points is always suitably aligned for storage of any type of object. The returned value is `NULL` if not enough storage is available, or if *size* was specified as 0. If `malloc()` returns a `NULL` because there is not enough storage, it will also return an error value in `errno`. The following are the possible values of `errno`:

`ENOMEM` Insufficient memory is available

### Example CBC3BM01

```
/* CBC3BM01
   This example prompts you for the number of array entries you want and
   then reserves enough space in storage for the entries.
   If malloc() was successful, the example assigns values to the entries
   and prints out each entry; otherwise, it prints out an error.
*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    long * array;    /* start of the array */
    long * index;    /* index variable */
    int   i;         /* index variable */
    int   num;       /* number of entries of the array */
```



```

printf( "Enter the size of the array\n" );
scanf( "%i", &num );

/* allocate num entries */
if ( (index = array = (long * )malloc( num * sizeof( long ))) != NULL )
{
    for ( i = 0; i < num; ++i )          /* put values in array */
        *index++ = i;                  /* using pointer notation */

    for ( i = 0; i < num; ++i )          /* print the array out */
        printf( "array[ %i ] = %i\n", i, array[i] );
}
else { /* malloc error */
    printf( "Out of storage\n" );
    abort();
}
}

```

### Output

```

Enter the size of the array
array[ 0 ] = 0
array[ 1 ] = 1
array[ 2 ] = 2
array[ 3 ] = 3
array[ 4 ] = 4

```

### Related Information

- “Using the System Programming C Facilities” in the *OS/390 C/C++ Programming Guide*
- “stdlib.h” on page 45
- “calloc() — Reserve and Initialize Storage” on page 141
- “free() — Free a Block of Storage” on page 458
- “realloc() — Change Reserved Storage Block Size” on page 1096

## maxcoll() — Return Maximum Collating Element

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#include <collate.h>
```

```
collcl_t maxcoll(void);
```

### General Description

Returns the largest possible value of a collating element in the current locale.

### Related Information

- “collate.h” on page 23
- “cclass() — Return Characters in a Character Class” on page 149
- “collequiv() — Return a List of Equivalent Collating Elements” on page 200
- “collorder() — Return List of Collating Elements” on page 202
- “collrange() — Calculate the Range List of Collating Elements” on page 204
- “colltostr() — Return a String for a Collating Element” on page 206
- “getmccoll() — Get Next Collating Element from String” on page 554
- “getwmccoll() — Get Next Collating Element from Wide String” on page 632
- “ismccollet() — Identify a Multi-Character Collating Element” on page 710
- “strtocoll() — Return Collating Element for String” on page 1454

## maxdesc() — Get Socket Numbers to Extend Beyond the Default Range

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#define _OPEN_SYS SOCK_EXT
#include <sys/types.h>
#include <sys/socket.h>
```

```
int maxdesc(int *totdesc,int *inetdesc);
```

### General Description

The maxdesc() call reserves space in the address space for socket descriptor numbers that will be activated as bulk mode sockets.

### Parameter Description

- totdesc* A pointer to an integer containing a value 1 greater than the largest desired socket number. The maximum allowed value is the hard limit returned by getrlimit() for RLIMIT\_NOFILE. This value is set by a BPXPRMnn parmlib member on its MAXFILEPROC statement. If you specify a value in *totdesc* greater than the hard limit (or a negative value), the largest socket number will be set to this hard limit, not to the larger value of *totdesc*.
- inetdesc* A pointer to an integer. This is defined to be compatible with previous releases of the maxdesc() interface. The value has no purpose in this implementation.

Set the integer pointed to by *totdesc* to 1 more than the desired maximum socket number. If your program does not use sockets for bulk mode I/O, then you do not need to use the maxdesc() function. If your program uses sockets for bulk mode I/O then set the integer pointed to by *totdesc* to the range of socket descriptors your application may use then add 1. Once this value is accepted, datagram sockets assigned descriptors in this range may be activated for bulk mode I/O. AF\_INET address families are allowed to activate bulk mode I/O. You must call maxdesc() before your program creates its first socket. Your program should use getstablesz() to verify that the number of sockets was changed.

**Note:** Because maxdesc() gives a capability to alter the number of sockets in use the size of the bit sets for the select() call must be increased at compile time. To increase the size of the bit sets you must define FD\_SETSIZE to be at least as large a value as supplied in *totdesc* before including sys/types.h in your program. The default size of FD\_SETSIZE is 2048 sockets as specified in sys/sys\_time.h.

## Returned Value

The value 0 indicates success. The value -1 indicates an error. The value of `errno` indicate the specific error.

### Errno Code      Description

EALREADY	Your program called <code>maxdesc()</code> after creating a socket, after a call to <code>setibmsockopt()</code> , or after a previous call to <code>maxdesc()</code> .
EFAULT	Using the <code>totdesc</code> parameter as specified results in an attempt to access storage outside of the caller's address space, or storage not modifiable by the caller.
ENOMEM	Your address space has insufficient storage.

## Example

The following is an example of the `maxdesc()` call.

```
int totdesc, inetdesc;
totdesc = 100;
inetdesc = 0;
rc = maxdesc(&totdesc, &inetdesc)
```

If successful, your application can create 100 sockets, for address family `AF_INET`. The socket numbers run from 0 through 99.

## Related Information

- “`sys/types.h`” on page 49
- “`sys/socket.h`” on page 48
- “`getstablesz()` — Get the Socket Table Size” on page 611
- “`getrlimit()` — Control Maximum Resource Consumption” on page 591
- “`setrlimit()` — Control Maximum Resource Consumption” on page 1264
- “`select()` — Monitor Activity on Files/Sockets and Message Queues” on page 1159
- “`listen()` — Prepare the Server for Incoming Client Requests” on page 752
- “`open()` — Open a File” on page 872
- “`close()` — Close a File” on page 191

## mblen() — Calculate Length of Multibyte Character

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
int mblen(const char *string, size_t n);
```

### General Description

Determines the length in bytes of the multibyte character pointed to by *string*. A maximum of *n* bytes is examined.

The behavior of this wide-character function is affected by the LC\_CTYPE category of the current locale. Changing the LC\_CTYPE category invalidates the internal shift state: undefined results can occur.

If the current locale supports EBCDIC DBCS characters, then the shift state is updated where applicable. (See “Conforming to ANSI Standards”, in the *OS/390 C/C++ Language Reference*.) The length returned may be up to 4 (for the shift-out character, 2-byte code, and the shift-in character). If *string* is a null pointer, this function resets itself to the initial state.

The function maintains the internal shift state that is altered by subsequent calls.

### Returned Value

If *string* is NULL, the mblen() function returns:

- Nonzero when DBCS-host code (EBCDIC systems) is used
- Nonzero if multibyte encodings are state-dependent
- Zero otherwise

If *string* is not NULL, the mblen() function returns:

- Zero if *string* points to the null character
- The number of bytes comprising the multibyte character
- The value -1 if *string* does not point to a valid multibyte character

### Example

```
#include <locale.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *mbs = "a"
        "\x0E"      /* shift out */
        "\x44\x66"  /* <j0158> */
        "\x44\x76"  /* <j0159> */
        "\x42\x4e"  /* <j0160> */
        "\x0F"      /* shift in */
        "b";
```

```

char *loc = setlocale(LC_ALL, "JA_JP.IBM-939");
int n;

if (!loc) /* setlocale() failure */
{
    exit(8);
}

printf("We're in the %s locale.\n", loc);

n = mblen(NULL, MB_CUR_MAX);
/*****
/* n is nonzero, indicating state-dependent encoding; mblen() has */
/* forced the internal shift state to "initial". */
*****/
printf("n = mblen(NULL, MB_CUR_MAX);    ==> n = %s\n",
       n ? "NONZERO" : "ZERO");

n = mblen(mbs, MB_CUR_MAX);
/*****
/* n is 1, 'a' is a multibyte character of length 1, internal */
/* shift state remains at "initial". */
*****/
printf("n = mblen(mbs, MB_CUR_MAX);    ==> n = %d\n", n);

n = mblen(mbs + 1, MB_CUR_MAX);
/*****
/* n is 3, 'shift out' plus two byte character '<j0158>'. The */
/* internal state changes to "shift out". */
*****/
printf("n = mblen(mbs + 1, MB_CUR_MAX); ==> n = %d\n", n);

n = mblen(mbs + 4, MB_CUR_MAX);
/*****
/* n is 2, two byte character '<j0159>'. The internal shift */
/* state remains "shift out" */
*****/
printf("n = mblen(mbs + 4, MB_CUR_MAX); ==> n = %d\n", n);

n = mblen(mbs + 6, MB_CUR_MAX);
/*****
/* n is 3, two byte character '<j0160>' plus 'shift in'. The */
/* internal shift state returns to "initial". */
*****/
printf("n = mblen(mbs + 6, MB_CUR_MAX); ==> n = %d\n", n);

n = mblen(mbs + 9, MB_CUR_MAX);
/*****
/* n is 1, 'b' is a multibyte character of length 1, internal */
/* shift state remains at "initial". */
*****/
printf("n = mblen(mbs + 9, MB_CUR_MAX); ==> n = %d\n", n);

n = mblen(mbs + 10, MB_CUR_MAX);
/*****
/* n is 0 (end of multibyte character string). */
*****/
printf("n = mblen(mbs + 10, MB_CUR_MAX); ==> n = %d\n", n);

return 0;
}

```

## Output

We're in the JA\_JP.IBM-939 locale.

```
n = mblen(NULL, MB_CUR_MAX);      ==> n = NONZERO
n = mblen(mbs, MB_CUR_MAX);        ==> n = 1
n = mblen(mbs + 1, MB_CUR_MAX);    ==> n = 3
n = mblen(mbs + 4, MB_CUR_MAX);    ==> n = 2
n = mblen(mbs + 6, MB_CUR_MAX);    ==> n = 3
n = mblen(mbs + 9, MB_CUR_MAX);    ==> n = 1
n = mblen(mbs + 10, MB_CUR_MAX);   ==> n = 0
```

## Related Information

- “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*
- “locale.h” on page 33
- “stdlib.h” on page 45
- “mbrlen() — Calculate Length of Multibyte Character” on page 794
- “mbrtowc() — Convert a Multibyte Character to a Wide Character” on page 797
- “mbsrtowcs() — Convert a Multibyte String to a Wide-Character String” on page 802
- “mbstowcs() — Convert Multibyte Characters to Wide Characters” on page 804
- “mbtowc() — Convert Multibyte Character to Wide Character” on page 806
- “setlocale() — Set Locale” on page 1241
- “strlen() — Determine String Length” on page 1436
- “wctomb() — Convert a Wide Character to a Multibyte Character” on page 1697
- “wcslen() — Calculate Length of Wide-Character String” on page 1715
- “wcsrtombs() — Convert Wide-Character String to Multibyte String” on page 1726
- “wctomb() — Convert Wide Character to Multibyte Character” on page 1751

## mbrlen() — Calculate Length of Multibyte Character

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

### Format

#### Non-XPG4

```
#include <wchar.h>
```

```
size_t mbrlen (const char *s, size_t n, mbstate_t *ps);
```

#### XPG4

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>
```

```
size_t mbrlen (const char *s, size_t n, mbstate_t *ps);
```

### General Description

Calculates the number of bytes required to return to the initial shift state. This is equivalent to

```
mbrtowc((wchar_t *)0, s, n, ps != NULL ? ps : &internal);
```

where `&internal` is the address of the internal `mbstate_t` object for the `mbrlen()` function.

`mbrlen()` is a restartable version of `mblen()`. That is, shift state information is passed as one of the arguments, and is updated on exit. With `mbrlen()`, you can switch from one multibyte string to another, provided that you have kept the shift-state information.

The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you change the category, undefined results can occur.

### Special Behavior for XPG4

If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the `wchar` header, then you must also define the `_MSE_PROTOS` feature test macro to make the declaration of the `mbrlen()` function in the `wchar` header available when you compile your program. Please see Table 3 on page 18 for a list of XPG4 and other feature test macros.

### Returned Value

If `s` is a null pointer, the `mbrlen()` function resets the shift state to the initial shift state and returns the value 0.

If `s` is not a null pointer, the `mbrlen()` function returns the first of the following that applies.



- 0 If the next  $n$  or fewer bytes complete the valid multibyte character that corresponds to the null wide character.
- positive If the next  $n$  or fewer bytes complete the valid multibyte character; the value returned is the number of bytes that complete the multibyte character.
- 2 If the next  $n$  bytes form an incomplete (but potentially valid) multibyte character, and all  $n$  bytes have been processed; it is unspecified whether this can occur when the value of  $n$  is less than that of the MB\_CUR\_MAX macro.
- Note:** When a -2 value is returned, and  $n$  is at least MB\_CUR\_MAX, the string would contain redundant shift-out and shift-in characters. To continue processing the multibyte string, increment the pointer by the value  $n$ , and call the mbrtowc() function.
- 1 If an encoding error occurs (when the next  $n$  or fewer bytes do not contribute to the complete and valid multibyte character), the value of the macro EILSEQ is stored in errno, but the conversion state remains unchanged.

### Example

#### CBC3BM03

```
/* CBC3BM03 */
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
    char      mbs[5] = "a";    /* string containing the multibyte char */
    mbstate_t ss      = 0;     /* set shift state to the initial state */
    int       length;

    /* Determine the length in bytes of a multibyte character pointed
    /* to by mbs. */

    length = mbrlen(mbs, MB_CUR_MAX, &ss);

    printf("    length: %d \n", length);
    printf("    mbs: \"%s\" \n", mbs);
    printf("MB_CUR_MAX: %d \n", MB_CUR_MAX);
    printf("    ss: %d \n", ss);
}
```

### Output

```
    length: 1
    mbs: "a"
MB_CUR_MAX: 4
    ss: 0
```

### Related Information

- “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*
- “locale.h” on page 33
- “wchar.h” on page 54
- “mblen() — Calculate Length of Multibyte Character” on page 791
- “mbtowc() — Convert Multibyte Character to Wide Character” on page 806

- “`mbrtowc()` — Convert a Multibyte Character to a Wide Character” on page 797
- “`mbsrtowcs()` — Convert a Multibyte String to a Wide-Character String” on page 802
- “`setlocale()` — Set Locale” on page 1241
- “`wcrtomb()` — Convert a Wide Character to a Multibyte Character” on page 1697
- “`wcsrtombs()` — Convert Wide-Character String to Multibyte String” on page 1726

## mbrtowc() — Convert a Multibyte Character to a Wide Character

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

### Format

#### Non-XP4

```
#include <wchar.h>
```

```
size_t mbrtowc (wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);
```

#### XP4

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>
```

```
size_t mbrtowc (wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);
```

### General Description

The `mbrtowc()` function is equivalent to `mbrtowc(NULL, "", 1, ps)`.

If `s` is a null pointer, the `mbrtowc()` function ignores the `n` and the `pwc`, and resets the shift state, pointed to by `ps`, to the initial shift state.

If `s` is not a null pointer, `mbrtowc()` inspects at most `n` bytes, beginning with the byte pointed to by `s`, and the shift state pointed to by `ps`, and determines the number of bytes that is needed to complete the valid multibyte character.

When the multibyte character is completed, `mbrtowc()` determines the value of the corresponding wide character and stores it in the object pointed to by `pwc`, so long as `pwc` is not a null pointer. Finally, `mbrtowc()` stores the actual shift state in the object pointed to by `ps`. If `ps` is a null pointer, `mbrtowc()` uses its own internal object to track the shift state.

`mbrtowc()` is a restartable version of `mbtowc()`. That is, shift-state information is passed as one of the arguments and is updated on exit. With `mbrtowc()`, you can switch from one multibyte string to another, provided that you have kept the shift-state information.

The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you change the category, undefined results may occur.

### Special Behavior for XP4

If you define any feature test macro specifying XP4 behavior before the statement in your program source file to include the `wchar` header, then you must also define the `_MSE_PROTOS` feature test macro to make the declaration of the `mbrtowc()` function in the `wchar` header available when you compile your program. Please see Table 3 on page 18 for a list of XP4 and other feature test macros.

## Returned Value

If *s* is a null pointer, the `mbrtowc()` function resets the shift state to the initial shift state and returns the value 0.

If *s* is not a null pointer, the `mbrtowc()` function returns one of the following, in the order shown.

- |                  |   |
|------------------|---|
| 0                | If the next <i>n</i> or fewer bytes complete the valid multibyte character that corresponds to the null wide character.   |
| positive integer | If the next <i>n</i> or fewer bytes complete the valid multibyte character; the value returned is the number of bytes that complete the multibyte character.  |
| -2               | If the next <i>n</i> bytes form an incomplete (but potentially valid) multibyte character, and all <i>n</i> bytes have been processed. It is unspecified whether this can occur when the value of <i>n</i> is less than that of the <code>MB_CUR_MAX</code> macro.<br><br><b>Note:</b> When a -2 value is returned, and <i>n</i> is at least <code>MB_CUR_MAX</code> , the string would contain redundant shift-out and shift-in characters. To continue processing the multibyte string, increment the pointer by the value <i>n</i> , and call the <code>mbrtowc()</code> function. |
| -1               | If an encoding error occurs (when the next <i>n</i> or fewer bytes do not contribute to the complete and valid multibyte character). The value of the macro <code>EILSEQ</code> is stored in <code>errno</code> , but the conversion state is unchanged.  |

## Example CBC3BM04

```
/* CBC3BM04 */
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
    wchar_t    wc;
    char       mbs[5] = "a";    /* string containing the multibyte char */
    mbstate_t  ss = 0;          /* set shift state to the initial state */
    int        length;

    /* Determine the length of the multibyte character pointed to by
     * mbs. Store the multibyte character in the wchar_t object
     * called wc.
     */

    length = mbrtowc(&wc, mbs, MB_CUR_MAX, &ss);

    printf("    length: %d \n", length);
    printf("        wc: '%lc' \n", wc);
    printf("        mbs: \"%s\" \n", mbs);
    printf("MB_CUR_MAX: %d \n", MB_CUR_MAX);
    printf("        ss: %d \n", ss);
}
```

## Related Information

- “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*
- “locale.h” on page 33
- “wchar.h” on page 54
- “mblen() — Calculate Length of Multibyte Character” on page 791
- “mbrlen() — Calculate Length of Multibyte Character” on page 794
- “mbsrtowcs() — Convert a Multibyte String to a Wide-Character String” on page 802
- “setlocale() — Set Locale” on page 1241
- “wctomb() — Convert a Wide Character to a Multibyte Character” on page 1697
- “wcsrtombs() — Convert Wide-Character String to Multibyte String” on page 1726

## mbsinit() — Test State Object for Initial State

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

### Format

#### Non-XPG4

```
#include <wchar.h>

int mbsinit( const mbstate_t *ps );
```

#### XPG4

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

int mbsinit( const mbstate_t *ps );
```

### General Description

If *ps* is not a null pointer the mbsinit() function determines whether the pointer to mbstate\_t object describes an initial conversion state.

The behavior of this wide-character function is affected by the LC\_CTYPE category of the current locale. If you change the category, undefined results can occur.

### Special Behavior for XPG4

If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, then you must also define the \_MSE\_PROTOS feature test macro to make the declaration of the mbsinit() function in the wchar header available when you compile your program. Please see Table 3 on page 18 for a list of XPG4 and other feature test macros.

### Returned Value

Returns nonzero if *ps* is a null pointer or if the pointed-to object describes an initial conversion state; otherwise, it returns 0.

### Example

#### CBC3BM05

```
/* CBC3BM05
   This example checks the conversion state to see if it is in the initial state.
*/
#include "stdio.h"
#include "wchar.h"
#include "stdlib.h"

main() {
    char    *string = "ABC";
    mbstate_t state = 0;
    wchar_t  wc;
    int      rc;

    rc = mbrtowc(&wc, string, MB_CUR_MAX, &state);
```

```

    if (mbsinit(&state))
        printf("In initial conversion state\n");
}

```

### Related Information

- “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*
- “locale.h” on page 33
- “wchar.h” on page 54
- “mbrlen() — Calculate Length of Multibyte Character” on page 794
- “mbrtowc() — Convert a Multibyte Character to a Wide Character” on page 797
- “mbsrtowcs() — Convert a Multibyte String to a Wide-Character String” on page 802
- “setlocale() — Set Locale” on page 1241
- “wctomb() — Convert a Wide Character to a Multibyte Character” on page 1697
- “wcsrtombs() — Convert Wide-Character String to Multibyte String” on page 1726

## mbsrtowcs() — Convert a Multibyte String to a Wide-Character String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

### Format

#### Non-XPG4

```
#include <wchar.h>
```

```
size_t mbsrtowcs(wchar_t *dst, const char **src, size_t len, mbstate_t *ps);
```

#### XPG4

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>
```

```
size_t mbsrtowcs(wchar_t *dst, const char **src, size_t len, mbstate_t *ps);
```

### General Description

Converts a sequence of multibyte characters that begins in the conversion state described by *ps* from the array indirectly pointed to by *src*. It converts this sequence into a sequence of corresponding wide characters, that, if *dst* is not a null pointer, are then stored into the array pointed to by *dst*. Conversion continues up to and including a terminating null character, and the terminating null wide character is also stored. Conversion stops earlier in two cases: (1) when a sequence of bytes is reached that does not form a valid multibyte character, or (2) if *dst* is not a null pointer, when *len* codes have been stored into the array pointed to by *dst*. Each conversion takes place as if by a call to the `mbrtowc()` function.

If *dst* is not a null pointer, the pointer object pointed to by *src* is assigned either a null pointer (if conversion stopped because a terminating null character was reached) or the address just past the last multibyte character converted. If conversion stopped because a terminating null character was reached, the resulting state is the initial state.

`mbsrtowcs()` is a restartable version of `mbstowcs()`. That is, shift-state information is passed as one of the arguments and is updated on exit. With `mbsrtowcs()`, you can switch from one multibyte string to another, provided that you have kept the shift-state information.

The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you change the category, undefined results can occur.

### Special Behavior for XPG4

If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the `wchar` header, then you must also define the `_MSE_PROTOS` feature test macro to make the declaration of the `mbsrtowcs()` function in the `wchar` header available when you compile your program. Please see Table 3 on page 18 for a list of XPG4 and other feature test macros.



## Returned Value

If the input string contains an invalid multibyte character, an encoding error occurs. The `mbsrtowcs()` function stores the value of the macro `EILSEQ` in `errno`, indicating an encoding error and returns `(size_t)-1`, and the conversion state is undefined. Otherwise it returns the number of multibyte characters successfully converted, not including the terminating null character, if any.

## Example CBC3BM06

```
/* CBC3BM06 */
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

#define SIZE 10

int main(void)
{
    wchar_t wcs[SIZE];
    char    mbs[SIZE]="abcd"; /* string containing the multibyte char */
    char    *ptr    = mbs;    /* pointer to the mbs string          */
    int     length;

    /* Determine the length of the multibyte string pointed to by      */
    /* mbs. Store the multibyte characters in the wchar_t array        */
    /* pointed to by wcs.                                             */

    length = mbsrtowcs(wcs, (const char**)&ptr, SIZE, NULL);
    wcs[length] = L'\0';

    printf("    length: %d \n", length);
    printf("        wcs: \"%ls\" \n", wcs);
    printf("        mbs: \"%s\" \n", mbs);
    printf("        &mbs: %p \n", mbs);
    printf("        &ptr: %p \n", ptr);
}
```

## Related Information

- “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*
- “locale.h” on page 33
- “wchar.h” on page 54
- “mblen() — Calculate Length of Multibyte Character” on page 791
- “mbrlen() — Calculate Length of Multibyte Character” on page 794
- “mbrtowc() — Convert a Multibyte Character to a Wide Character” on page 797
- “mbstowcs() — Convert Multibyte Characters to Wide Characters” on page 804
- “setlocale() — Set Locale” on page 1241
- “wctomb() — Convert a Wide Character to a Multibyte Character” on page 1697
- “wcsrtombs() — Convert Wide-Character String to Multibyte String” on page 1726

mbstowcs() — Convert Multibyte Characters to Wide Characters

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

Format

```
#include <stdlib.h>

size_t mbstowcs(wchar_t *pwc, const char *string, size_t n);
```

General Description

Determines the length of the sequence of the multibyte characters that start in the initial shift state and that are pointed to by *string*. It then converts each of the multibyte characters to a *wchar\_t*, and stores no more than *n* codes in the array pointed to by *pwc*. The conversion stops if either an invalid multibyte sequence is encountered or if *n* codes have been converted.

Processing continues up to and including the terminating null character, and characters that follow it are not processed. The terminating null character is converted into a code with the value 0.

The behavior of this wide-character function is affected by the LC\_CTYPE category of the current locale. If you change the category, undefined results can occur.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

Returned Value

If an invalid multibyte character is encountered, the function returns (size\_t)-1. Otherwise, it returns the number of *pwc* array elements modified, not counting the terminating 0 code (the *wchar\_t* 0 code), which is 0 if *pwc* is a null pointer. Note that, if the return value is *n*, the resulting *wchar\_t* array will *not* be null terminated.

Example  
CBC3BM07

```
/* CBC3BM07
   This example uses mbstowcs() to convert a multibyte character string to
   a wide character string.
*/
#include <stdio.h>
#include <stdlib.h>

int main()
{
    char      mbsin[8] = "\x50\x0e\x42\xf1\x0f\x50\x00";
    wchar_t   wcsout[5];
    size_t    wcssize;

    printf("mbsin is 0x%.2x 0x%.2x 0x%.2x 0x%.2x 0x%.2x 0x%.2x 0x%.2x\n",
           mbsin[0], mbsin[1], mbsin[2],
           mbsin[3], mbsin[4], mbsin[5],
           mbsin[6]);
```

```

wcssize = mbstowcs(wcsout, mbsin, 5);

printf("mbstowcs(wcsout, mbsin, 5); returned %d\n", wcssize);

printf("wcsout is 0x%.4x 0x%.4x 0x%.4x 0x%.4x\n",
      wcsout[0], wcsout[1],
      wcsout[2], wcsout[3]);
}

```

### Output

```

mbsin is 0x50 0x0e 0x42 0xf1 0x0f 0x50 0x00
mbstowcs(wcsout, mbsin, 5); returned 3
wcsout is 0x0050 0x42f1 0x0050 0x0000

```

### Related Information

- “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*
- “locale.h” on page 33
- “stdlib.h” on page 45
- “mblen() — Calculate Length of Multibyte Character” on page 791
- “mbsrtowcs() — Convert a Multibyte String to a Wide-Character String” on page 802
- “mbtowc() — Convert Multibyte Character to Wide Character” on page 806
- “setlocale() — Set Locale” on page 1241
- “wcslen() — Calculate Length of Wide-Character String” on page 1715
- “wcstombs() — Convert Wide-Character String to Multibyte Character String” on page 1740

## mbtowc() — Convert Multibyte Character to Wide Character

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
int mbtowc(wchar_t *pwc, const char *string, size_t n);
```

### General Description

Converts a multibyte character to a wide character and returns the number of bytes of the multibyte character. It first determines the length of the multibyte character pointed to by *string*. It then converts the multibyte character to the corresponding wide character and places the wide character in the location pointed to by *pwc*, if *pwc* is not a null pointer. A maximum of *n* bytes is examined.

The behavior of this wide-character function is affected by the LC\_CTYPE category of the current locale. If you change the category, undefined results can occur.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

If *string* is NULL, the mbtowc() function returns:

- Nonzero if the multibyte encoding in the current locale (LC\_CTYPE) is shift-dependent.
- 0 otherwise.
- The current shift state is set to the initial state.

Otherwise if *string* is not NULL, the mbtowc() function returns:

- The number of bytes comprising the converted multibyte character, if *n* or fewer bytes form a valid multibyte character.
- 0 if *string* points to the null character.
- -1 if *string* does not point to a valid multibyte character, and the next *n* bytes do not form a valid multibyte character.

If the current locale supports EBCDIC DBCS characters, the shift state is updated where applicable. The length returned may be up to 4 characters long (for the shift-out character, 2-byte code, and the shift-in character).

After the function is placed into its initial state, it interprets multibyte characters—pointed to by *string*—accordingly. During the processing of shift-dependent encoded characters, you cannot stop processing one string, then move temporarily to processing another string, and return to the first, because the state would be valid for the second string, not the place where you stopped in the first string.

## Example

```

/* This example uses mbtowc() to convert a multibyte character into a wide
   character.
   */
#include <stdio.h>
#include <stdlib.h>

int temp;
char string [6];
wchar_t arr[6];

int main(void)
{ /* Set string to point to a multibyte character. */
  :
  temp = mbtowc(arr, string, MB_CUR_MAX);
  printf("wide character string: %ls",arr);
}

```

## Related Information

- “Internationalization: Locales and Character Sets” in the *OS/390 C/C++ Programming Guide*
- “locale.h” on page 33
- “stdlib.h” on page 45
- “mblen() — Calculate Length of Multibyte Character” on page 791
- “mbrtowc() — Convert a Multibyte Character to a Wide Character” on page 797
- “mbstowcs() — Convert Multibyte Characters to Wide Characters” on page 804
- “setlocale() — Set Locale” on page 1241
- “wcslen() — Calculate Length of Wide-Character String” on page 1715
- “wctomb() — Convert Wide Character to Multibyte Character” on page 1751

## memccpy() — Copy Bytes in Memory

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <string.h>
```

```
void * memccpy(void *s1, const void *s2, int c, size_t n);
```

### General Description

The `memccpy()` function copies bytes from memory area `s2` into memory area `s1`, stopping after the first occurrence of byte `c` (converted to an unsigned char) is copied, or after `n` bytes are copied, whichever comes first.

### Returned Value

The `memccpy()` function returns a pointer to the byte after the copy of `c` in `s1`, or a null pointer if `c` was not found in the first `n` bytes of `s2`.

### Related Information

None.

## memchr() — Search Buffer

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <string.h>
```

```
void *memchr(const void *buf, int c, size_t count);
```

### General Description

The `memchr()` built-in function searches the first *count* bytes pointed to by *buf* for the first occurrence of *c* converted to an unsigned character. The search continues until it finds *c* or examines *count* bytes.

### Returned Value

Returns a pointer to the location of *c* in *buf*. It returns NULL if *c* is not within the first *count* bytes of *buf*.

### Example

#### CBC3BM11

```
/* CBC3BM11
   This example finds the first occurrence of "x" in the string that you
   provide. If it is found, the string that starts with that character is printed.
   If you compile this code as MYPROG, then it could be invoked like this,
   with exactly one parameter: MYPROG skiing
*/
#include <stdio.h>
#include <string.h>

int main(int argc, char ** argv)
{
    char * result;

    if ( argc != 2 )
        printf( "Usage: %s string\n", argv[0] );
    else
    {
        if ((result = (char *)memchr( argv[1], 'x', strlen(argv[1]))) ) != NULL)
            printf( "The string starting with x is %s\n", result );
        else
            printf( "The letter x cannot be found in the string\n" );
    }
}
```

### Output

The string starting with x is xing

### **Related Information**

- “string.h” on page 46
- “memcmp() — Compare Bytes” on page 811
- “memcpy() — Copy Buffer” on page 813
- “memmove() — Move Buffer” on page 814
- “memset() — Set Buffer to Value” on page 816
- “strchr() — Search for Character” on page 1416



## memcmp() — Compare Bytes

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <string.h>
```

```
int memcmp(const void *buf1, const void *buf2, size_t count);
```

### General Description

The memcmp() built-in function compares the first *count* bytes of *buf1* and *buf2*.

The relation is determined by the sign of the difference between the values of the leftmost first pair of bytes that differ. The values depend on EBCDIC encoding. This function is *not* locale sensitive.

### Returned Value

Indicates the relationship between *buf1* and *buf2* as follows:

Value	Meaning
Less than 0	The contents of the buffer pointed to by <i>buf1</i> less than the contents of the buffer pointed to by <i>buf2</i>
0	The contents of the buffer pointed to by <i>buf1</i> identical to the contents of the buffer pointed to by <i>buf2</i>
Greater than 0	The contents of the buffer pointed to by <i>buf1</i> greater than the contents of the buffer pointed to by <i>buf2</i>

### Example CBC3BM12

```
/* CBC3BM12
   This example compares first and second arguments passed to main to
   determine which, if either, is greater.
*/
#include <stdio.h>
#include <string.h>

int main(int argc, char ** argv)
{
    int len;
    int result;

    if ( argc != 3 )
    {
        printf( "Usage: %s string1 string2\n", argv[0] );
    }
    else
    {
        /* Determine the length to be used for comparison */
```

```

    if (strlen( argv[1] ) < strlen( argv[2] ))
        len = strlen( argv[1] );
    else
        len = strlen( argv[2] );

    result = memcmp( argv[1], argv[2], len );

    printf( "When the first %i characters are compared,\n", len );
    if ( result == 0 )
        printf( "\"%s\" is identical to \"%s\"\n", argv[1], argv[2] );
    else if ( result < 0 )
        printf( "\"%s\" is less than \"%s\"\n", argv[1], argv[2] );
    else
        printf( "\"%s\" is greater than \"%s\"\n", argv[1], argv[2] );
}
}

```

**Output**

If the program is passed the arguments *firststring* and *secondstring*, you would obtain following:

When the first 11 characters are compared,  
 “firststring” is less than “secondstring”

**Related Information**

- “string.h” on page 46
- “memchr() — Search Buffer” on page 809
- “memcpy() — Copy Buffer” on page 813
- “memmove() — Move Buffer” on page 814
- “memset() — Set Buffer to Value” on page 816
- “strcmp() — Compare Strings” on page 1418

## memcpy() — Copy Buffer

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <string.h>
```

```
void *memcpy(void *dest, const void *src, size_t count);
```

### General Description

The `memcpy()` built-in function copies *count* bytes from the object pointed to by *src* to the object pointed to by *dest*. See “Built-in Functions” on page 64 for information about the use of built-in functions. For `memcpy()`, the source characters may be overlaid if copying takes place between objects that overlap. Use the `memmove()` function to allow copying between objects that overlap.

### Returned Value

Returns the value of *dest*.

### Example CBC3BM13

```
/* CBC3BM13
   This example copies the contents of source to target.
*/
#include <string.h>
#include <stdio.h>

#define MAX_LEN 80

char source[ MAX_LEN ] = "This is the source string";
char target[ MAX_LEN ] = "This is the target string";

int main(void)
{
    printf( "Before memcpy, target is \"%s\"\n", target );
    memcpy( target, source, sizeof(source));
    printf( "After memcpy, target becomes \"%s\"\n", target );
}
```

### Output

Before memcpy, target is "This is the target string"  
After memcpy, target becomes "This is the source string"

### Related Information

- “string.h” on page 46
- “memchr() — Search Buffer” on page 809
- “memcmp() — Compare Bytes” on page 811
- “memmove() — Move Buffer” on page 814
- “memset() — Set Buffer to Value” on page 816
- “strcpy() — Copy String” on page 1422

## memmove() — Move Buffer

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <string.h>
```

```
void *memmove(void *dest, const void *src, size_t count);
```

### General Description

Copies *count* bytes from the object pointed to by *src* to the object pointed to by *dest*. The `memmove()` function allows copying between possibly overlapping objects as if the *count* bytes of the object pointed to by *src* must first copied into a temporary array before being copied to the object pointed to by *dest*.

### Returned Value

Returns the value of *dest*.

### Example

#### CBC3BM14

```
/* CBC3BM14
   This example copies the word shiny from position target + 2 to position
   target + 8.
*/
#include <string.h>
#include <stdio.h>
#define SIZE    21

char target[SIZE] = "a shiny white sphere";

int main( void )
{
    char * p = target + 8; /* p points at the starting character
                           of the word we want to replace */
    char * source = target + 2; /* start of "shiny" */

    printf( "Before memmove, target is \"%s\"\n", target );
    memmove( p, source, 5 );
    printf( "After memmove, target becomes \"%s\"\n", target );
}
```

### Output

```
Before memmove, target is "a shiny white sphere"
After memmove, target becomes "a shiny shiny sphere"
```

**Related Information**

- “string.h” on page 46
- “memchr() — Search Buffer” on page 809
- “memcmp() — Compare Bytes” on page 811
- “memcpy() — Copy Buffer” on page 813
- “memset() — Set Buffer to Value” on page 816
- “strcpy() — Copy String” on page 1422

## memset() — Set Buffer to Value

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <string.h>
```

```
void *memset(void *dest, int c, size_t count);
```

### General Description

The `memset()` built-in function sets the first *count* bytes of *dest* to the value *c* converted to an unsigned int.

### Returned Value

Returns the value of *dest*.

### Example CBC3BM15

```
/* CBC3BM15
   This example sets 10 bytes of the buffer to "A" and the next 10 bytes
   to "B".
*/
#include <string.h>
#include <stdio.h>
#define BUF_SIZE 20
#define HALF_BUF_SIZE BUF_SIZE/2

int main(void)
{
    char buffer[BUF_SIZE + 1];
    char *string;

    memset(buffer, 0, sizeof(buffer));
    string = (char *)memset(buffer, 'A', HALF_BUF_SIZE);
    printf("\nBuffer contents: %s\n", string);
    memset(buffer+HALF_BUF_SIZE, 'B', HALF_BUF_SIZE);
    printf("\nBuffer contents: %s\n", buffer);
}
```

### Output

Buffer contents: AAAAAAAAAA

Buffer contents: AAAAAAAAAABBBBBBBBBB

### Related Information

- “string.h” on page 46
- “memchr() — Search Buffer” on page 809
- “memcmp() — Compare Bytes” on page 811
- “memcpy() — Copy Buffer” on page 813
- “memmove() — Move Buffer” on page 814

mkdir() — Make a Directory

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

Format

```
#define _POSIX_SOURCE
#include <sys/stat.h>

int mkdir(const char *pathname, mode_t mode);
```

General Description

Creates a new, empty directory, *pathname*. The file permission bits in *mode* are modified by the file creation mask of the process, and then used to set the file permission bits of the directory being created. For more information on the file creation mask, see “umask() — Set and Retrieve File Creation Mask” on page 1647.

| To provide an ASCII input/output format for applications using this function, define  
| feature test macro \_\_LIBASCII as described on page 22.

The *mode* argument is created with one of the following symbols defined in the sys/stat.h header file.

Any mode flags that are not defined will be turned off, and the function will be allowed to proceed.

- S\_ISUID Privilege to set the user ID (UID) for execution. When this file is run through an exec function, the effective user ID of the process is set to the owner of the file. The process then has the same authority as the file owner, rather than the authority of the actual invoker.
- S\_ISGID Privilege to set group ID (GID) for execution. When this file is run through an exec function, the effective group ID of the process is set to the group ID of the file. The process then has the same authority as the file owner, rather than the authority of the actual invoker.
- S\_ISVTX Indicates shared text. Keep loaded as an executable file in storage.
- S\_IRUSR Read permission for the file owner.
- S\_IWUSR Write permission for the file owner.
- S\_IXUSR Search permission (for a directory) or execute permission (for a file) for the file owner.
- S\_IRWXU Read, write, and search, or execute, for the file owner; S\_IRWXG is the bitwise inclusive OR of S\_IRUSR, S\_IWUSR, and S\_IXUSR.
- S\_IRGRP Read permission for the file's group.

**S\_IWGRP**

Write permission for the file's group.

**S\_IXGRP** Search permission (for a directory) or execute permission (for a file) for the file's group.

**S\_IRWXG**

Read, write, and search or execute permission for the file's group. S\_IRWXG is the bitwise inclusive **OR** of S\_IRGRP, S\_IWGRP, and S\_IXGRP.

**S\_IROTH** Read permission for users other than the file owner.

**S\_IWOTH**

Write permission for users other than the file owner.

**S\_IXOTH** Search permission for a directory, or execute permission for a file, for users other than the file owner.

**S\_IRWXO**

Read, write, and search or execute permission for users other than the file owner. S\_IRWXO is the bitwise inclusive OR of S\_IROTH, S\_IWOTH, and S\_IXOTH.

The owner ID of the new directory is set to the effective user ID of the process. The group ID of the new directory is set to the group ID of the owning directory.

mkdir() sets the access, change, and modification times for the new directory. It also sets the change and modification times for the directory that contains the new directory.

If *pathname* names a symbolic link, mkdir() fails.

### Returned Value

If successful, mkdir() returns the value 0. If unsuccessful, it does not create a directory, it returns the value -1, and sets errno to one of the following:

EACCES	The process did not have search permission on some component of <i>pathname</i> , or did not have write permission on the parent directory of the directory to be created.
EEXIST	Either the named file refers to a symbolic link, or there is already a file or directory with the given <i>pathname</i> .
ELOOP	A loop exists in symbolic links. This error is issued if more than POSIX_SYMLINK_MAX (defined in the limits.h header file) symbolic links are detected in the resolution of <i>pathname</i> .
EMLINK	The link count of the parent directory has already reached LINK_MAX (defined in the limits.h header file).
ENAMETOOLONG	<i>pathname</i> is longer than PATH_MAX characters or some component of <i>pathname</i> is longer than NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH_MAX. The PATH_MAX and NAME_MAX values can be determined using pathconf().



ENOENT	Some component of <i>pathname</i> does not exist, or <i>pathname</i> is an empty string.
ENOSPC	The file system does not have enough space to contain a new directory, or the parent directory cannot be extended.
ENOTDIR	A component of the <i>pathname</i> prefix is not a directory.
EROFS	The parent directory of the directory to be created is on a read-only file system.

### Example CBC3BM16

```

/* CBC3BM16
   The following example creates a new directory.
   */
#define _POSIX_SOURCE
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

main() {
    char new_dir[]="new_dir";

    if (mkdir(new_dir, S_IRWXU|S_IRGRP|S_IXGRP) != 0)
        perror("mkdir() error");
    else if (chdir(new_dir) != 0)
        perror("first chdir() error");
    else if (chdir("..") != 0)
        perror("second chdir() error");
    else if (rmdir(new_dir) != 0)
        perror("rmdir() error");
    else
        puts("success!");
}

```

### Related Information

- “limits.h” on page 32
- “sys/stat.h” on page 48
- “chdir() — Change the Working Directory” on page 167
- “chmod() — Change the Mode of a File or Directory” on page 174
- “stat() — Get File Information” on page 1404
- “umask() — Set and Retrieve File Creation Mask” on page 1647

## mkfifo() — Make a FIFO Special File

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <sys/stat.h>
```

```
int mkfifo(const char *pathname, mode_t mode);
```

### General Description

Sets the access, change, and modification times for the new file. It also sets the change and modification times for the directory that contains the new file.

mkfifo() creates a new FIFO special file, *pathname*. The file permission bits in *mode* are changed by the file creation mask of the process, and then used to set the file permission bits of the FIFO file being created. If *pathname* contains a symbolic link, mkfifo() fails. For more information on the file creation mask, see “umask() — Set and Retrieve File Creation Mask” on page 1647; for information about the file permission bits, see “chmod() — Change the Mode of a File or Directory” on page 174.

The owner ID of the FIFO file is set to the effective user ID of the process. The group ID of the FIFO file is set to the group ID of the owning directory. *pathname* cannot end in a symbolic link.

### Returned Value

If mkfifo() succeeds, it returns the value 0. If mkfifo() fails, no FIFO file is created, the function returns the value -1, and errno is set to one of the following:

- EACCES** The process does not have search permission on some component of *pathname*, or does not have write permission on the parent directory of the file to be created.
- EEXIST** Either the named file refers to a symbolic link, or there is already a file or directory with the given *pathname*.
- EINTR** A signal is received while this open is blocked waiting for an open() for read (if O\_WRONLY was specified) or for an open() for write (if O\_RDONLY was specified).
- ELOOP** A loop exists in symbolic links. This error is issued if more than POSIX\_SYMLINK\_MAX (defined in the limits.h header file) symbolic links are detected in the resolution of *pathname*.
- EMLINK** The link count of the parent directory has already reached the maximum defined for the system.
- ENAMETOOLONG** *pathname* is longer than PATH\_MAX characters or some component of *pathname* is longer than NAME\_MAX characters while \_POSIX\_NO\_TRUNC is in effect. For symbolic links, the length of the

path name string substituted for a symbolic link exceeds PATH\_MAX. The PATH\_MAX and NAME\_MAX values can be determined using pathconf().

- ENOENT    Some component of *pathname* does not exist, or *pathname* is an empty string.
- ENOSPC    The file system does not have enough space to contain a new file, or the parent directory cannot be extended.
- ENOTDIR   A component of the *pathname* prefix is not a directory.
- EROFS     The parent directory of the FIFO file is on a read-only file system.

### Example

#### CBC3BM17

```

/* CBC3BM17 */
    This example uses mkfifo() to create a FIFO special file named
    temp.fifo and then writes and reads from the file before closing it.
*/
#define _POSIX_SOURCE
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

main() {
    char fn[]="temp.fifo";
    char out[20]="FIFO's are fun!", in[20];
    int rfd, wfd;

    if (mkfifo(fn, S_IRWXU) != 0)
        perror("mkfifo() error");
    else {
        if ((rfd = open(fn, O_RDONLY|O_NONBLOCK)) < 0)
            perror("open() error for read end");
        else {
            if ((wfd = open(fn, O_WRONLY)) < 0)
                perror("open() error for write end");
            else {
                if (write(wfd, out, strlen(out)+1) == -1)
                    perror("write() error");
                else if (read(rfd, in, sizeof(in)) == -1)
                    perror("read() error");
                else printf("read '%s' from the FIFO\n", in);
                close(wfd);
            }
            close(rfd);
        }
        unlink(fn);
    }
}

```

### Output

read 'FIFO's are fun!' from the FIFO

### **Related Information**

- “limits.h” on page 32
- “sys/stat.h” on page 48
- “chmod() — Change the Mode of a File or Directory” on page 174
- “open() — Open a File” on page 872
- “pipe() — Create an Unnamed Pipe” on page 907
- “stat() — Get File Information” on page 1404
- “umask() — Set and Retrieve File Creation Mask” on page 1647

mknod() — Make a Directory or File

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 OS/390 UNIX	both	MVS 4.3

Format  
\_OPEN\_SYS

```
#define _OPEN_SYS
#include <sys/stat.h>

int mknod(const char *path, mode_t mode, rdev_t dev_identifier);
```

\_XOPEN\_SOURCE\_EXTENDED 1

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/stat.h>

int mknod(const char *path, mode_t mode, dev_t dev_identifier);
```

General Description

Creates a new directory, regular file, character special file, or FIFO special file (named pipe), with the path name specified in the *path* argument.

| To provide an ASCII input/output format for applications using this function, define  
| feature test macro \_\_LIBASCII as described on page 22.

The first byte of the *mode* argument determines the file type of the file:

- S\_IFCHR    Character special file
- S\_IFFIFO    FIFO special file
- S\_IFREG    Regular file
- S\_IFDIR    Directory file

The file permission bits of the new file are initialized with the remaining bits in *mode* and changed by the file creation mask of the process. For more information on these symbols, refer to “chmod() — Change the Mode of a File or Directory” on page 174.

*dev\_identifier* applies only to a character special file. It is ignored for the other file types. *dev\_identifier* contains a value representing the device major and device minor numbers. The major number is contained in the high-order 16 bits; it identifies a device driver supporting a class of devices, such as interactive terminals. The minor number is contained in the low-order 16 bits of *dev\_identifier*; it identifies a specific device within the class referred to by the device major number. With OS/390 UNIX services, the device major numbers are:

- 1            Master pseudoterminal
- 2            Slave pseudoterminal

3	/dev/tty
4	/dev/null
5	/dev/fdn
6	Sockets
7	OCSRTY
8	OCSADMIN
9	"/dev/console"

**Device major numbers 1,2 and 7:** The device minor numbers range between 0 and one less than the maximum number of pseudoterminal pairs defined by the installation.

**Device major numbers 3,4,6,8 and 9:** The device minor number is ignored.

**Device major number 5:** The device minor number value represents the file descriptor to be referred to. For example, device minor 0 refers to file descriptor 0.

When it completes successfully, mknod() marks for update the following fields of the file: `st_atime`, `st_ctime`, and `st_mtime`. It also marks for update the `st_ctime` and `st_mtime` fields of the directory that contains the new file.

### Returned Value

If successful, mknod() returns the value 0. If unsuccessful, it returns the value -1 and sets `errno` to one of the following:

EACCES	Search permission is denied on a component of <i>path</i> , or write permission is denied on the parent directory of the file to be created.
EEXIST	A file by that name already exists.
ELOOP	A loop exists in symbolic links. This error is issued if more than POSIX_SYMLLOOP (defined in the <code>limits.h</code> header file) symbolic links are detected in the resolution of <i>path</i> .
EMLINK	The link count of the parent directory has already reached the maximum defined for the system.
ENAMETOOLONG	<i>pathname</i> is longer than PATH_MAX characters, or some component of <i>pathname</i> is longer than NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH_MAX. The PATH_MAX and NAME_MAX values can be determined through <code>pathconf()</code> .
ENOENT	A component of <i>path</i> was not found, or no <i>path</i> was specified.
ENOSPC	The file system does not have enough space to contain a new directory, or the parent directory cannot be extended.
ENOTDIR	A component of <i>path</i> is not a directory
EROFS	The file named in <i>path</i> cannot be created, because it would reside on a read-only file system.

### Special Behavior for XPG4.2

The following are new values of `errno`:

**EPERM** The invoking process does not have appropriate privileges and the file type is not FIFO-special.

### Example CBC3BM18

```
/* CBC3BM18 */
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

#define master 0x00010000

main() {
    char fn[]="char.spec";

    if (mknod(fn, S_IFCHR|S_IRUSR|S_IWUSR, master|0x0001) != 0)
        perror("mknod() error");
    else if (unlink(fn) != 0)
        perror("unlink() error");
}
```

### Related Information

- “limits.h” on page 32
- “sys/stat.h” on page 48
- “mkfifo() — Make a FIFO Special File” on page 820
- “mkdir() — Make a Directory” on page 817
- “open() — Open a File” on page 872

## mkstemp() — Make a Unique Filename

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON)

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>
```

```
int mkstemp(char *template);
```

### General Description

The `mkstemp()` function replaces the contents of the string pointed to by *template* with a unique file name, and returns a file descriptor for the file open for reading and writing. The function thus prevents any possible race condition between testing whether the file exists and opening it for use. The string in *template* should look like a file name with six trailing 'X's; `mkstemp()` replaces each 'X' with a character from the portable file name character set. The characters are chosen such that the resulting name does not duplicate the name of an existing file.

### Returned Value

The `mkstemp()` function returns an open file descriptor. Otherwise -1 is returned if no suitable file could be created.

There are no errors defined for `mkstemp()`

### Related Information

- “`stdlib.h`” on page 45
- “`mktemp()` — Make a Unique Filename” on page 827



## mktemp() — Make a Unique Filename

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON)

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>
```

```
char * mktemp(char *template);
```

### General Description

The `mktemp()` function replaces the contents of the string pointed by *template* by a unique filename and returns *template*. The application must initialize *template* to be a filename with six trailing 'X's; `mktemp()` replaces each 'X' with a single byte character from the portable filename character set.

This function is supported only in a POSIX program. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

The `mktemp()` function returns a pointer to *template*. If a unique name cannot be created, *template* points to a null string.

There are no errors defined for `mktemp()`

### Related Information

- “`stdlib.h`” on page 45
- “`mkstemp()` — Make a Unique Filename” on page 826

## mktime() — Convert Local Time

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <time.h>
```

```
time_t mktime(struct tm *tmptr);
```

### General Description

Converts broken-down time, expressed as a local time, in the `tm` structure pointed to by `tmptr`, to calendar time. Calendar time is the number of seconds since epoch, which was at 00:00:00 January 1, 1970.

The `tm` structure is described in Table 27 on page 640.

The values of the structure members passed to `mktime()` are not restricted to the ranges described in `gmtime()` — Convert Time to Broken-Down UTC Time. The values of `tm_wday` and `tm_yday` are ignored and are assigned their correct values on return.

On the successful completion of the function, all the members of the structure pointed to by `time` are set to represent the specified time with their values forced into the ranges described in `gmtime()` — Convert Time to Broken-Down UTC Time. The values of `tm_wday` are set after the values of `tm_mon` and `tm_year` are determined.

If an application uses `localtime()` to determine local time, `localtime()` will determine if daylight savings applies (assuming DST info is available) and will correctly set the `tm_isdst` flag. If the application wants to determine seconds from Epoch corresponding to a `tm` structure returned by `localtime()`, it should *not* modify the `tm_isdst` flag set by `localtime()`.

If an application sets `tm_isdst = 0` before calling `mktime()`, it is asserting that daylight savings does *not* apply, regardless of the system DST start and end dates. Likewise, if the application has set a value for `tm_isdst` to be greater than 0, it is asserting that the time represented by the `tm` structure has been shifted for daylight savings. Therefore, `mktime()` unshifts the time in determining seconds since Epoch.

Setting `tm_isdst` to -1 tells the `mktime()` function to determine whether daylight savings time applies. If so, `mktime()` returns `tm_isdst` greater than 0. If not, it returns `tm_isdst` of 0 unless DST information is not available on the system, in which case `mktime()` returns `tm_isdst` of -1.

Your time zone may not be using a Daylight Savings Time, perhaps because the TZ environment variable does not specify a daylight savings time name or perhaps because DSTNAME is unspecified in the current LC\_TOD locale category. In such

a case, if you code `tm_isdst=1` and call `mktime()`, `(time-t)-1` is returned to indicate an error.

**Note** This function is sensitive to time zone information which is provided by:

- The TZ environmental variable when POSIX(ON) and TZ is correctly defined, or by the \_TZ environmental variable when POSIX(OFF) and \_TZ is correctly defined.
- The LC\_TOD category of the current locale if POSIX(OFF) or TZ is not defined.

The time zone external variables `tzname`, `timezone`, and `daylight` declarations remain feature test protected in `time.h`.

See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information about using POSIX support.

## Returned Value

Returns the calendar time corresponding to broken-down time, expressed as local time, using the `tm` structure, which is pointed to by *tm\_ptr*. If `mktime()` cannot convert the broken-down time to a calendar time, it returns `(time_t)-1` to indicate an error, such as time before January 1, 1970.

## Notes:

- The `ctime()`, `localtime()`, and `mktime()` functions now return coordinated universal time (UTC) unless customized locale information is made available, which includes setting the `timezone_name` variable.
- In POSIX you can supply the necessary information by using environment variables.
- In non-POSIX applications, you can supply customized locale information by setting time zone and daylight information in `LC_TOD`.
- By customizing the locale, you allow the time functions to preserve both time and date, correctly adjusting for daylight time on a given date.

## Example CBC3BM19

```
/* CBC3BM19
   This example prints the day of the week that is 40 days and 16 hours
   from the current date.
*/
#include <stdio.h>
#include <time.h>

char *wday[] = { "Sunday", "Monday", "Tuesday", "Wednesday",
                 "Thursday", "Friday", "Saturday" };

int main(void)
{
    time_t t1, t3;
    struct tm *t2;

    t1 = time(NULL);
    t2 = localtime(&t1);
    t2->tm_mday += 40;
    t2->tm_hour += 16;
    t3 = mktime(t2);
```

```
    printf("40 days and 16 hours from now, it will be a %s \n",  
          wday[t2 -> tm_wday]);  
}
```

### Output

40 days and 16 hours from now, it will be a Sunday

### Related Information

- “time.h” on page 51
- “asctime() — Convert Time to Character String” on page 106
- “ctime() — Convert Time to a Character String” on page 250
- “gmtime() — Convert Time to Broken-Down UTC Time” on page 640
- “localdtconv() — Date/Time Formatting Convention Inquiry” on page 754
- “localtime() — Convert Time and Correct for Local Time” on page 758
- “strftime() — Convert to Formatted Time” on page 1432
- “time() — Determine Current Time” on page 1570
- “tzset() — Set the Time Zone” on page 1637

## \_\_mlockall() — Lock the Address Space of a Process

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#define _OPEN_SOURCE_EXTENDED 2
#include <sys/mman.h>
```

```
int __mlockall(int flags);
```

### General Description

The function `__mlockall()` causes all of the pages mapped by the address space of a process to be memory resident until unlocked or until the process exits or execs another process image. The *flags* argument determines whether the pages are to be locked or unlocked.

#### Flags

#### Meaning

BPX\_SWAP      Lock the current pages mapped for this address space.  
 BPX\_NONSWAP    Unlock the current pages previously locked.

### Returned Value

Upon successful completion, `__mlockall()` returns 0. If the call was unsuccessful, no change is made to the memory state of the address space, a -1 is returned to the caller.

**Note:** This function will return a EINVAL with an `errno2()` of 09300be if the OE kernal is not available. APAR OW26631 should be installed to make the function available.

### Related Information

- “sys/mman.h” on page 47

## mmap() — Map Pages of Memory

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 OS/390 UNIX	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot, int flags,
           int fildes, off_t off);
```

### General Description

The `mmap()` function establishes a mapping between the process' address space at an address *pa* for *len* bytes and the file associated with the file descriptor *fildes* at offset *off* for *len* bytes. The value of *pa* is an unspecified function of the argument *addr* and values of *flags*, further described below. A successful `mmap()` call returns *pa* as its results. The address ranges covered by [*pa*, *pa* + *len*] and [*off*, *off* + *len*] must be legitimate for the possible (not necessarily current) address space of a process and the file, respectively.

If the size of the mapped file changes after the call to `mmap()`, the effect of references to portions of the mapped region that correspond to added or removed portions of the file is unspecified.

The `mmap()` function is supported for regular files. Support for any other type of file is unspecified.

The *prot* argument determines whether read, write, execute, or some combination of accesses are permitted to the pages being mapped. The protection options are defined in `<sys/mman.h>`:

```
PROT_READ    page can be read
PROT_WRITE   page can be written
PROT_EXEC    page can be executed
PROT_NONE    page can be accessed
```

Implementations need not enforce all combinations of access permissions. However, write shall only be permitted when `PROT_WRITE` has been set.

The *flags* argument provides other information about the handling of the mapped pages. The options are defined in `<sys/mman.h>`:

```
MAP_SHARED   Share changes
MAP_PRIVATE  Changes are private
MAP_FIXED    Interpret addr exactly
__MAP_MEGA   Map in megabyte increments
```

The MAP\_PRIVATE and MAP\_SHARED flags control the visibility of write references to the memory region. Exactly one of these flags must be specified. The mapping type is retained across a *fork*.

If MAP\_SHARED is set in *flags*, write references to the memory region by the calling process may change the file and are visible in all MAP\_SHARED mappings of the same portion of the file by any process.

If MAP\_PRIVATE is set in *flags*, write references to the memory region by the calling process do not change the file and are not visible to any process in other mappings of the same portion of the file.

All changes to the mapped data made by processes that have mapped the memory region using MAP\_SHARED are shared and are visible to all other processes that have mapped the same file-offset range.

When MAP\_FIXED is set in the *flags* argument, the implementation is informed that the value of *pa* must be *addr*, exactly. If MAP\_FIXED is set, *mmap()* may return (void\*)-1 and set *errno* to EINVAL. If a MAP\_FIXED request is successful, the mapping established by *mmap()* replaces any previous mappings for the process' pages in the range [*pa*, *pa + len*].

When MAP\_FIXED is not set, the implementation uses *addr* in an unspecified manner to arrive to *pa*. The *pa* so chosen will be an area of the address space which the implementation deems suitable for a mapping of *len* bytes to the file. All implementation interpret an *addr* value of 0 as granting the implementation complete freedom in selecting *pa*, subject to constraints described below. A nonzero value of *addr* is taken to be a suggestion of a process address near which the mapping should be placed. When the implementation selects a value for *pa*, it never places a mapping at address 0, nor does it replace any extant mapping, nor map into dynamic memory allocation areas.

The *off* argument is constrained to be aligned and sized according to the value returned by *sysconf()* when passed SC\_PAGESIZE or SC\_PAGE\_SIZE. When MAP\_FIXED is specified, the argument *addr* must also meet these constraints. The implementation performs mapping operations over whole pages. Thus, while the argument *len* need not meet a size or alignment constraint, the implementation will include, in any mapping operation, any partial page specified by the range [*pa*, *pa + len*].

The implementation always zero-fills any partial page at the end of a memory region. Further, the implementation never writes out any modified portions of the last page of a file that are beyond the end of the mapped portion of the file. If the mapping established by *mmap()* extends into pages beyond the page containing the last byte of the file, an application references to any of the pages in the mapping that are beyond the last page results in the delivery of a SIGBUS or SIGSEGV signal.

The *mmap()* function adds an extra reference to the file associated with the file descriptor *fd* which is not removed by a subsequent *close()* on that file descriptor. This reference is removed when there are not more mappings to the file.

The *st\_atime* field of the mapped file may be marked for update at any time between the *mmap()* call and the corresponding *munmap()* call. The initial read or

write reference to a mapped region will cause the file's `set_atime` field to be marked for update if it has not already been marked for update.

The `st_ctime` and `st_mtime` fields of a file that is mapped with `MAP_SHARED` and `PROT_WRITE`, will be marked for update at some point in the interval between a write reference to the mapped region and the next call to `msync()` with `MS_ASYNC` or `MS_SYNC` for that portion of the file by any process. If there is no such call, these fields may be marked for update at any time after a write reference if the underlying file is modified as a result.

If a memory mapped region is not unmapped before the process terminates, process termination will not automatically write out to disk any modified data in the mapped region. Modified private data in a `MAP_PRIVATE` region will be discarded. If the map region is `MAP_SHARED`, the modified data will continue to reside in the cache (if the same file-offset range is being shared) and may ultimately be written out to disk by another process via the `msync()` service. However, if no other processes map the same file-offset range as `MAP_SHARED`, the modified data is discarded.

There may be implementation-dependent limits on the number of memory regions that can be mapped (per process or per system). If such a limit is imposed, whether the number of memory regions that can be mapped by a process is decreased by the use of `shmat()` is implementation dependant.

Specification of the `__MAP_MEGA` option results in the system allocating storage to map the file in megabyte increments. This option should only be used for large files. Any file over half a megabyte in size will likely achieve better performance by using this option. When using this option, `mmaps` and `munmaps` are in megabyte ranges on megabyte boundaries.

When `__MAP_MEGA` is specified, all changes to the mapped data are shared. Modifications to the mapped data are visible to all other processes that map the same file-offset range. That is, `__MAP_MEGA` behaves much like `MAP_SHARED`. `__MAP_MEGA` is mutually exclusive with `MAP_PRIVATE` and `MAP_SHARED`. Specification of `__MAP_MEGA` with either `MAP_PRIVATE` or `MAP_SHARED` will result in the request failing with `errno` set to `EINVAL`.

The `__MAP_MEGA` option may be specified with `MAP_FIXED`.

**Map\_address parameter:** If the map address is not zero and `__MAP_MEGA` has been specified, then for non `MAP_FIXED` requests, the kernel will attempt to create the mapping at the `map_address`, truncated to the nearest megabyte boundary. If unsuccessful, it will proceed as if a `map_address` of zero were specified. For `MAP_FIXED` requests, the value of `map_address` must be multiples of the segment size (megabyte multiples). If not, the `mmap` request fails with `errno` set to `EINVAL`.

**Map\_length parameter:** When `__MAP_MEGA` is specified, mapping operations are performed over whole segments (megabyte chunks). If the length is not a multiple of the segment size, the entire trailing portion of the last segment will also be mapped into the user storage.

**File\_Descriptor:** The file descriptor identifies the file being mapped. If an attempt is made to map a file that is already mapped but was mapped with a different specification of `__MAP_MEGA`, then the current request fails with `errno` set to `EINVAL-MMapTypeMismatch`. That is, at any point in time a file may be mapped



with the `__MAP_MEGA` option or without the `__MAP_MEGA` option but not both ways at the same time.

For a `__MAP_MEGA` mapping, if this is the first map to the file represented by the specified file descriptor then whether the file was opened for read or for write will determine what protection options may be specified for the file by this `mmap` and any future `mmaps` and `mprotects`, by this or any other process mapping to the same file. If the file was opened for read but not write then only `PROT_READ`, `PROT_EXEC` or `PROT_NONE` will be allowed. If the file was opened for write, then any of the protection options will be allowed.

**Protect\_options:** The specification made for `Protect_options` has a global effect when the file is mapped with the `__MAP_MEGA` option. The `Protect_option` specified immediately effects all processes currently mapped to the same file-offset range.

## Returned Value

Upon successful completion, `mmap()` returns the address at which the mapping was placed(*pa*). Otherwise, it returns a value of -1 and sets `errno` to indicate the error.

The `mmap()` function will fail if:

- |        |   |
|--------|---|
| EBADF  | The <i>fil</i> des argument is not a valid open file descriptor.  |
| EACCES | The <i>fil</i> des argument is not open for read, regardless of the protection specified, or <i>fil</i> des is not open for write and <code>PROT_WRITE</code> was specified for a <code>MAP_SHARED</code> type mapping.   |
| ENXIO  | Address in the range [ <i>off</i> , <i>off</i> + <i>len</i> ] are invalid for <i>fil</i> des  |
| EINVAL | The <i>addr</i> argument (if <code>MAP_FIXED</code> was specified) or <i>off</i> is not a multiple of the page size as returned by <code>sysconf()</code> , or are considered invalid by the implementation.  |
| EINVAL | The value of <i>flags</i> is invalid (neither <code>MAP_PRIVATE</code> nor <code>MAP_SHARED</code> is set).   |
| EMFILE | The number of mapped regions would exceed an implementation-dependent limit (per process or per system).  |
| ENODEV | The <i>fil</i> des argument refers to a file whose type is not supported by <code>mmap()</code> .   |
| ENOMEM | <code>MAP_FIXED</code> was specified, and the range [ <i>addr</i> , <i>addr</i> + range [ <i>addr</i> , <i>addr</i> + <i>len</i> ], rounding <i>len</i> ] exceeds that allowed for the address space for a process; or if <code>MAP_FIXED</code> was not specified and there is insufficient room in the address space to effect the mapping. |

## Related Information

- “exec Functions” on page 322
- “fcntl() — Control Open File Descriptors” on page 350
- “fork() — Create a New Process” on page 422
- “lockf() — Record Locking on Files” on page 760
- “msync() — Synchronize Memory with Physical Storage” on page 856
- “munmap() — Unmap Pages of Memory” on page 858
- “mprotect() — Set Protection of Memory Mapping” on page 841
- “shmat() — Shared Memory Attach Operation” on page 1285

- “sysconf() — Determine System Configuration Options” on page 1483

## modf() — Extract Fractional and Integral Parts of Floating-Point Value

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double modf(double x, double *intptr);
```

### General Description

Breaks down the floating-point value *x* into fractional and integral parts. These parts are stored as double, in the object pointed to by *intptr*. Both the fractional and integral parts are given the same sign as *x*.

### Returned Value

Returns the signed fractional portion of *x*.

### Example

#### CBC3BM20

```
/* CBC3BM20
   This example breaks the floating-point number -14.876 into its
   fractional and integral components.
*/
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, y, d;

    x = -14.876;
    y = modf(x, &d);

    printf("x = %lf\n", x);
    printf("Integral part = %lf\n", d);
    printf("Fractional part = %lf\n", y);
}
```

### Output

```
x = -14.876000
Integral part = -14.000000
Fractional part = -0.876000
```

### Related Information

- “math.h” on page 35
- “frexp() — Extract Mantissa and Exponent of the Floating-Point Value” on page 462
- “ldexp() — Multiply by a Power of Two” on page 738

## mount() — Make a File System Available

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 4.3

### Format

```
#include <sys/stat.h>
```

```
int mount(const char *path, char *filesystem,
          char *filesystype, mtm_t mtm,
          int parmlen, char *parm);
```

### General Description

Adds a file system to the hierarchical file system. The same file system cannot be mounted at more than one place in the hierarchical file system.

In order to mount a file system, the caller must be an authorized program, or must be running for a user with appropriate privileges.

<i>path</i>	The mount point directory that the file system is to be mounted to.
<i>filesystem</i>	The name of the file system to be mounted; it must be unique within the system. For a hierarchical file system (HFS) data set, this is a 1-to-44-character MVS data set name specified as all uppercase letters.  This name is terminated with null characters.
<i>filesystype</i>	The name for the file system that will perform the mount. This 8-character name must match the TYPE operand on a FILESYSTYPE statement in the BPXPRMxx parmlib member for the file system.
<i>mtm</i>	A flag field that specifies the mount mode and additional mount options:  <div style="margin-left: 40px;"> MTM_RDONLY Mount the file system as a read-only file system.   MTM_RDWR Mount the file system as a read/write file system.   MTM_NOSUID The SETUID and SETGID mode flags will be ignored for programs that reside in this file system.   MTM_SYNCHONLY The mount must be completed synchronously or fail if it cannot. </div>
<i>parmlen</i>	Length of the <i>parm</i> argument. The maximum length is 1024 characters. For a hierarchical file system (HFS) data set, this is not specified.

*parm* A parameter passed to the physical file system that performs the mount. This parameter may not be required. The form and content of the *parm* are determined by the physical file system. A hierarchical file system (HFS) data set does *not* require a *parm*.

## Returned Value

If successful, `mount()` returns the value 0. If the `mount()` is proceeding asynchronously, it returns the value 1. If unsuccessful, it returns the value -1 and sets `errno` to one of the following:

EBUSY	The specified file system is unavailable.
EINVAL	A parameter was incorrectly specified. Verify <i>filesystype</i> and <i>mtm</i> . Another possible reason for this error is that the mount point is the root of a file system or that the file system is already mounted.
EIO	An I/O error occurred.
ELOOP	A loop exists in symbolic links. This error is issued if more than POSIX_SYMLLOOP (defined in the <code>limits.h</code> header file) symbolic links are detected in the resolution of <i>pathname</i> .
ENOENT	The mount point does not exist.
ENOMEM	There is not enough storage available to save the information required for this file system.
ENOTDIR	The mount point is not a directory.
EPERM	Superuser authority is required to issue a mount.

## Example CBC3BM21

```

/* CBC3BM21
   This example adds a file system to the hierarchical file system.
   */
#include <sys/stat.h>
#include <stdio.h>
#include <unistd.h>

main() {
    char mount_point[]="/new_fs";
    char HFS[]="POSIX.NEW.HFS";
    char filesystype[9]="HFS    ";

    puts("before mount()");
    system("df -Pk");
    if (mount(mount_point, HFS, filesystype, MTM_RDWR, 0, NULL) != 0)
        perror("mount() error");
    else {
        puts("After mount()");
        system("df -Pk");
        if (umount(HFS, MTM_UMOUNT) != 0)
            perror("umount() error");
    }
}

```

## Output

## mount

before mount()

Filesystem	1024-blocks	Used	Available	Capacity	Mounted on
POSIX.ROOT.FS	9600	8660	940	90%	/

After mount()

Filesystem	1024-blocks	Used	Available	Capacity	Mounted on
POSIX.NEW.HFS	200	20	180	10%	/new_fs
POSIX.ROOT.FS	9600	8660	940	90%	/

### Related Information

- “limits.h” on page 32
- “sys/stat.h” on page 48
- “umount() — Remove a Virtual File System” on page 1649
- “w\_getmntent() — Get Information on Mounted File Systems” on page 1756
- “w\_statfs() — Get the File System Status” on page 1789

## mprotect() — Set Protection of Memory Mapping

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 OS/390 UNIX	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/mman.h>
int mprotect(void *addr, size_t len, int prot);
```

### General Description

The `mprotect()` function changes the access protections on the mappings specified by the *len* up to the next multiple of the page size as returned by `sysconf()`, to be that specified by *prot*. Legitimate values for *prot* are the same as those permitted for `mprotect()` and are defined in `<sys/mman.h>`:

`PROT_READ` page can be read  
`PROT_WRITE` page can be written  
`PROT_EXEC` page can be executed  
`PROT_NONE` page cannot be accessed

The range provided by the `Map_address` and `Map_length` may span regular maps as well as `__MAP_MEGA` maps. `Mprotect` effects `__MAP_MEGA` maps very differently than regular maps. The difference is in the scope of the change. When a change is made to a `__MAP_MEGA` map, the change effects all processes which are currently mapped to the same file-offset range represented by the pages within the provided range. For example, changing a file-offset range (storage pages) that is currently in use with a protection of write to a protection of read, makes the file-offset range read for all processes, not just the current one. In other words, the changes are global. On the other hand, changes to regular maps effect only the process that issues `mprotect`.

When `mprotect()` fails for reasons other than `EINVAL`, the protection on some of the pages in the range `[addr, addr + len)` may have been changed.

### Returned Value

Upon successful completion, `mprotect()` returns 0. Otherwise, it returns -1 and sets `errno` to indicate the error.

The `mprotect()` function will fail if:

- EACCES** The *prot* argument specifies a protection that violates the access permission the process has to the underlying memory object.
- EINVAL** The *addr* argument is not a multiple of the page size as returned by `sysconf()`.
- ENOMEM** Addresses in the range `[addr, addr + len)` are invalid for the address space of a process, or specify one or more pages

The mprotect() function will fail if:

**EAGAIN** The *prot* argument specifies PROT\_WRITE over a MAP\_PRIVATE mapping and there are insufficient memory resources to reserve for locking the private page.

### Related Information

- “mmap() — Map Pages of Memory” on page 832
- “sysconf() — Determine System Configuration Options” on page 1483



# mrnd48() — Pseudo-random Number Generator

## Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

## Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>
```

```
long int mrnd48(void);
```

## General Description

The drand48(), erand48(), jrand48(), lrand48(), mrnd48() and nrand48() functions generate uniformly distributed pseudorandom numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The functions drand48() and erand48() return non-negative, double-precision, floating-point values, uniformly distributed over the interval [0.0,1.0).

The functions lrand48() and nrand48() return non-negative, long integers, uniformly distributed over the interval [0,2\*\*31).

The functions mrnd48() and jrand48() return signed long integers, uniformly distributed over the interval [-2\*\*31,2\*\*31).

The mrnd48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(i), according to the linear congruential formula:

$$X(n+1) = (aX(n) + c) \bmod (2^{**}48) \quad n \geq 0$$

The initial values of X, a, and c are:

```
X(0) = 1
a    = 5deece66d (base 16)
c    = b          (base 16)
```

C/370 provides storage to save the most recent 48-bit integer value of the sequence, X(i). This storage is shared by the drand48(), lrand48() and mrnd48() functions. The value, X(n), in this storage may be reinitialized by calling the lcong48(), seed48() or srand48() function. Likewise, the values of a and c, may be changed by calling the lcong48() function. Thereafter, whenever the seed48() or srand48() function is called to change X(n), the initial values of a and c are also reestablished.

## Special Behavior for OS/390 UNIX Services

You can make the mrnd48() function and other functions in the drand48 family thread specific by setting the environment variable \_RAND48 to the value THREAD before calling any function in the drand48 family.

If you do not request thread specific behavior for the drand48 family, C/370 serializes access to the storage for X(n), a and c by functions in the drand48 family when they are called by a multithreaded application.

If thread specific behavior is requested, and the `mrnd48()` function is called from thread `t`, the `mrnd48()` function generates the next 48-bit integer value in a sequence of 48-bit integer values,  $X(t,i)$ , for the thread `t`. The sequence of values for a thread is generated according to the linear congruential formula:

$$X(t,n+1) = (a(t)X(t,n) + c(t)) \bmod (2^{**}48) \quad n \geq 0$$

The initial values of  $X(t)$ ,  $a(t)$  and  $c(t)$  for the thread `t` are:

```
X(t,0) = 1
a(t)   = 5deece66d (base 16)
c(t)   = b         (base 16)
```

C/370 provides storage which is specific to the thread `t` to save the most recent 48-bit integer value of the sequence,  $X(t,i)$ , generated by the `drand48()`, `lrand48()` or `mrnd48()` function. The value,  $X(t,n)$ , in this storage may be reinitialized by calling the `lcong48()`, `seed48()` or `srand48()` function from the thread `t`. Likewise, the values of  $a(t)$  and  $c(t)$  for thread `t` may be changed by calling the `lcong48()` function from the thread. Thereafter, whenever the `seed48()` or `srand48()` function is called from the thread `t` to change  $X(t,n)$ , the initial values of  $a(t)$  and  $c(t)$  are also reestablished.

## Returned Value

The `mrnd48()` function transforms the generated 48-bit value,  $X(n+1)$ , to a signed long integer value on the interval  $[-2^{**}31, 2^{**}31)$  and returns this transformed value.

## Special Behavior for OS/390 UNIX Services

If thread specific behavior is requested for the `drand48` family and the `mrnd48()` function is called on thread `t`, the `mrnd48()` function transforms the generated 48-bit value,  $X(t,n+1)$ , to a signed long integer value on the interval  $[-2^{**}31, 2^{**}31)$  and returns this transformed value.

## Related Information

- “`stdlib.h`” on page 45
- “`drand48()` — Pseudo-random Number Generator” on page 286
- “`erand48()` — Pseudo-random Number Generator” on page 314
- “`jrand48()` — Pseudo-random Number Generator” on page 724
- “`lcong48()` — Pseudo-random Number Initializer” on page 736
- “`lrand48()` — Pseudo-random Number Generator” on page 773
- “`nrnd48()` — Pseudo-random Number Generator” on page 868
- “`seed48()` — Pseudo-random Number Initializer” on page 1156
- “`srand48()` — Pseudo-random Number Initializer” on page 1401

# msgctl() — Message Control Operations

## Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 5.2.2

## Format

```
#define _XOPEN_SOURCE
#include <sys/msg.h>

int msgctl(int msgid, int cmd, struct msqid_ds *buf);
```

## General Description

The msgctl() function provides message control operations as specified by *cmd*. The following values for *cmd*, and the message control operations they specify, are (These symbolic constants are defined by the <sys/ipc.h> header):

- IPC\_STAT    Place the current value of each member of the msqid\_ds data structure associated with *msgid* into the structure pointed to by *buf*. The contents of this structure are defined in <sys/msg.h>. This command requires read permission.
- IPC\_SET    Set the value of the following members of the msqid\_ds data structure associated with *msgid* to the corresponding value found in the structure pointed to by *buf*.
  - msg\_perm.uid
  - msg\_perm.gid
  - msg\_perm.mode
  - msg\_qbytesIPC\_SET can only be executed by a process with the appropriate privileges or that has an effective user ID equal to the value of msg\_perm.cuid or msg\_perm.uid in the msqid\_ds data structure associated with *msgid*. Only a process with appropriate privileges can raise the value of msg\_qbytes.
- IPC\_RMID    Remove the message queue identifier specified by *msgid* from the system and destroy the message queue and msqid\_ds data structure associated with it. **IPC\_RMID** can only be executed by a process with appropriate privileges or one that has an effective user ID equal to the value of msg\_perm.cuid or msg\_perm.uid in the msqid\_ds data structure associated with *msgid*.

## Returned Value

If successful, msgctl() returns zero.

If unsuccessful, msgctl() returns -1, and returns the error value in errno. The following are the possible values of errno:

- EACCES    The argument *cmd* is **IPC\_STAT** and the calling process does not have read permission.

- EINVAL**      The value of *msgid* is not a valid message queue identifier, or the value of *cmd* is not a valid command.
- EPERM**      The argument *cmd* is **IPC\_RMID** or **IPC\_SET** and the effective user ID of the calling process is not equal to that of a process with appropriate privileges and it is not equal to the value of *msg\_perm.cuid* or *msg\_perm.uid* in the data structure associated with *msgid*.
- Or the argument *cmd* is **IPC\_SET**, an attempt is being made to increase the value of *msg\_qbytes*, and the effective user ID of the calling process does not have appropriate privileges.

**Related Information**

- “sys/ipc.h” on page 47
- “sys/msg.h” on page 48
- “msgget() — Get Message Queue” on page 847
- “msgrcv() — Message Receive Operation” on page 850
- “msgsnd() — Message Send Operations” on page 852
- “msgxrcv() — Extended Message Receive Operation” on page 854

## msgget() — Get Message Queue

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE
#include <sys/msg.h>
```

```
int msgget(key_t key, int msgflg);
```

### General Description

The `msgget()` function returns the message queue identifier associated with the argument *key*.

A message queue identifier, associated message queue and data structure (see `<sys/msg.h>`) are created for the argument *key* if one of the following is true:

- The argument *key* is equal to **IPC\_PRIVATE**
- The argument *key* does not already have a message queue identifier associated with it, and the flag **IPC\_CREAT** is on in *msgflg*.

Valid values for the argument *msgflg* include any combination of the following constants defined in `<sys/ipc.h>` and `<sys/modes.h>`:

#### IPC\_CREAT

Create a message queue if the *key* specified does not already have an associated ID. **IPC\_CREAT** is ignored when **IPC\_PRIVATE** is specified

**IPC\_EXCL** Causes the `msgget()` function to fail if the *key* specified has an associated ID. **IPC\_EXCL** is ignored when **IPC\_CREAT** is not specified or **IPC\_PRIVATE** is specified

#### IPC\_RCVTYPEPID

Creates a message queue that can only be read from `msgrcv()` when `Message_Type` is the process ID of the invoker. This restriction does not apply if the `msgrcv()` invoker has the same effective UID as the message queue creator.

#### IPC\_SNDTYPEPID

Creates a message queue that can only be written to `msgsnd()` when `MSG_Type` is the process ID of the invoker. This restriction does not apply if the `msgsnd()` invoker has the same effective UID as the message queue creator.

**S\_IRUSR** Permits read access when the effective user ID of the caller matches either `msg_perm.cuid` or `msg_perm.uid`

**S\_IWUSR** Permits write access when the effective user ID of the caller matches either `msg_perm.cuid` or `msg_perm.uid`

S_IRGRP	Permits read access when the effective group ID of the caller matches either <code>msg_perm.cgid</code> or <code>msg_perm.gid</code>
S_IWGRP	Permits write access when the effective group ID of the caller matches either <code>msg_perm.cgid</code> or <code>msg_perm.gid</code>
S_IROTH	Permits other read access
S_IWOTH	Permits other write access

When a message set associated with argument *key* already exists, setting **IPC\_EXCL** and **IPC\_CREAT** in argument *msgflg* will force `msgget()` to fail.

Upon creation, the `msg_ds` data structure associated with the new message queue identifier is initialized as follows:

- The fields `msg_perm.cuid`, `msg_perm.uid`, `msg_perm.cgid`, and `msg_perm.gid` are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order 9 bits of `msg_perm.mode` are set equal to the low-order 9 bits of *msgflg*.
- The fields `msg_qnum`, `msg_lspid`, `msg_lrpipd`, `msg_stime`, and `msg_rtime` are set to zero.
- The field `msg_ctime` is set equal to the current time.
- The field `msg_qbytes` is set equal to the system limit.

### Usage Note

In a client/server environment, two message queues can be used. One inbound to the server created with **IPC\_SNDTYPEPID** and the other outbound from the server created with **IPC\_RCVTYPEPID**. This arrangement guarantees that the server knows the process ID of the client and the client is the only process that receives the server's returned message. The server may invoke `msgrcv()` with `PID=0` to see if any messages belong to process IDs that have gone away.

### Returned Value

If successful, `msgget()` returns a non-negative integer, namely a message queue identifier.

If unsuccessful, `msgget()` returns `-1`, and returns the error value in `errno`. The following are the possible values of `errno`:

EACCES	A message queue identifier exists for the argument <i>key</i> , but access permission as specified by the low order 9 bits of <i>msgflg</i> could not be granted
EEXIST	A message queue identifier exists for the argument <i>key</i> , but both <b>IPC_CREAT</b> and <b>IPC_EXCL</b> are specified in <i>msgflg</i>
EINVAL	The value of argument <i>msgflg</i> is not currently supported
ENOENT	A message queue identifier does not exist for the argument <i>key</i> and <b>IPC_CREAT</b> is not specified.
ENOSPC	A message queue identifier is to be created but the system-imposed limit on the maximum number of allowed message queue identifiers system-wide would be exceeded.

When *msgflg* equals 0, the following applies:

- If a message queue identifier has already been created with *key* earlier, and the calling process of this `msgget()` has read and/or write permissions to it, then `msgget()` returns the associated message queue identifier.
- If a message queue identifier has already been created with *key* earlier, and the calling process of this `msgget()` does not have read and/or write permissions to it, then `msgget()` returns -1 and sets `errno` to `EACCES`.
- If a message queue identifier has not been created with *key* earlier, then `msgget()` returns -1 and sets `errno` to `ENOENT`.

### Related Information

- “`sys/ipc.h`” on page 47
- “`sys/msg.h`” on page 48
- “`sys/types.h`” on page 49
- “`ftok()` — Generate an Interprocess Communication (IPC) key” on page 488
- “`msgctl()` — Message Control Operations” on page 845
- “`msgrcv()` — Message Receive Operation” on page 850
- “`msgsnd()` — Message Send Operations” on page 852
- “`msgxrcv()` — Extended Message Receive Operation” on page 854

## msgrcv() — Message Receive Operation

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE
#include <sys/msg.h>
```

```
int msgrcv(int msgid, void *msgp, size_t msgsz, long int msgtyp, int msgflg);
```

### General Description

The `msgrcv()` function reads a message from the queue associated with the message queue identifier specified by `msgid` and places it in the user-defined buffer pointed to by `msgp`.

The argument, `msgp`, points to a user-defined buffer that must contain first a field of type `long int` that will specify the type of the message, and then a data portion that will hold the data bytes of the message. The structure below is an example of what this user-defined buffer should look like:

```
struct message
{
    long int  mtype;    Message type
    int      mtext[n]; Message text
}
```

The structure member, `mtype`, is the received message's type as specified by the sending process. The structure member, `mtext`, is the text of the message.

The argument, `msgsz`, specifies the size in bytes of `mtext`. The received message is truncated to `msgsz` bytes if it is larger than `msgsz` and the **MSG\_NOERROR** flag was specified in the argument, `msgflg`. The truncated portion of the message is lost and no indication of the truncation is given to the calling process.

The argument, `msgtyp`, specifies the type of message requested, as follows:

- If `msgtyp` is equal to zero, the first message on the queue is received.
- If `msgtyp` is greater than zero, the first message of type, `msgtyp`, is received.
- If `msgtyp` is less than zero, the first message of the lowest type that is less than or equal to the absolute value of `msgtyp` is received.

The argument, `msgflg`, specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

- If the **IPC\_NOWAIT** flag is on in `msgflg`, the calling process will return immediately with a return value of `-1` and `errno` set to **ENOMSG**.
- If the **IPC\_NOWAIT** flag is off in `msgflg` the calling process will suspend execution until one of the following occurs:
  - A message of the desired type is placed on the queue.



- The message queue identifier, *msgid*, is removed from the system; when this occurs, *errno* is set to **EIDRM** and a value of **-1** is returned.
- The calling process receives a signal that is to be caught; in this case a message is not received and the calling process resumes execution. A value of **-1** is returned and *errno* is set to **EINTR**.

Upon successful completion, the following actions are taken with respect to the data structure, *msgid\_ds*, associated with *msgid*:

1. *msg\_qnum* is decremented by 1.
2. *msg\_lrp\_id* is set equal to the process ID of the calling process.
3. *msg\_rtime* is set equal to the current time.

### Returned Value

If successful, *msgrcv()* returns a value equal to the number of bytes actually placed into the *mtext* field of the user-defined buffer pointed to by *msgp*. A value of zero indicates that only the *mtype* field was received from the message queue.

If unsuccessful, *msgrcv()* returns **-1**, and returns the error value in *errno*. The following are the possible values of *errno*:

- |        |  |
|--------|--|
| E2BIG  | The value of <i>mtext</i> is greater than <i>msgsz</i> and the flag <b>MSG_NOERROR</b> was not specified.  |
| EACCES | The calling process does not have read permission to the message queue associated with the message queue identifier <i>msgid</i> or the message queue was built with <b>IPC_RCVTYPEPID</b> and the <i>Message_Type</i> was other than the invokers process ID ( <b>JRTypeNotPID</b> ). |
| EIDRM  | The message queue identifier, <i>msgid</i> , has been removed from the system while the caller of <i>msgrcv()</i> was waiting.   |
| EINTR  | The function <i>msgrcv()</i> was interrupted by a signal before a message could be received.   |
| EINVAL | The value of argument <i>msgid</i> is not a valid message queue identifier or the value of <i>msgsz</i> is less than zero.   |
| ENOMSG | The flag <b>IPC_NOWAIT</b> was specified and the message queue does not contain a message of the desired type.   |

### Related Information

- “*sys/ipc.h*” on page 47
- “*sys/msg.h*” on page 48
- “*msgctl()* — Message Control Operations” on page 845
- “*msgget()* — Get Message Queue” on page 847
- “*msgsnd()* — Message Send Operations” on page 852
- “*msgxrcv()* — Extended Message Receive Operation” on page 854

## msgsnd() — Message Send Operations

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE
#include <sys/msg.h>
```

```
int msgsnd(int msgid, const void *msgp, size_t msgsz, int msgflg);
```

### General Description

The `msgsnd()` function is used to send a message to the queue associated with the message queue identifier specified by *msgid*.

The argument, *msgp*, points to a user-defined buffer that must contain first a field of type long int that will specify the type of the message, and then a data portion that will hold the data bytes of the message. The structure below is an example of what this user-defined buffer should look like:

```
struct message
{
    long int  mtype;    Message type
    int       mtext[n]; Message text
}
```

The structure member, *mtype*, must be a nonzero positive value that can be used by the receiving process for message selection. The structure member, *mtext*, is any text of length, *msgsz*, bytes.

The argument, *msgsz*, specifies the size in bytes of *mtext*. When only *mtype* is to be sent with no *mtext*, *msgsz* is set to zero. The argument can range from zero to a system-imposed maximum or the maximum number of bytes allowed in the message queue.

The argument, *msgflg*, specifies the action to be taken if one or more of the following are true:

- Placing the message on the message queue would cause the current number of bytes on the message queue (*msg\_cbytes*) to exceed the maximum number of bytes allowed on this queue, as specified in *msg\_qbytes*.
- The total number of messages on the queue is equal to the system-imposed limit.

These actions are as follows:

- If the **IPC\_NOWAIT** flag is on in *msgflg*, the message not be sent and the calling process will return immediately. `msgsnd()` will return a -1 and set *errno* to *EAGAIN*.
- If the **IPC\_NOWAIT** flag is off in *msgflg*, the calling process will suspend execution until one of the following occurs:

1. The condition responsible for the suspension no longer exists, in which case the message is sent.
2. The message queue identifier, *msgid*, is removed from the system; when this occurs, *errno* is set to *EIDRM* and a value of -1 is returned.
3. The calling process receives a signal that is to be caught; in this case a message is not sent and the calling process resumes execution. A value of -1 is returned and error is set to *EINTR*.

Upon successful completion, the following actions are taken with respect to the data structure, *msgid\_ds*, associated with *msgid*:

1. *msg\_qnum* is incremented by 1.
2. *msg\_lspid* is set equal to the process ID of the calling process.
3. *msg\_stime* is set equal to the current time.

### Returned Value

If successful, *msgsnd()* returns a value of 0 (zero).

If unsuccessful, no message is sent and *msgsnd()* returns - 1, and returns the error value in *errno*. The following are the possible values of *errno*:

EACCES	The calling process does not have write permission to the message queue associated with the message queue identifier <i>msgid</i> or the message queue was built with <i>IPC_SNDTYPEPID</i> and the <i>MSG_TYPE</i> was other than the invokers process ID ( <i>JRTypeNotPID</i> ).
EAGAIN	The message cannot be sent for one of the reasons cited above and <i>IPC_NOWAIT</i> was specified.
EIDRM	The message queue identifier, <i>msgid</i> , has been removed from the system while the caller of <i>msgsnd()</i> was waiting.
EINTR	The function <i>msgsnd()</i> was interrupted by a signal before a message could be sent.
EINVAL	The value of argument, <i>msgid</i> , is not a valid message queue identifier, or the value of <i>mtype</i> is less than 1; or the value of <i>msgsz</i> is less than zero or greater than the system-imposed limit.
ENOMEM	Not enough system storage exists to complete the <i>msgsnd()</i> function.

### Related Information

- “*sys/ipc.h*” on page 47
- “*sys/msg.h*” on page 48
- “*msgctl()* — Message Control Operations” on page 845
- “*msgget()* — Get Message Queue” on page 847
- “*msgrcv()* — Message Receive Operation” on page 850
- “*msgxrcv()* — Extended Message Receive Operation” on page 854

## msgxrcv() — Extended Message Receive Operation

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 5.2.2

### Format

```
#define _OPEN_SYS_IPC_EXTENSIONS
#include <sys/msg.h>
```

```
int msgxrcv(int msgid, void *msgp, size_t msgsz, long int msgtyp, int msgflg);
```

**Note:** To expose the msgxrcv() name, the feature test macro **\_OPEN\_SYS\_IPC\_EXTENSIONS** should be defined. Otherwise, the function's name is **\_\_msgxrcv()**.

### General Description

The msgxrcv() function reads an extended message from the queue associated with the message queue identifier specified by *msgid* and places it in the user-defined buffer pointed to by *msgp*.

The argument, *msgp*, points to a user-defined buffer where the extended message will be received. This buffer must be defined by a data structure of the following format: .

```
struct msgxbuf {
    time_t    mtime;    Time and date message was sent
    uid_t     muid;     Sender's effective user ID
    gid_t     mgid;     Sender's effective group ID
    pid_t     mpid;     Sender's process ID
    long int  mtype;    Message type
    int       mtext[n]; Message text
}
```

The structure member, *mtype*, is the received message's type as specified by the sending process. The structure member, *mtext*, is the text of the message.

The argument, *msgsz*, specifies the size in bytes of *mtext*. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and the **MSG\_NOERROR** flag was specified in the argument, *msgflg*. The truncated portion of the message is lost and no indication of the truncation is given to the calling process.

The argument, *msgtyp*, specifies the type of message requested, as follows:

- If *msgtyp* is equal to zero, the first message on the queue is received.
- If *msgtyp* is greater than zero, the first message of type, *msgtyp*, is received.
- If *msgtyp* is less than zero, the first message of the lowest type that is less than or equal to the absolute value of *msgtyp* is received.

The argument, *msgflg*, specifies the action to be taken if a message of the desired type is not on the queue. These are as follows:

- If the **IPC\_NOWAIT** flag is on in *msgflg*, the calling process will return immediately with a return value of **-1** and **errno** set to **ENOMSG**.

- If the **IPC\_NOWAIT** flag is off in *msgflg* the calling process will suspend execution until one of the following occurs:
  - A message of the desired type is placed on the queue.
  - The message queue identifier, *msgid*, is removed from the system; when this occurs, *errno* is set to **EIDRM** and a value of **-1** is returned.
  - The calling process receives a signal that is to be caught; in this case a message is not received and the calling process resumes execution. A value of **-1** is returned and *errno* is set to **EINTR**.

Upon successful completion, the following actions are taken with respect to the data structure, *msgid\_ds*, associated with *msgid*:

1. *msg\_qnum* is decremented by 1.
2. *msg\_lrpid* is set equal to the process ID of the calling process.
3. *msg\_rtime* is set equal to the current time.

## Returned Value

If successful, *msgxrcv()* returns a value equal to the number of bytes actually placed into the *mtext* field of the user-defined buffer pointed to by *msgp*. A value of zero indicates that only the *mtype* field was received from the message queue.

If unsuccessful, *msgxrcv()* returns **-1**, and returns the error value in *errno*. The following are the possible values of *errno*:

- |               |  |
|---------------|--|
| <b>E2BIG</b>  | The value of <i>mtext</i> is greater than <i>msgsz</i> and the flag <b>MSG_NOERROR</b> was not specified.                          |
| <b>EACCES</b> | The calling process does not have read permission to the message queue associated with the message queue identifier <i>msgid</i> . |
| <b>EIDRM</b>  | The message queue identifier, <i>msgid</i> , has been removed from the system while the caller of <i>msgxrcv()</i> was waiting.    |
| <b>EINTR</b>  | The function <i>msgxrcv()</i> was interrupted by a signal before a message could be received.                                      |
| <b>EINVAL</b> | The value of argument <i>msgid</i> is not a valid message queue identifier or the value of <i>msgsz</i> is less than zero.         |
| <b>ENOMSG</b> | The flag <b>IPC_NOWAIT</b> was specified and the message queue does not contain a message of the desired type.                     |

## Related Information

- “sys/ipc.h” on page 47
- “sys/msg.h” on page 48
- “msgctl() — Message Control Operations” on page 845
- “msgget() — Get Message Queue” on page 847
- “msgrcv() — Message Receive Operation” on page 850
- “msgsnd() — Message Send Operations” on page 852

## msync() — Synchronize Memory with Physical Storage

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/mman.h>
int msync(void *addr, size_t len, int flags);
```

### General Description

The `msync()` function writes all modified copies of pages over the range [*addr*, *addr* + *len*) to the underlying hardware, or invalidates any copies so that further references to the pages will be obtained by the system from their permanent storage locations.

The *flags* argument is:

`MS_ASYNC` (perform asynchronous writes) or `MS_SYNC` (perform synchronous writes)

`MS_INVALIDATE` invalidate mappings

The function synchronizes the file contents to match the current contents to the memory region.

- All write references to the memory region made prior to the call are visible by subsequent read operations on the file.
- It is unspecified whether writes to the same portion of the file prior to the call are visible by read references to the memory region.
- It is unspecified whether unmodified pages in the specified range are also written to the underlying hardware.

If *flags* is `MS_ASYNC`, the function may return immediately once all write operations are scheduled; if *flags* is `MS_SYNC`, the function does not return until all write operations are completed.

`MS_INVALIDATE` synchronizes the contents of the memory region to match the current file contents.

- All writes to the mapped portion of the file made prior to the call are visible by subsequent read references to the mapped memory region.
- It is unspecified whether writes references prior to the call, by any process, to memory regions mapped to the same portion of the file using `MAP_SHARED`, are visible by read references to the region.

If `msync()` causes any write to the file, then the file's `st_ctime` and `st_mtime` fields are marked for update.

## Returned Value

Upon successful completion, `msync()` returns 0. Otherwise, it returns -1 and sets `errno` to indicate the error.

The `msync()` function will fail if:

- |        |  |
|--------|--|
| EINVAL | The <i>addr</i> argument is not a multiple of the page size as returned by <i>sysnonf</i> .  |
| EIO    | An I/O error occurred while reading from or writing to the file system.  |
| ENOMEM | Some or all the addresses in the range [ <i>addr</i> , <i>addr</i> + range [ <i>addr</i> , <i>addr</i> + <i>len</i> ) are <i>len</i> ) are invalid for the address space of the process or pages not mapped are specified. |

## Related Information

- “`mmap()` — Map Pages of Memory” on page 832
- “`sysconf()` — Determine System Configuration Options” on page 1483

## munmap() — Unmap Pages of Memory

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2 OS/390 UNIX	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/mman.h>
int munmap(void *addr, size_t len);
```

### General Description

The `munmap()` function removes the mappings for pages in the range `[addr, addr + len)` rounding the `len` argument up to the next multiple of the page size as returned by `sysconf()`. If `addr` is not the address of a mapping established by a prior call to `mmap()`, the behavior is undefined. After a successful call to `munmap()` and before any subsequent mapping of the unmapped pages, further references to these pages will result in the delivery of a SIGBUS or SIGSEGV signal to the process.

**\_\_MAP\_MEGA mapping:** The `munmap` service removes the mapping for pages in the requested range. The requested range may span multiple maps, and the maps may represent the same or different files. The pages in the range may be part of a regular mapping or may be part of a `__MAP_MEGA` mapping. When unmapping a regular mapping, entire pages are unmapped; when unmapping a `__MAP_MEGA` mapping, entire segments are unmapped.

**Map\_address:** The value of map address must be a multiple of the page size. The specified value does not have to be the start of a mapping. However, if the value specified for `Map_address` falls within a `__MAP_MEGA` map, then the address is rounded down to a megabyte multiple so that an entire segment is included in the unmap operation. It is not possible to unmap a part of a segment when processing a `__MAP_MEGA` map.

**Map\_length:** The length can be the size of the whole mapping, or a part of it. If the specified length is not in multiples of the page size, it will be rounded up to a page boundary. If the `Map_address` plus the `Map_length` falls within a `__MAP_MEGA` map, then the length is rounded up to a segment boundary, thus including the entire segment (not necessarily the entire `__MAP_MEGA` mapping).

### Returned Value

Upon successful completion, `munmap()` returns 0. Otherwise, it returns -1 and sets `errno` to indicate the error.

The `munmap()` function will fail if:

- EINVAL     The `addr` argument is not a multiple of the page size as returned by `sysconf`.
- EINVAL     Addresses in the outside the valid range for the address space of a process.
- EINVAL     The `len` argument is 0.



**Related Information**

- “mmap() — Map Pages of Memory” on page 832
- “sysconf() — Determine System Configuration Options” on page 1483

## nextafter() — Next Representable Double Float

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>
```

```
double nextafter(double x, double y);
```

### General Description

The `nextafter()` function computes the next representable double-precision floating point value following  $x$  in the direction of  $y$ . Thus, if  $y$  is less than  $x$ , `nextafter()` returns the largest representable floating point number less than  $x$ .

### Returned Value

`nextafter()` always succeeds.

### Related Information

- “`math.h`” on page 35

## nftw() — Traverse a File Tree

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ftw.h>

int nftw(const char *path,
        int (*fn)(const char *, const struct stat *,
                  int, struct FTW *),
        int ndirs, int flags);
```

### General Description

The `nftw()` function recursively descends the directory hierarchy rooted in *path*. It is similar to `ftw()` except that it takes an additional argument *flags*, which is a bitwise inclusive-OR of zero or more of the following flags:

#### FTW\_CHDIR

If set, `nftw()` will change the current working directory to each directory as it reports files in that directory. If clear, `nftw()` will not change the current working directory.

#### FTW\_DEPTH

If set, `nftw()` will report all files in a directory before reporting the directory itself. If clear, `nftw()` will report any directory before reporting the files in that directory.

#### FTW\_MOUNT

If set, `nftw()` will only report files in the same filesystem as *path*. If clear, `nftw()` will report all files encountered during the walk.

#### FTW\_PHYS

If set, `nftw()` performs a physical walk and does not follow symbolic links. If clear, `nftw()` will follow links instead of reporting them, and will not report the same file twice.

At each file it encounters, `nftw()` calls the user-supplied function *fn* with four arguments:

- the first argument is the pathname of the object.
- the second argument is a pointer to a stat buffer containing information on the object.
- the third argument is an integer giving additional information. Its value is one of the following:

FTW\_D      for a directory

FTW\_DP     for a directory whose subdirectories have been visited. (This condition will only occur if FTW\_DEPTH is included in *flags*.)

FTW\_DNR    for a directory that cannot be read

FTW\_F      for a file

FTW_SL	for a symbolic link
FTW_SLN	for a symbolic link that does not name an existing file. (This condition will only occur if FTW_PHYS is not included in <i>flags</i> .)
FTW_NS	for an object other than a symbolic link on which stat() could not be successfully executed. If the object is a symbolic link, and stat() failed, it is unspecified whether nftw() passes FTW_SL or FTW_NS to the user-supplied function.

- the fourth argument is a pointer to an FTW structure. The value of *base* is the offset of the object's filename in the pathname passed as the first argument to *fn()*. The value of *level* indicates depth relative to the root of the walk, where the root level is 0.

The argument *depth* limits the directory depth for the search. At most one file descriptor will be used for each directory level.

### Special Behavior for C++

Because C and C++ linkage conventions are incompatible, nftw() cannot receive a C++ function pointer as the argument. If you attempt to pass a C++ function pointer to nftw(), the compiler will flag it as an error. You can pass a C or C++ function to nftw() by declaring it as `extern "C"`.

### Returned Value

The nftw() function continues until the first of the following conditions occurs:

- An invocation of *fn()* returns a nonzero value, in which case nftw() returns that value.
- The nftw() function detects an error other than EACCES (see FTW\_DNR and FTW\_NS above), in which case nftw() returns -1 and sets *errno* to indicate the error.
- The tree is exhausted, in which case nftw() returns 0.

The nftw() function will fail under the following conditions, setting *errno* to the given values: All other *errno*'s returned by nftw() are unchanged.

EACCES	Search permission is denied for any component of <i>path</i> or read permission is denied for <i>path</i> , or <i>fn()</i> returns -1 and does not reset <i>errno</i> .
ELOOP	Too many symbolic links were encountered.
ENOENT	A component of <i>path</i> does not name an existing file or <i>path</i> is an empty string.
ENOTDIR	A component of <i>path</i> is not a directory.
EMFILE	{OPEN_MAX} file descriptors are currently open in the calling process.
ENFILE	Too many files are currently open in the system.
ENAMETOOLONG	Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.
ENAMETOOLONG	The length of <i>path</i> exceeds {PATH_MAX}, or a pathname component is longer than {PATH_MAX}.

errno may also be set if the function *fn* causes it to be set.

### Related Information

- “ftw.h” on page 29
- “ftw() — Traverse a File Tree” on page 491
- “lstat() — Get Status of File or Symbolic Link” on page 778
- “opendir() — Open a Directory” on page 877
- “readdir() — Read an Entry from a Directory” on page 1086
- “stat() — Get File Information” on page 1404

## nice() — Change Priority of a Process

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE
#include <unistd.h>
```

```
int nice(int increment);
```

### General Description

nice() adds the value of *increment* to the nice value of the calling process. A process' nice value is a non-negative number for which a more positive value results in a lower CPU priority.

A maximum nice value of  $2 \times \{\text{NZERO}\} - 1$  and a minimum value of zero are imposed by the system. Requests for values above or below these limits result in the nice value being set to the corresponding limit. Only a process with appropriate privileges can lower the nice value.

The changing of a process' nice value has the equivalent effect on a process' scheduling priority value, since they both represent the process' relative CPU priority. For example, increasing one's nice value to its maximum value of  $(2 \times \text{NZERO}) - 1$  has the equivalent effect of setting one's scheduling priority value to its maximum value (19), and will be reflected on the nice(), getpriority(), and setpriority() functions.

### Returned Value

If successful, nice() return the new nice value minus (NZERO).

If unsuccessful, nice() returns -1, and returns the error value in errno. The following are the possible values of errno:

- EPERM      The value of *increment* was negative and the calling process does not have the appropriate privileges.
- ENOSYS     The system does not support this function.

Because nice() can return the value -1 on successful completion, it is necessary to set the external variable errno to 0 prior to a call to nice(). If nice() returns the value -1, then errno can be checked to see if an error occurred or if the value is a legitimate nice value.

### Related Information

- “limits.h” on page 32
- “unistd.h” on page 53
- “getpriority() — Get Process Scheduling Priority” on page 578
- “setpriority() — Set Process Scheduling Priority” on page 1257

## nlist() — Get Entries from a Name List

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 5.2.2

### Format

```
#include <nlist.h>
```

```
int nlist (const char * loadname, struct nlist * np);
```

### General Description

The `nlist()` function allows a program to examine the name list in the executable file named by the *loadname* parameter. It selectively extracts a list of values and places them in the array of `nlist` structures pointed to by the *np* parameter.

The name list specified by the *np* parameter consists of an array of structures containing names of variables, types and values. The list is terminated with an element that has a null string in the name structure member. Each variable name is looked up in the name list of the executable file. If the name is found, the type and the value of the name is copied into the `nlist` structure field. If the name is not found, both the type and value entry will be set to zero.

All entries are set to zero if the specified executable file cannot be read or it does not contain a valid name list.

### Notes:

1. The only variable type that will be supported by this version of `nlist()` is external function.
2. `nlist()` will extract the offset of the external functions from *loadname*
3. The type returned in `nlist` structure will always be 2 to indicate function if the function name is found in *loadname*.
4. *loadname* must be a HFS linear format load module containing `main()`.
5. *loadname* cannot be a dll (dynamic link library) or a fetchable load module.

### Returned Value

If `nlist()` is successful, zero is returned. The offset and type of functions if found will be returned in the `nlist` structure. If `nlist()` is unsuccessful, -1 is returned.

### Related Information

None.

## nl\_langinfo() — Retrieve Locale Information

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#include <langinfo.h>
```

```
char *nl_langinfo(nl_item item);
```

### General Description

Retrieves from the current locale the string that describes the requested information specified by *item*.

For a list of macros that define the constants used to identify the information queried in the current locale, see Table 6 on page 30.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

Returns a pointer to a null-terminated string containing information concerning the active language or cultural area. The active language or cultural area is determined by the most recent `setlocale()` call. The array pointed to by the returned value is modified by subsequent calls to the function. The array shall not be modified by the user's program.

If the item is not valid, the function returns a pointer to an empty string.

### Example CBC3BN01

```
/* CBC3BN01
   This example retrieves the current codeset name using the
   nl_langinfo() function.

*/
#include "langinfo.h"
#include "locale.h"
#include "stdio.h"

main() {
    char *codeset;
    setlocale(LC_ALL, "");
    codeset = nl_langinfo(CODESET);
    printf("codeset is %s\n", codeset);
}
```



**Related Information**

- “langinfo.h” on page 30
- “nl\_types.h” on page 38
- “localdtconv() — Date/Time Formatting Convention Inquiry” on page 754
- “localeconv() — Query Numeric Conventions” on page 756
- “setlocale() — Set Locale” on page 1241

## nrnd48() — Pseudo-random Number Generator

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>
```

```
long int nrnd48(unsigned short int x16v[3]);
```

### General Description

The drand48(), erand48(), jrand48(), lrand48(), mrand48() and nrnd48() functions generate uniformly distributed pseudorandom numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The functions drand48() and erand48() return non-negative, double-precision, floating-point values, uniformly distributed over the interval [0.0,1.0).

The functions lrand48() and nrnd48() return non-negative, long integers, uniformly distributed over the interval [0,2\*\*31).

The functions mrand48() and jrand48() return signed long integers, uniformly distributed over the interval [-2\*\*31,2\*\*31).

The nrnd48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(i), according to the linear congruential formula:

$$X(n+1) = (aX(n) + c) \bmod (2^{**}48) \quad n \geq 0$$

The nrnd48() function uses storage provided by the argument array, x16v[3], to save the most recent 48-bit integer value in the sequence, X(i). The nrnd48() function uses x16v[0] for the low order (rightmost) 16 bits, x16v[1] for the middle order 16 bits, and x16v[2] for the high order 16 bits of this value.

The initial values of a, and c are:

```
a   = 5deece66d (base 16)
c   = b          (base 16)
```

The values a and c, may be changed by calling the lcong48() function. The initial values of a and c are restored if either the seed48() or srand48() function is called.

### Special Behavior for OS/390 UNIX Services

You can make the nrnd48() function and other functions in the drand48 family thread specific by setting the environment variable \_RAND48 to the value THREAD before calling any function in the drand48 family.

If you do not request thread specific behavior for the drand48 family, C/370 serializes access to the storage for X(n), a and c by functions in the drand48 family when they are called by a multithreaded application.

If thread specific behavior is requested and the nrand48() function is called from thread *t*, the nrand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, *X(t,i)*, for the thread according to the linear congruential formula:

$$X(t,n+1) = (a(t)X(t,n) + c(t)) \bmod (2^{**}48) \quad n \geq 0$$

The nrand48() function uses storage provided by the argument array, *x16v[3]*, to save the most recent 48-bit integer value in the sequence, *X(t,i)*. The nrand48() function uses *x16v[0]* for the low order (rightmost) 16 bits, *x16v[1]* for the middle order 16 bits, and *x16v[2]* for the high order 16 bits of this value.

The initial values of *a(t)* and *c(t)* on the thread *t* are:

```
a(t)   = 5deece66d (base 16)
c(t)   = b         (base 16)
```

The values *a(t)* and *c(t)* may be changed by calling the lcong48() function from the thread *t*. The initial values of *a(t)* and *c(t)* are restored if either the seed48() or srand48() function is called from the thread.

## Returned Value

The nrand48() function saves the generated 48-bit value, *X(n+1)*, in storage provided by the argument array, *x16v[3]*. The nrand48() function transforms the generated 48-bit value to a non-negative, long integer value on the interval  $[0, 2^{**}31)$  and returns this transformed value.

## Special Behavior for OS/390 UNIX Services

If thread specific behavior is requested for the drand48 family and the nrand48() function is called on thread *t*, the nrand48() function saves the generated 48-bit value, *X(t,n+1)*, in storage provided by the argument array, *x16v[3]*. The nrand48() function transforms the generated 48-bit value to a non-negative, long integer value on the interval  $[0, 2^{**}31)$  and returns this transformed value.

## Related Information

- “stdlib.h” on page 45
- “drand48() — Pseudo-random Number Generator” on page 286
- “erand48() — Pseudo-random Number Generator” on page 314
- “jrand48() — Pseudo-random Number Generator” on page 724
- “lcong48() — Pseudo-random Number Initializer” on page 736
- “lrand48() — Pseudo-random Number Generator” on page 773
- “mrand48() — Pseudo-random Number Generator” on page 843
- “seed48() — Pseudo-random Number Initializer” on page 1156
- “srand48() — Pseudo-random Number Initializer” on page 1401

## ntohl() — Translate a Long Integer into Host Byte Order

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>

in_addr_t ntohl(in_addr_t netlong);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long ntohl(unsigned long a);
```

### General Description

The `ntohl()` call translates a long integer from network byte order to host byte order.

Parameter	Description
<i>a</i>	The unsigned long integer to be put into host byte order.
<code>in_addr_t <i>netlong</i></code>	Is typed to the unsigned long integer to be put into host byte order.

### Notes:

1. For OS/390, host byte order and network byte order are the same.
2. Since this function is implemented as a macro, you need one of the feature test macros and the `inet` header file.

### Returned Value

`ntohl()` returns the translated long integer.

### Related Information

- “`htonl()` — Translate Address Host to Network Long” on page 648
- “`htons()` — Translate an Unsigned Short Integer into Network Byte Order” on page 649
- “`ntohs()` — Translate an Unsigned Short Integer into Host Byte Order” on page 871

## ntohs() — Translate an Unsigned Short Integer into Host Byte Order

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <arpa/inet.h>

in_port_t ntohs(in_port_t netshort);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <arpa/inet.h>
#include <netinet/in.h>

unsigned short ntohs(unsigned short a);
```

### General Description

The `ntohs()` call translates a short integer from network byte order to host byte order.

#### Parameter

#### Description

*a* The unsigned short integer to be put into host byte order.

`in_port_t netshort` Is typed to the unsigned short integer to be put into host byte order.

#### Notes:

1. For OS/390, host byte order and network byte order are the same.
2. Since this function is implemented as a macro, you need one of the feature test macros and the `inet` header file.

### Returned Value

`ntohs()` returns the translated short integer.

### Related Information

- “`htonl()` — Translate Address Host to Network Long” on page 648
- “`htons()` — Translate an Unsigned Short Integer into Network Byte Order” on page 649
- “`ntohl()` — Translate a Long Integer into Host Byte Order” on page 870

# open() — Open a File

## Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.	both	MVS 4.3

## Format

```
#define _POSIX_SOURCE
#include <fcntl.h>
```

```
int open(const char *pathname, int options, ...);
```

## General Description

Opens a file and returns a number called a *file descriptor*.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

The *pathname* argument must be a hierarchical file system (HFS) file name. You can use this file descriptor to refer to the file in subsequent I/O operations, for example, `read()` or `write()`. Each file opened by a process gets a new file descriptor.

Note the restriction that FIFOs, POSIX terminals, and character special files applies only to OS/390 C programs running POSIX(ON). See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information.

The argument *pathname* is a string giving the name of the file you want to open. The integer *options* specifies options for the open operation by taking the bitwise inclusive OR of symbols defined in the `fcntl.h` header file. The options indicate whether the file should be accessed for reading, writing, reading and writing, and so on.

An additional argument (...) is required if the `O_CREAT` option is specified in *options*. This argument may be called the *mode* and has the `mode_t` type. It specifies file permission bits to be used when a file is created. All the file permission bits are set to the bits of *mode*, except for those set in the file-mode creation mask of the process. Here is a list of symbols that can be used for a mode.

- S\_ISUID** Privilege to set the user ID (UID) for execution. When this file is run through an `exec` function, the effective user ID of the process is set to the owner of the file, so that the process has the same authority as the file owner rather than the authority of the actual invoker.
- S\_ISGID** Privilege to set group ID (GID) for execution. When this file is run through an `exec` function, the effective group ID of the process is set to the group ID of the file, so that the process has the same authority as the file owner rather than the authority of the actual invoker.
- S\_ISVTX** Indicates shared text. Keep loaded as an executable file in storage.
- S\_IRUSR** Read permission for the file owner.

S_IWUSR	Write permission for the file owner.
S_IXUSR	Search permission (for a directory) or execute permission (for a file) for the file owner.
S_IRWXU	Read, write, and search, or execute, for the file owner; S_IRWXG is the bitwise inclusive OR of S_IRUSR, S_IWUSR, and S_IXUSR.
S_IRGRP	Read permission for the file's group.
S_IWGRP	Write permission for the file's group.
S_IXGRP	Search permission (for a directory) or execute permission (for a file) for the file's group.
S_IRWXG	Read, write, and search or execute permission for the file's group. S_IRWXG is the bitwise inclusive <b>OR</b> of S_IRGRP, S_IWGRP, and S_IXGRP.
S_IROTH	Read permission for users other than the file owner.
S_IWOTH	Write permission for users other than the file owner.
S_IXOTH	Search permission for a directory, or execute permission for a file, for users other than the file owner.
S_IRWXO	Read, write, and search or execute permission for users other than the file owner. S_IRWXO is the bitwise inclusive OR of S_IROTH, S_IWOTH, and S_IXOTH.

Most open operations position a *file offset* (an indicator showing where the next read or write will take place in the file) at the beginning of the file; however, there are options that can change this position. One of the following *must* be specified in the *options* argument of the `open()` operation:

O_RDONLY	Open for reading only
O_WRONLY	Open for writing only
O_RDWR	Open for both reading and writing

One or more of the following can also be specified in *options*:

O_APPEND	Positions the file offset at the end of the file before each write operation.
O_CREAT	Indicates that the call to <code>open()</code> has a <i>mode</i> argument.  If the file being opened already exists, O_CREAT has no effect, except when O_EXCL is also specified; see O_EXCL following.  If the file being opened does not exist, it is created. The file's user ID is set to the process's effective user ID, and its group ID is set to the group ID of its directory. File permission bits are set according to <i>mode</i> .  If O_CREAT is specified and the file did not previously exist, a successful <code>open()</code> sets the access time, change time, and modification time for the file. It also updates the change time and modification time fields in the parent directory.

- O\_EXCL** If both **O\_EXCL** and **O\_CREAT** are specified, `open()` fails if the file already exists. If both **O\_EXCL** and **O\_CREAT** are specified and *pathname* names a symbolic link, `open()` fails regardless of the contents of the symbolic link.
- O\_NOCTTY** If *pathname* specifies a terminal, `open()` does not make the terminal the controlling terminal of the process (and the session). If **O\_NOCTTY** is not specified, the terminal becomes the controlling terminal if the following conditions are true:
- The process is a session leader.
  - There is no controlling terminal for the session.
  - The terminal is not already a controlling terminal for another session.
- O\_NONBLOCK**  
Has different meanings depending on the situation.
- When you are opening a FIFO special file with **O\_RDONLY** or **O\_WRONLY**:  
  
If **O\_NONBLOCK** is specified, a read-only `open()` returns immediately. A write-only `open()` returns with an error if no other process has the FIFO open for reading.  
  
If **O\_NONBLOCK** is not specified, a read-only `open()` blocks until another process opens the FIFO for writing. A write-only `open()` blocks until another process opens the FIFO for reading.
  - When you are opening a character special file that supports a nonblocking `open()`, **O\_NONBLOCK** controls whether subsequent reads and writes can block.
- O\_TRUNC** Truncates the file to zero length if the file exists and is a regular file, provided that the file is successfully opened with **O\_RDWR** or **O\_WRONLY**. The mode and owner of the file are unchanged. This option should not be used with **O\_RDONLY**. **O\_TRUNC** has no effect on FIFO special files or directories.
- If **O\_TRUNC** is specified and the file previously existed, a successful `open()` updates the change time and modification time for the file.
- O\_SYNC** Force synchronous update. If this flag is 1, every `write()` operation on the file is written to permanent storage. That is, the file system buffers are forced to permanent storage. This flag is supported on MVS 5.1, but ignored on MVS 4.3. See `fsync()` also.
- On return from a function that performs a synchronous update, the program is assured that all data for the file has been written to permanent storage.

If *pathname* refers to a STREAM file, *oflag* may be constructed from **O\_NONBLOCK** OR-ed with either **O\_RDONLY**, **O\_WRONLY** or **O\_RDWR**. Other flag values are not applicable to STREAMS devices and have no effect on them. The value **O\_NONBLOCK** affects the operation of STREAMS drivers and certain functions applied to file descriptors associated with STREAMS files. For STREAMS drivers, the implementation of **O\_NONBLOCK** is device-specific.



**Note:** OS/390 UNIX services do not supply any STREAMS devices or pseudodevices. It is impossible for `open()` to return a valid STREAMS file descriptor.

## Returned Value

If successful, `open()` returns a file descriptor. If unsuccessful, it returns the value `-1` and sets `errno` to one of the following:

EACCES	Access is denied. Possible reasons: include: <ul style="list-style-type: none"> <li>• The process does not have search permission on a component in <i>pathname</i>.</li> <li>• The file exists, but the process does not have permission to open the file in the way specified by the flags.</li> <li>• The file does not exist, and the process does not have write permission on the directory where the file is to be created.</li> <li>• <code>O_TRUNC</code> was specified, but the process does not have write permission on the file.</li> </ul>
EBUSY	The process attempted to open a file that is in use.
EEXIST	<code>O_CREAT</code> and <code>O_EXCL</code> were specified, and either the named file refers to a symbolic link, or the named file already exists.
EINTR	<code>open()</code> was interrupted by a signal.
EINVAL	The <i>options</i> parameter does not specify a valid combination of the <code>O_RDONLY</code> , <code>O_WRONLY</code> and <code>O_TRUNC</code> bits.
EIO	The <i>pathname</i> argument names a STREAMS file and a hang-up or error occurred during the <code>open()</code> .
EISDIR	<i>pathname</i> is a directory, and <i>options</i> specifies write or read/write access.
ELOOP	A loop exists in symbolic links. This error is issued if the number of symbolic links detected in the resolution of <i>pathname</i> is greater than <code>POSIX_SYMLINK_MAX</code> .
EMFILE	The process has reached the maximum number of file descriptors it can have open.
ENAMETOOLONG	<i>pathname</i> is longer than <code>PATH_MAX</code> characters, or some component of <i>pathname</i> is longer than <code>NAME_MAX</code> characters while <code>_POSIX_NO_TRUNC</code> is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds <code>PATH_MAX</code> . The <code>PATH_MAX</code> and <code>NAME_MAX</code> values can be determined using <code>pathconf()</code> .
ENFILE	The system has reached the maximum number of file descriptors it can have open.
ENOENT	Typical causes: <ul style="list-style-type: none"> <li>• <code>O_CREAT</code> is not specified, and the named file does not exist.</li> <li>• <code>O_CREAT</code> is specified, and either the prefix of <i>pathname</i> does not exist or the <i>pathname</i> argument is an empty string.</li> </ul>

ENOMEM	The <i>pathname</i> argument names a STREAMS file and the system is unable to allocate resources.
ENOSPC	The directory or file system intended to hold a new file has insufficient space.
ENOSR	The <i>pathname</i> argument names a STREAMS-based file and the system is unable to allocate a STREAM.
ENOSYS	For master pseudoterminals, slave initialization did not complete.
ENOTDIR	A component of <i>pathname</i> is not a directory.
ENXIO	O_NONBLOCK and O_WRONLY were specified and the named file is a FIFO, but no process has the file open for reading. For a pseudoterminal, the requested minor number exceeds the maximum number supported by the installation.
EPERM	For slave pseudoterminals, permission to open is denied for one of these reasons: <ul style="list-style-type: none"> <li>• It is the first open of the slave after the master pseudoterminal was opened, and the user ID associated with the two opening processes is not the same.</li> <li>• There was an internal error in the security system after the master pseudoterminal was opened.</li> <li>• The attempt to open the slave used a different path name than earlier opens used.</li> </ul>
EROFS	<i>pathname</i> is on a read-only file system, and one or more of the options O_WRONLY, O_RDWR, O_TRUNC, or O_CREAT (if the file does not exist) was specified.

### Example

The following opens an output file for appending:

```
int fd;
fd = open("outfile", O_WRONLY | O_APPEND);
```

The following statement creates a new file with read/write/execute permissions for the creating user. If the file already exists, open() fails.

```
fd = open("newfile", O_WRONLY | O_CREAT | O_EXCL, S_IRWXU);
```

### Related Information

- “fcntl.h” on page 28
- “close() — Close a File” on page 191
- “creat() — Create a New File or Rewrite an Existing One” on page 233
- “dup() — Duplicate an Open File Descriptor” on page 288
- “exec Functions” on page 322
- “fcntl() — Control Open File Descriptors” on page 350
- “fsync() — Write Changes to Direct-Access Storage” on page 483
- “lseek() — Change the Offset of a File” on page 776
- “read() — Read From a File or Socket” on page 1080
- “stat() — Get File Information” on page 1404
- “umask() — Set and Retrieve File Creation Mask” on page 1647
- “write() — Write Data on a File or Socket” on page 1780

# opendir() — Open a Directory

## Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

## Format

```
#define _POSIX_SOURCE
#include <dirent.h>
```

```
DIR *opendir(const char *dirname);
```

## General Description

Opens a directory so that it can be read with `readdir()` or `__readdir2()`. *dirname* is a string giving the name of the directory you want to open. The first `readdir()` or `__readdir2()` call reads the first entry in the directory.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

## Returned Value

If successful, `opendir()` returns a pointer to a `DIR` object. This object describes the directory and is used in subsequent operations on the directory, in the same way that `FILE` objects are used in file I/O operations. If unsuccessful, `opendir()` returns a `NULL` pointer and sets `errno` to one of the following:

- EACCES** The process does not have permission to search some component of *dirname*, or it does not have read permission on the directory itself.
- ELOOP** A loop exists in the symbolic links. This error is issued if more than `POSIX_SYMLINK` (defined in the `limits.h` header file) symbolic links are encountered during resolution of the *dirname* argument.
- EMFILE** The process has too many other file descriptors already open.
- ENAMETOOLONG** *dirname* is longer than `PATH_MAX` characters, or some component of *dirname* is longer than `NAME_MAX` characters while `_POSIX_NO_TRUNC` is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds `PATH_MAX`. The `PATH_MAX` and `NAME_MAX` values can be determined using `pathconf()`.
- ENFILE** The entire system has too many other file descriptors already open.
- ENOMEM** There is not enough storage available to open the directory.
- ENOENT** The directory *dirname* does not exist.
- ENOTDIR** Some component of the *dirname* path name is not a directory.

## Example

### CBC3B001

```

/* CBC3B001
   This example opens a directory.
*/
#define _POSIX_SOURCE
#include <dirent.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <stdio.h>

void traverse(char *fn, int indent) {
    DIR *dir;
    struct dirent *entry;
    int count;
    char path[1025];
    struct stat info;

    for (count=0; count<indent; count++) printf(" ");
    printf("%s\n", fn);

    if ((dir = opendir(fn)) == NULL)
        perror("opendir() error");
    else {
        while ((entry = readdir(dir)) != NULL) {
            if (entry->d_name[0] != '.') {
                strcpy(path, fn);
                strcat(path, "/");
                strcat(path, entry->d_name);
                if (stat(path, &info) != 0)
                    fprintf(stderr, "stat() error on %s: %s\n", path,
                        strerror(errno));
                else if (S_ISDIR(info.st_mode))
                    traverse(path, indent+1);
            }
        }
        closedir(dir);
    }
}

main() {
    puts("Directory structure:");
    traverse("/etc", 0);
}

```

## Output

```

Directory structure:
/etc
/etc/samples
/etc/samples/IBM
/etc/IBM

```

## Related Information

- “dirent.h” on page 24
- “stdio.h” on page 43
- “sys/types.h” on page 49
- “closedir() — Close a Directory” on page 194
- “\_\_opendir2() — Open a Directory” on page 880
- “readdir() — Read an Entry from a Directory” on page 1086

- “rewinddir() — Reposition a Directory Stream to the Beginning” on page 1141
- “seekdir() — Set Position of Directory Stream” on page 1158
- “telldir() — Current Location of Directory Stream” on page 1557

## \_\_opendir2() — Open a Directory

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R6

### Format

```
#define _OPEN_SYS_DIR_EXT
#include <dirent.h>
```

```
DIR *__opendir2(const char *dirname, size_t bufsize);
```

### General Description

Opens a directory so that it can be read with `readdir()` or `__readdir2()`. The first `readdir()` or `__readdir2()` call reads the first entry in the directory.

*dirname* is a string giving the name of the directory you want to open. *bufsize* is the size (in bytes) of the internal work buffer used by `readdir()` or `__readdir2()` to hold directory entries. The larger the buffer, the less overhead there will be when reading through large numbers of directory entries. This buffer will exist until the directory is closed. If the specified buffer size is too small, it is ignored. A minimum-size buffer is used instead.

`__opendir2()` is the same as `opendir()`, except that the buffer size can be specified as a parameter.

### Returned Value

If successful, `__opendir2()` returns a pointer to a `DIR` object. This object describes the directory and is used in subsequent operations on the directory, in the same way that `FILE` objects are used in file I/O operations.

If unsuccessful, `__opendir2()` returns a `NULL` pointer and sets `errno` to one of the following:

- EACCES** The process does not have permission to search some component of *dirname*, or it does not have read permission on the directory itself.
- ELOOP** A loop exists in the symbolic links. This error is issued if more than `POSIX_SYMLINK` (defined in the `limits.h` header file) symbolic links are encountered during resolution of the *dirname* argument.
- EMFILE** The process has too many other file descriptors already open.
- ENAMETOOLONG** *dirname* is longer than `PATH_MAX` characters, or some component of *dirname* is longer than `NAME_MAX` characters while `_POSIX_NO_TRUNC` is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds `PATH_MAX`. The `PATH_MAX` and `NAME_MAX` values can be determined using `pathconf()`.
- ENFILE** The entire system has too many other file descriptors already open.
- ENOMEM** There is not enough storage available to open the directory using a buffer that is length *bufsize* bytes long.

ENOENT     The directory *dirname* does not exist.

ENOTDIR    Some component of the *dirname* path name is not a directory.

### Related Information

- “dirent.h” on page 24
- “stdio.h” on page 43
- “sys/types.h” on page 49
- “closedir() — Close a Directory” on page 194
- “opendir() — Open a Directory” on page 877
- “readdir() — Read an Entry from a Directory” on page 1086
- “rewinddir() — Reposition a Directory Stream to the Beginning” on page 1141
- “seekdir() — Set Position of Directory Stream” on page 1158
- “telldir() — Current Location of Directory Stream” on page 1557

## openlog() — Open the System Control Log

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <syslog.h>
```

```
void openlog(const char *ident, int logopt, int facility);
```

### General Description

The `openlog()` function optionally opens a connection to the logging facility, and sets process attributes that affect subsequent calls to the `syslog()` function. The argument *ident* is a string that is prefixed to every message. *logopt* is a bit field indicating logging options. Current values of *logopt* are:

**LOG\_PID** Log the processID with each message. This is useful for identifying specific processes.

**LOG\_CONS** Write messages to the system console if they cannot be sent to the logging facility. This option is safe to use in processes that have no controlling terminal, since the `syslog()` function forks before opening the console.

**LOG\_NDELAY** Open the connection to the logging facility immediately. Normally the open is delayed until the first message is logged. This is useful for programs that need to manage the order in which file descriptors are allocated..

**LOG\_ODELAY** Delay open until `syslog()` is called.

**LOG\_NOWAIT** Do not wait for child processes that have been forked to log messages onto the console. This option should be used by processes that enable notification of child termination using `SIGCHLD`, since the `syslog()` function may otherwise block waiting for a child whose exit status has already been collected.

The *facility* argument encodes a default facility to be assigned to all messages that do not have an explicit facility already encoded. The initial default facility is as follows:

**LOG\_USER** Message generated by random processes. This is the default facility identifier if none is specified.



**Returned Value**

The `openlog()` function does not return a value.

No errors are defined.

**Related Information**

- “syslog.h” on page 47
- “closelog() — Close the Control Log” on page 196
- “setlogmask() — Set the Mask for the Control Log” on page 1250
- “syslog() — Send a Message to the Control Log” on page 1486

## \_\_openMvsRel() — Get the OS/390 UNIX Release Number

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#include <unistd.h>
```

```
int __openMvsRel( void );
```

### General Description

The `__openMvsRel()` function returns the release number of the active services kernel.

### Returned Value

The `__openMvsRel()` function returns an integer which indicates the release number of the active OS/390 UNIX kernel as follows:

- 0 - OS/390 UNIX kernel is not started or not installed.
- 1 - OS/390 UNIX release 1 (MVS 4.3) kernel is active.
- 2 - OS/390 UNIX release 2 (MVS 5.1) kernel is active.
- 3 - OS/390 UNIX release 3 (MVS 5.2) kernel is active.
- 4 - OS/390 UNIX release 4 (OS/390 release 2) kernel is active.

`__openMvsRel()` returns the level of the active kernel even if invoked from CICS (where the kernel itself cannot be used).

No `errno`s are defined for the `__openMvsRel()` function.

### Related Information

- “`unistd.h`” on page 53
- “`__isPosixOn()` — Test for Posix Runtime Option” on page 713

## \_\_open\_stat() — Open a File and Get File Status Information

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R6

### Format

```
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/stat.h>
```

```
int __open_stat(const char *pathname, int options, mode_t mode,
               struct stat *info);
```

### General Description

Opens a file and returns a number called a *file descriptor*. \_\_open\_stat() also returns information about the opened file. \_\_open\_stat() is a combination of open() and fstat().

The parameters are:

#### Parameters      Description

<i>pathname</i>	<p>This parameter is a null-terminated character string containing the hierarchical file system (HFS) pathname of the file to be opened.</p> <p><i>pathname</i> can begin with or without a slash.</p> <ul style="list-style-type: none"> <li>A pathname beginning with a slash is an absolute pathname. The slash refers to the root directory, and the search for the file starts at the root directory.</li> <li>A pathname not beginning with a slash is a relative pathname. The search for the file begins at the working directory.</li> </ul> <p>See “open() — Open a File” on page 872 for more information about the <i>pathname</i> parameter and the types of files that can be opened.</p>
<i>options</i>	<p>An integer containing option bits for the open operation. These options are the same as those in the <i>options</i> parameter passed to open(). These bits are defined in fcntl.h. For a list of these option bits and their meaning, see “open() — Open a File” on page 872.</p>
<i>mode</i>	<p><i>mode</i> is the same as the optional third parameter for open(), which is used when a new file is being created. For __open_stat(), the <i>mode</i> parameter is always required. If a new file is not being created, <i>mode</i> is ignored, and may be set to 0. When __open_stat() creates a file, the flag bits in <i>mode</i> specify the file permissions and other characteristics for the new file. The flag bits in <i>mode</i> are defined in sys/modes.h. For more information about the <i>mode</i> parameter, see “open() — Open a File” on page 872.</p>

*info* The *info* parameter points to an area of memory where the system will store information about the file that is opened. This parameter is the same as the *info* parameter in `fstat()` or `stat()`. If the file is successfully opened, the system returns file status information in a `stat` structure, as defined in `sys/stat.h`. The elements of this structure are described in “`stat()` — Get File Information” on page 1404.

## Returned Value

If successful, `__open_stat()` returns a file descriptor.

If unsuccessful, it returns `-1`, and sets `errno` to one of the following:

<b>EACCESS</b>	Access to the file was denied. One of the following errors occurred: <ul style="list-style-type: none"> <li>• The calling process does not have permission to search one of the directories specified in the <i>pathname</i> parameter.</li> <li>• The calling process does not have permission to open the file in the way specified by the <i>options</i> parameter.</li> <li>• The file does not exist, and the calling process does not have permission to write into files in the directory the file would have been created in.</li> <li>• The truncate option was specified, but the process does not have write permission for the file.</li> </ul>
<b>EAGAIN</b>	Resources were temporarily unavailable.
<b>EBUSY</b>	<i>pathname</i> specifies a master pseudoterminal that is either already in use or for which the corresponding slave is open.
<b>EEXIST</b>	The exclusive create option was specified, but the file already exists. Use <code>__errno2()</code> to determine the exact reason the error occurred.
<b>EFBIG</b>	A request to create a new file is prohibited because the file size limit for the process is set to 0.
<b>EINTR</b>	The <code>__open_stat()</code> operation was interrupted by a signal.
<b>EINVAL</b>	The <i>options</i> parameter does not specify a valid combination of the <code>O_RDONLY</code> , <code>O_WRONLY</code> and <code>O_TRUNC</code> bits, or the file type specified in the <i>mode</i> parameter is not valid.  Use <code>__errno2()</code> to determine the exact reason the error occurred.
<b>EISDIR</b>	The file specified by <i>pathname</i> is a directory and the <i>options</i> parameter specifies write or read/write access.  Use <code>__errno2()</code> to determine the exact reason the error occurred.
<b>ELOOP</b>	A loop exists in symbolic links encountered during resolution of the <i>pathname</i> parameter. This error is issued if more than 8 symbolic links are detected in the resolution of <i>pathname</i> .
<b>EMFILE</b>	The process has reached the maximum number of file descriptors it can have open.
<b>ENAMETOOLONG</b>	<i>pathname</i> is longer than 1023 characters, or a component of <i>pathname</i> is longer than 255 characters. (The system does not support filename truncation.)

ENODEV	<p>Typical causes of this error are:</p> <ul style="list-style-type: none"> <li>• An attempt was made to open a character special file for a device not supported by the system.</li> <li>• An attempt was made to open a character special file for a device that is not yet initialized.</li> </ul> <p>Use <code>__errno2()</code> to determine the exact reason the error occurred.</p>
ENOENT	<p>Typical causes of this error are:</p> <ul style="list-style-type: none"> <li>• The request did not specify that the file was to be created, but the file named by <i>pathname</i> was not found.</li> <li>• The request asked for the file to be created, but some component of <i>pathname</i> was not found, or the <i>pathname</i> parameter was blank.</li> </ul> <p>Use <code>__errno2()</code> to determine the exact reason the error occurred.</p>
ENOSPC	The directory or file system intended to hold a new file has insufficient space.
ENOTDIR	A component of <i>pathname</i> is not a directory.
ENXIO	The <code>__open_stat()</code> request specified write-only and nonblock for a FIFO special file, but no process has the file open for reading. For pseudoterminals, this <code>errno</code> can mean that the minor number associated with <i>pathname</i> is too big.
EPERM	The caller is not permitted to open the specified slave pseudoterminal or the corresponding master is not yet open. <code>EPERM</code> is also returned if the slave is closed with <code>HUPCL</code> set and an attempt is made to reopen it.
EROFS	<p>The <i>pathname</i> parameter names a file on a read-only file system, but options that would allow the file to be altered were specified: write-only, read/write, truncate, or -- for a new file -- create.</p> <p>Use <code>__errno2()</code> to determine the exact reason the error occurred.</p>

## Related Information

- “fcntl.h” on page 28
- “sys/stat.h” on page 48
- “close() — Close a File” on page 191
- “creat() — Create a New File or Rewrite an Existing One” on page 233
- “fstat() — Get Status Information about a File” on page 479
- “stat() — Get File Information” on page 1404
- “umask() — Set and Retrieve File Creation Mask” on page 1647
- “open() — Open a File” on page 872

## \_\_osenv() — Capture the WLM and Pthread Security Attributes

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R4

### Format

```
#define _OPEN_SYS
#include <unistd.h>

int __osenv( int osenv_func,
             int osenv_parms,
             __osenv_token *osenv_token_ptr)
```

### General Description

The `osenv()` function captures the WLM and pthread security attributes associated with the pthread and creates an environment (OSENv) to represent these attributes. The caller will then become associated with the OSENv and a token representing the OSENv will be returned to the caller.

The `osenv()` function takes the following arguments:

*osenv\_func* Specifies the following functions:

- `_OSENv_GET` Captures the current security and WLM enclave membership information and places this information in an environment (OSENv) that is associated with the caller. A token representing OSENv is returned.
- `_OSENv_SET` Changes the thread's security and WLM information to the attributes associated with the OSENv token passed as input. The thread will then become associated with the OSENv.
- `_OSENv_UNSET` Unassociates the passed OSENv from the thread. If the OSENv is not associated with any thread then the OSENv token is no longer eligible for use and the resources associated with the OSENv are released.
- `_OSENv_PERSIST` Marks the OSENv token currently in use by the task as having a persistent association with the task. An OSENv token will not be unassociated from the task when the task issues an `_OSENv_UNSET` function with the OSENv token, thereby allowing the application to reuse the OSENv token with a subsequent `_OSENv_SET` request.
- `_OSENv_UNPERSIST` Resets the persistent association of an OSENv token with the task, so that the resources associated with the OSENv token can be released and the token marked invalid when no further interest exists.

*osenv\_parms*

Specifies one of the following options:

- `_OSENV_WLM` Requests that the WLM enclave information associated with the pthread be captured and associated with the OSENV token. If the pthread is not in an WLM enclave, no WLM enclave will be reflected in the OSENV environment.
- `_OSENV_SECURITY` Requests that the task level pthread security associated with the pthread be captured and associated with the OSENV token. If task level security exists but is NOT pthread security then the request is rejected. If the pthread is under NO task level security, then NO task level security will be reflected in the OSENV environment.

*osenv\_token\_ptr,*

The OSENV token that represents the OSENV that was created and associated with the caller. The OSENV contains the WLM and pthread security attributes currently associated with the pthread.

**Usage Notes**

Multiple function requests are not recommended except for `_OSENV_SET` and `_OSENV_UNSET`.

The following usage notes are listed by function:

- `_OSENV_GET`

The environment represented by *osenv\_token\_ptr* may be propagated to other pthreads via `_OSENV_SET`.

If WLM enclave membership is not requested, (`_OSENV_WLM` is not set), then no WLM enclave attributes will be reflected in the OSENV environment.

If no security environment is requested (`_OSENV_SECURITY` is not set), then no task level security will be reflected in the OSENV environment.

- `_OSENV_SET`

If WLM enclave membership is requested (`_OSENV_WLM`) then the caller's current WLM enclave membership will be extracted and saved.

The following are the rules of WLM enclave attachment:

If the WLM enclave associated with the OSENV and the WLM enclave currently active with the pthread are the same, then the WLM enclave currently active remains unchanged (no action is taken).

If the WLM enclave associated with the OSENV and the WLM enclave currently active with the pthread are different, then the WLM enclave request fails.

If there is **no** WLM enclave currently active with the pthread, then the pthread will join the OSENV associated WLM enclave.

If the OSENV does not have an associated WLM enclave and the pthread does **not** belong to a WLM enclave, then no action is taken.

If the OSENV does not have an associated WLM enclave and the pthread belongs to a WLM enclave, then the WLM enclave request fails.

`_OSENV_SET()` must be balanced with `_OSEVN_UNSET()`.

- `_OSENV_UNSET`

If the pthread has previously requested that the OSENV persist (`_OSENV_PERSIST`) after an unset, the pthread's association with the OSENV will endure; otherwise, the caller's association with the OSENV environment will be removed.

If the input OSENV is no longer associated with a pthread, the the OSENV attributes are returned to the system and the OSENV token is marked invalid.

`_OSENV_UNSET` must be balanced with `_OSENV_SET`.

- `_OSENV_PERSIST`

`__osenv_persist()` must be balanced with `__osenv_unpersist()`.

- `_OSENV_UNPERSIST`

`__osenv_persist()` must be balanced with `__osenv_unpersist()`.

## Restrictions

The current task must not be actively associated with an OSENV environment. Prior invocations of `_OSENV_GET()` or `_OSENV_SET` must have been followed by `_OSENV_UNSET`.

If `_OSENV_SECURITY` is specified to capture pthread security, then the caller may not have a task level security environment unless it is set by `pthread_security_np()`.

If `_OSENV_WLM` is specified to set WLM Enclave membership and the caller is currently in a WLM Enclave, then the caller must belong to the OSENV WLM Enclave.

Alterations to the OSENV attributes by other programming interfaces (native interfaces) may not be detected by the next `_osenv_unset()` and the results may be unpredictable.

## Returned Value

If successful, `osenv()` returns the value 0.

If unsuccessful, `osenv()` returns -1 and sets `errno` to one of the following:

**EINVAL** The system determined that one or more of the parameters passed are in error. Consult the reason code for more information.

**ENOSYS** The function is not implemented.

**ESRCH** No such process or thread exists.

**EMVSERR** A MVS environmental or internal error has occurred.

**EMVSPARM**

Bad parameters were passed to the service.

**EMVSSAFEXTRERR**

SAF/RACF extract error.



EMVSSAF2EFF

SAF/RACF error.

EALREADY Operation already in progress.

EMVSWLMERROR

WLM detected error information. Consult the reason code for more information.

### **Related Information**

- “pthread\_security\_np() — Create or Delete Thread Level Security” on page 1029

**\_\_osenv**

---

## **Volume 2 - Part 3. Library Functions (continued)**

---

### **Functions P-Z in the OS/390 C/C++ Library**

This section lists all the OS/390 C/C++ Run-Time Library functions in alphabetical order from P-Z.

## \_\_passwd() — Verify/Change User Password

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 5.2.2

### Format

```
int __passwd(const char *username, const char *oldpass, const char *newpass);
```

### General Description

The `__passwd()` function verifies and/or changes the *username* password. The *username* is a null-terminated character string of 1 to 8 bytes. The *oldpass* is the current password for user *username*, and is a null-terminated character string of 1 to 8 bytes. The *newpass* is the replacement password. If *newpass* is **NULL**, then *oldpass* represents the password to be verified, and no password change is performed. Otherwise, *newpass* is a null-terminated character string of 1 to 8 bytes. Other installation-dependent restrictions on passwords may apply, both in terms of length and content.

If the *BPX.DAEMON* facility class profile is defined, then all modules within the address space must be loaded from a controlled library. This includes all modules in the application and run-time libraries. See also "Checking Which Module is not Defined to Program Control" in *OS/390 UNIX System Services Planning*.

### Returned Value

If successful, `__passwd()` returns 0. When *newpass* is **NULL**, the password has been verified. When *newpass* is not **NULL**, the new password has been set.

If unsuccessful, `__passwd()` returns -1, and returns the error value in `errno`. The following are the possible values of `errno`:

- EACCES     The *oldpass* is not authorized.
- EINVAL     The *username*, *oldpass*, or *newpass* argument is invalid.
- EMVSERR   The specified function is not supported in an address space where a load was done from an uncontrolled library.
- EMVSEXPIRE  
            The *oldpass* has expired.
- EMVSPASSWORD  
            The *newpass* is not valid, or does not meet the installation-exit requirements.
- EMVSSAFEXTRERR  
            The *username* access has been revoked.
- EMVSSAF2ERR  
            Internal processing error.
- ESRCH     The *username* user is unknown or not defined to the kernel.

### Related Information

- “endpwent() — User Database Functions” on page 311
- “getpass() — Read a String of Characters Without Echo” on page 567

## pathconf() — Determine Configurable Path Name Variables

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
long pathconf(const char *pathname, int varcode);
```

### General Description

Lets an application determine the value of a configuration variable, *varcode*, associated with a particular file or directory, *pathname*.

The *varcode* argument may be any one of the following symbols, defined in the *unistd.h* header file, each standing for a configuration variable:

- \_PC\_LINK\_MAX** Represents *LINK\_MAX*, the maximum number of links the file can have. If *pathname* is a directory, *pathconf()* returns the maximum number of links that can be established to the directory itself.
- \_PC\_MAX\_CANON** Represents *MAX\_CANON*, the maximum number of bytes in a terminal canonical input line. *pathname* must refer to a character special file for a terminal.
- \_PC\_MAX\_INPUT** Represents *MAX\_INPUT*, the maximum number of bytes for which space is available in a terminal input queue. That is, it refers to the maximum number of bytes that a portable application can have the user enter before the application actually reads the input. *pathname* must refer to a character special file for a terminal.
- \_PC\_NAME\_MAX** Represents *NAME\_MAX*, the maximum number of characters in a file name (not including any terminating null at the end if the file name is stored as a string). This symbol refers only to the file name itself, that is, the last component of the file's path name. *pathconf()* returns the maximum length of file names.
- \_PC\_PATH\_MAX** Represents *PATH\_MAX*, the maximum number of characters in a complete path name (not including any terminating null at the end if the path name is stored as a string). *pathconf()* returns the maximum length of a relative path name.
- \_PC\_PIPE\_BUF** Represents *PIPE\_BUF*, the maximum number of bytes that can be written “atomically” to a pipe. If more than this number of bytes is written to a pipe, the operation may take more than one physical write operation and physical read operation to read the data on the other end of the pipe. If *pathname* is a FIFO special file, *pathconf()* returns the value for the file itself. If *pathname* is a directory, *pathconf()* returns the value for any

FIFOs that exist or that can be created under the directory. If *pathname* is any other kind of file, an errno of EINVAL will be returned, indicating an invalid path name was specified.

#### **\_PC\_CHOWN\_RESTRICTED**

Represents \_POSIX\_CHOWN\_RESTRICTED defined in the unistd.h header file, and restricts use of chown() to a process with appropriate privileges. It also changes the group ID of a file to the effective group ID of the process or to one of its supplementary group IDs. If *pathname* is a directory, pathconf() returns the value for any kind of file under the directory, but not for subdirectories of the directory.

#### **\_PC\_NO\_TRUNC**

Represents \_POSIX\_NO\_TRUNC defined in the unistd.h header file, and generates an error if a file name is longer than NAME\_MAX. If *pathname* refers to a directory, the value returned by pathconf() applies to all files under that directory.

#### **\_PC\_VDISABLE**

Represents \_POSIX\_VDISABLE defined in the unistd.h header file. This symbol indicates that terminal special characters can be disabled using this character value, if it is defined; see the callable service tcsetattr() for details. *pathname* must refer to a character special file for a terminal.

### **Returned Value**

If successful, pathconf() return the value of the variable requested in *varcode*.

If unsuccessful, pathconf() returns the value -1. If a particular variable has no limit, such as PATH\_MAX, pathconf() returns the value -1 but does not change errno.

If pathconf() cannot determine an appropriate value, it sets errno to one of the following:

- |               |  |
|---------------|--|
| <b>EACCES</b> | The process does not have search permission on some component of the <i>pathname</i> .   |
| <b>EINVAL</b> | <p><i>varcode</i> is not a valid variable code, or the given variable cannot be associated with the specified file.</p> <ul style="list-style-type: none"> <li>• If <i>varcode</i> refers to MAX_CANON, MAX_INPUT, or _POSIX_VDISABLE, and <i>pathname</i> does not refer to a character special file, pathconf() returns -1 and sets errno to EINVAL.</li> <li>• If <i>varcode</i> refers to NAME_MAX, PATH_MAX, or POSIX_NO_TRUNC, and <i>pathname</i> does not refer to a directory, pathconf() returns the requested information.</li> <li>• If <i>varcode</i> refers to PC_PIPE_BUF and <i>pathname</i> refers to a pipe or a FIFO, the value returned applies to the referenced object itself. If <i>pathname</i> refers to a directory, the value returned applies to any FIFOs that exist or can be created within the directory. If <i>pathname</i> refers to any other type of file, the function sets errno to EINVAL.</li> </ul> |
| <b>ELOOP</b>  | A loop exists in symbolic links. This error is issued if more than POSIX_SYMLINKS symbolic links are detected in the resolution of <i>pathname</i> .   |

**ENAMETOOLONG**

*pathname* is longer than PATH\_MAX characters, or some component of *pathname* is longer than NAME\_MAX while \_POSIX\_NO\_TRUNC is in effect.

For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH\_MAX.

**ENOENT** There is no file named *pathname*, or the *pathname* argument is an empty string.

**ENOTDIR** Some component of the *pathname* is not a directory.

### Example

#### CBC3BP01

```

/* CBC3BP01
   This example determines the maximum number of characters in a file name.
*/
#define _POSIX_SOURCE
#include <errno.h>
#include <unistd.h>
#include <stdio.h>

main()
{
    long result;
    errno = 0;
    puts("examining NAME_MAX limit for root filesystem");
    if ((result = pathconf("/", _PC_NAME_MAX)) == -1)
        if (errno == 0)
            puts("There is no limit to NAME_MAX.");
        else perror("pathconf() error");
    else
        printf("NAME_MAX is %ld\n", result);
}

```

**Output**

```

examining NAME_MAX limit for root filesystem
NAME_MAX is 255

```

**Related Information**

- “unistd.h” on page 53
- “fpathconf() — Determine Configurable Path Name Variables” on page 427



## pause() — Suspend a Process Pending a Signal

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 POSIX.4a XPG4 XPG4.2	both	POSIX(ON) MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
int pause(void);
```

### General Description

Suspends execution of the calling thread. The thread does not resume execution until a signal is delivered, executing a signal handler or ending the thread. Some signals can be blocked by the process's *thread*. See “sigprocmask() — Examine or Change a Thread” on page 1339 for details.

If an incoming unblocked signal ends the thread, pause() never returns to the caller. If an incoming signal is handled by a signal handler, pause() returns after the signal handler returns.

### Returned Value

If pause() returns, it always returns a value of –1 and it sets errno to EINTR, indicating that a signal was received and handled successfully.

### Example

#### CBC3BP02

```
/* CBC3BP02
   This example suspends execution and determines the current time.
*/
#define _POSIX_SOURCE
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <time.h>

void catcher(int signum) {
    puts("inside catcher...");
}

void timestamp() {
    time_t t;
    time(&t);
    printf("the time is %s", ctime(&t));
}

main() {
    struct sigaction sigact;

    sigemptyset(&sigact.sa_mask);
    sigact.sa_flags = 0;
    sigact.sa_handler = catcher;
```

## pause

```
sigaction(SIGALRM, &sigact, NULL);

alarm(10);
printf("before pause... ");
timestamp();
pause();
printf("after pause... ");
timestamp();
}
```

### Output

before pause... the time is Fri Jun 16 09:42:29 1995  
inside catcher...

after pause... the time is Fri Jun 16 09:42:39 1995

### Related Information

- “unistd.h” on page 53
- “alarm() — Set an Alarm” on page 102
- “kill() — Send a Signal to a Process” on page 728
- “raise() — Raise Signal” on page 1074
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “signal() — Handle Interrupts” on page 1330
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351
- “wait() — Wait for a Child Process to End” on page 1687

## pclose() — Close a Pipe Stream to or from a Process

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	POSIX(ON) MVS 4.3

### Format

```
#define _XOPEN_SOURCE
#include <stdio.h>
```

```
int pclose(FILE *stream);
```

### General Description

The `pclose()` function closes a stream that was opened by `popen()`, waits for the command to terminate, and returns the status of the process that was running the shell command. However, if a call caused the termination status to be unavailable to `pclose()`, then `pclose()` returns -1 with `errno` set to `ECHILD` to report this situation; this can happen if the application calls one of the following functions:

- `wait()`
- `waitid()`
- `waitpid()` with a *pid* argument less than or equal to the process ID of the shell command
- any other function that could do one of the above

In any case, `pclose()` will not return before the child process created by `popen()` has terminated.

If the shell command cannot be executed, the child termination status returned by `pclose()` will be as if the shell command terminated using `exit(127)` or `_exit(127)`.

The `pclose()` function will not affect the termination status of any child of the calling process other than the one created by `popen()` for the associated stream.

If the argument *stream* to `pclose()` is not a pointer to a stream created by `popen()`, the termination status returned will be -1.

Threading Behavior: The `pclose()` function can be executed from any thread within the parent process.

### Returned Value

On successful completion, `pclose()` returns the termination status of the shell command. Otherwise, it returns -1 with the `errno` set to indicate the error.

The following are the possible values of `errno`:

**ECHILD**      The status of the child process could not be obtained.

### **Related Information**

- “stdio.h” on page 43
- “popen() — Initiate a Pipe Stream to or from a Process” on page 914
- “waitpid() — Wait for a Specific Child Process to End” on page 1692

perror() — Print Error Message

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

Format

```
#include <stdio.h>

void perror(const char *string);
```

X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdio.h>

void perror(const char *string);
```

Berkeley Sockets

```
#define _OE_SOCKETS
#include <stdio.h>

void perror(const char *string);
```

General Description

Prints an error message to stderr. If *string* is not NULL and it does not point to a null character, the string pointed to by *string* is printed to the standard error stream, followed by a colon and a space. The message associated with the value in *errno* is then printed followed by a new-line character. The content of the message is the same as the content of a string returned by *strerror()* with the argument *errno*.

To produce accurate results, you should ensure that *perror()* is called immediately after a library function returns with an error; otherwise, subsequent calls may alter the *errno* value.

If the error is associated with the *stderr* file, a call to *perror()* is not valid.

There is an environment variable *\_EDC\_ADD\_ERRNO2*, which when set to 1, will append the current *errno2* value to the end of the *perror()* string as shown.

EDC5121I Invalid argument. (errno2=0x0C0F8402)

|  
|

To provide an ASCII input/output format for applications using this function, define feature test macro *\_\_LIBASCII* as described on page 22.

Returned Value

Returns no value.

### Example

#### CBC3BP03

```

/* CBC3BP03
   This example tries to open a stream. If the fopen() function fails,
   the example prints a message and ends the program.
*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    FILE *fh;

    if ((fh = fopen("myfile.dat","r")) == NULL) {
        perror("Could not open data file");
        abort();
    }
}

```

The following example tries to open a stream socket. If the socket fails, the example prints a message and ends the program.

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/socket>

int main(void)
{
    int s;

    if ((s = socket (AF_INET,SOCK_STREAM,0)) <0)
    {
        perror("Could not open socket");
        exit(-1);
    }
}

```

### Related Information

- “stdio.h” on page 43
- “strerror() — Get Pointer to Runtime Error Message” on page 1427

## \_\_pid\_affinity() — Add or Delete Process Affinity

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R6

### Format

```
#define _OPEN_SYS
#include <unistd.h>
```

```
int __pid_affinity(int  function_code,
                  pid_t target_pid,
                  pid_t signal_pid,
                  int  signal);
```

### General Description

The `__pid_affinity()` function adds or deletes an entry in a process' affinity list. When a process terminates, each process in its affinity list is notified (sent a signal) of the termination. The `__pid_affinity()` function provides the ability to dynamically create or break an association between two processes that is similar to the notification mechanism between parent and child processes without the processes being related.

The *function\_code* can be set to one of the following symbolics, as defined in the `unistd.h` header file:

`__PAF_ADD_PID`

Add the process and signal specified by *signal\_pid* and *signal* to the affinity list of the process specified by *target\_pid*.

`__PAF_DELETE_PID`

Delete the process and signal specified by *signal\_pid* and *signal* from the affinity list of the process specified by *target\_pid*.

The *target\_pid* identifies the process whose affinity list will be altered.

The *signal\_pid* identifies the process that upon termination of the *target\_pid* will be sent *signal* signal.

The *signal* identifies the signal that the *signal\_pid* process will receive when the *target\_pid* process terminates.

### Usage Notes

1. Either the *Target\_Pid* or *Signal\_Pid* must contain the PID of the caller's process.
2. The `__pid` affinity service is limited to adding and deleting entries in the caller's affinity list, or adding and deleting entries that contain the caller's PID (*Signal\_Pid*) in other processes affinity list.
3. When the `PAF_DELETE_PID#` function is specified the *Signal* is ignored. It is not validated and may contain any value.
4. An entry is only deleted (`PAF_DELETE_PID#` specified) when the *Signal\_Pid* matches an entry in the *Target\_Pid* process' affinity list.

5. Entries with duplicate PIDs are not allowed in an affinity list. If adding an entry (PAF\_ADD\_PID# specified) and an entry with a PID that matches the *Signal\_Pid* is found the entry is reused. This may result in the loss of a specific signal.
6. No permission is required when adding the caller's PID to another process' affinity list. All processes have permission to send a signal to themselves (raise()).
7. The PIDs specified by the *Target\_Pid* and *Signal\_Pid* parameters must be greater than 1. Specifying a PID equal to or less than 1 will result in an error.

### Returned Value

The \_\_pid\_affinity() function returns 0 if the request is successful. If unsuccessful, it returns the value -1 and sets errno to one of the following:

Return_code	Explanation
EINVAL	One or more of the following conditions were detected: <ul style="list-style-type: none"><li>• The value specified by <i>Function_code</i> is not supported.</li><li>• The value specified by <i>Signal</i> is not a supported signal.</li><li>• <i>Target_Pid</i> does not contain a value greater than 1.</li><li>• <i>Signal_Pid</i> does not contain a value greater than 1.</li><li>• The <i>Signal_Pid</i> or <i>Target_Pid</i> does not specify the caller PID.</li></ul>
EMVSERR	A MVS environmental or internal error has occurred.
EMVSSAF2ERR	An internal SAF/RACF error has occurred.
EPERM	The caller does not have permission to send the signal to the <i>Signal_Pid</i> process.
ESRCH	One or more of the following conditions were detected: <ul style="list-style-type: none"><li>• No process corresponding to <i>Target_Pid</i> was found.</li><li>• No process corresponding to <i>Signal_Pid</i> was found.</li></ul>

### Related Information

- “unistd.h” on page 53
- “kill() — Send a Signal to a Process” on page 728



## pipe() — Create an Unnamed Pipe

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
int pipe(int fdinfo[2]);
```

### General Description

Creates a *pipe*, an I/O channel that a process can use to communicate with another process (in the same process or another process), or in some cases with itself. Data is written into one end of the pipe and read from the other. *fdinfo*[2] points to a memory area where pipe() can store two file descriptors. pipe() stores a file descriptor for the input end of the pipe in *fdinfo*[1], and stores a file descriptor for the output end of the pipe in *fdinfo*[0]. Thus, processes can read from *fdinfo*[0] and write to *fdinfo*[1]. Data written to *fdinfo*[1] is read from *fdinfo*[0] on a first-in-first-out basis.

When pipe() creates a pipe, the O\_NONBLOCK and FD\_CLOEXEC flags are turned off on both ends of the pipe. You can turn these flags on with fcntl(). See “fcntl() — Control Open File Descriptors” on page 350 for details.

If pipe() successfully creates a pipe, it updates the access, change, and modification times for the pipe.

It is unspecified whether *fdinfo*[0] is also open for writing and whether *fdinfo*[1] is also open for reading. OS/390 UNIX pipes are not STREAMS-based.

### Returned Value

If successful, pipe() returns the value 0. If unsuccessful, it returns the value –1 and sets errno to one of the following:

EMFILE	Opening the pipe would exceed the limit on the number of file descriptors the process can have open. This limit is given by OPEN_MAX, defined in the limits.h header file.
ENFILE	Opening the pipe would exceed the number of files that the system can have open simultaneously.
ENOMEM	Opening the pipe requires more space than is available.

## Example

### CBC3BP04

```

/* CBC3BP04
   This example creates an I/O channel.
   The output shows the data written into one end and read from the other.
*/
#define _POSIX_SOURCE
#include <unistd.h>
#include <stdio.h>

void reverse(char *s) {
    char *first, *last, temp;

    first = s;
    last = s+strlen(s)-1;
    while (first != last) {
        temp = *first;
        *(first++) = *last;
        *(last--) = temp;
    }
}

main() {
    char original[]="This is the original string";
    char buf[80];
    int  p1[2], p2[2];

    if (pipe(p1) != 0)
        perror("first pipe() failed");
    else if (pipe(p2) != 0)
        perror("second pipe() failed");
    else if (fork() == 0) {
        close(p1[1]);
        close(p2[0]);
        if (read(p1[0], buf, sizeof(buf)) == -1)
            perror("read() error in parent");
        else {
            reverse(buf);
            if (write(p2[1], buf, strlen(buf)+1) == -1)
                perror("write() error in child");
        }
        exit(0);
    }
    else {
        close(p1[0]);
        close(p2[1]);
        printf("parent is writing '%s' to pipe 1\n", original);
        if (write(p1[1], original, strlen(original)+1) == -1)
            perror("write() error in parent");
        else if (read(p2[0], buf, sizeof(buf)) == -1)
            perror("read() error in parent");
        else printf("parent read '%s' from pipe 2\n", buf);
    }
}

```

## Output

```

parent is writing 'This is the original string' to pipe 1
parent read 'gnirts lanigiro eht si sihT' from pipe 2

```

**Related Information**

- “unistd.h” on page 53
- “close() — Close a File” on page 191
- “fcntl() — Control Open File Descriptors” on page 350
- “open() — Open a File” on page 872
- “read() — Read From a File or Socket” on page 1080
- “write() — Write Data on a File or Socket” on page 1780

## poll() — Monitor Activity on File Descriptors and Message Queues

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### Sockets

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <poll.h>
```

```
int poll(struct pollfd fds[], nfds_t nmsgsfd, int timeout);
```

#### Message Queues and Sockets

```
#define _XOPEN_SOURCE_EXTENDED 1
#define _OPEN_MSGQ_EXT
#include <sys/types.h>
#include <sys/time.h>
#include <sys/msg.h>
#include <poll.h>
```

```
int poll(void *listptr, nmsgsfd_t nmsgsfd, int timeout);
```

`_OPEN_MSGQ_EXT` must be defined if message queues are to be monitored.

### General Description

The `poll()` function provides applications with a mechanism for multiplexing input/output over the following set of file descriptors:

- regular files
- terminal and pseudoterminal devices
- STREAMS-based files
- sockets
- message queues.
- FIFOs
- pipes

For each member of the array(s) pointed to by *listptr*, `poll()` examines the given file descriptor or message queue for the event(s) specified in the member. The number of `pollmsg` structures and the number of `pollfd` structures in the arrays are specified by *nmsgsfd*. The `poll()` function identifies those file descriptors on which an application can read or write data, or on which an error event has occurred.

**listptr**      A pointer to an array of `pollfd` structures, `pollmsg` structures, or to a `pollist` structure. Each structure specifies a file descriptor or message queue identifier and the events of interest for this file or message queue. The type of parameter to pass depends on whether you want to monitor file and socket descriptors, message queue identifiers, or both. To monitor socket descriptors only, set the high-order halfword of *nmsgsfd* to 0, the low-order halfword to the number of `pollfd` structures to be provided, and pass a pointer to an array of `pollfd` structures. To monitor message queues only, set the low-order halfword of *nmsgsfd* to 0, the high-order halfword to the number of `pollmsg`

structures to be provided, and pass a pointer to an array of pollmsg structures. To monitor both, set *nmsgsfds* as described below, and pass a pointer to a pollist structure. If a pollist structure is to be used, a structure similar to the following should be defined in a user program. The pollfd structure must precede the pollmsg structure.

```
struct pollist {
    struct pollfd fds[3];
    struct pollmsg msgids[2];
} list;
```

- nmsgsfds**    The number of pollmsg structures and the number of pollfd structures pointed to by *listptr*.
- This parameter is divided into two parts. The first half (the high-order 16 bits) gives the number of pollmsg structures containing message queue identifiers. This number must not exceed the value 32,767. The second half (the low-order 16 bits) gives the number of pollfd structures containing file descriptors to check. If either half of the *nmsgsfds* parameter is equal to a value of 0, the corresponding pollmsg structures or pollfd structures is assumed not to be present.
- timeout**    The amount of time, in milliseconds, to wait for an event to occur.
- If none of the defined events have occurred on any selected descriptor, poll() waits at least *timeout* milliseconds for an event to occur on any of the selected descriptors. If the value of *timeout* is 0, poll() returns immediately. If the value of *timeout* is -1, poll() blocks until a requested event occurs or until the call is interrupted.

The above processing also applies to message queues.

Each pollfd or pollmsg structure contains the following fields:

- fd/msgid - open file descriptor or message queue identifier
- events - requested events
- revents - returned events

The events and revents fields are bitmasks constructed by OR-ing a combination of the following event flags:

- POLLIN**            Same as POLLRDNORM
- POLLRDNORM**    Normal data may be read without blocking.
- POLLRDBAND**    Data from a nonzero priority band may be read without blocking. For STREAMS, this flag is set in revents even if the message is of zero length.
- POLLWRNORM**    Normal data may be written without blocking.
- POLLWRBAND**    Priority data (priority band greater than 0) may be written.
- POLLPRI**           Out-of-band data may be received without blocking.
- POLLOUT**           Same as POLLWRNORM
- POLLNVAL**        The specified fd/msgid value is invalid. This flag is only valid in the revents bitmask; it is ignored in the events bitmask.
- POLLERR**           An error or exceptional condition has occurred. This flag is only valid in the revents bitmask; it is ignored in the events bitmask.

**POLLHUP** The device has been disconnected. This event and **POLLOUT** are mutually exclusive, a stream can never be writable if a hang-up has occurred. However, this event and **POLLIN**, **POLLRDNORM**, **POLLRDBAND** or **POLLPRI** are not mutually exclusive. This flag is only valid in the revents bitmask. It is ignored in the events member.

**Note:** Poll bits are supported as follows.

#### Regular Files

Always `poll()` true for reading and writing. This means that all `poll()` read and write bits are supported. They will never return with **POLLERR** or **POLLHUP**.

#### FIFOs / PIPEs

Do not have the concept of out-of-band data or priority band data. They support **POLLIN**, **POLLRDNORM**, **POLLOUT**, and **POLLWRNORM**. They ignore **POLLPRI**, **POLLRDBAND**, and **POLLWRBAND**. They never return **POLLERR** or **POLLHUP**.

#### TTYs / OCS

Same support as FIFOs and PIPEs, except that TTYs may return **POLLERR**.

#### Sockets

Have the concept of out-of-band data. They support **POLLIN**, **POLLRDNORM**, **POLLOUT**, **POLLWRNORM**, and **POLLPRI** for out-of-band data. They ignore **POLLRDBAND** and **POLLWRBAND**. They never return **POLLHUP** or **POLLERR**.

If the value of `fd/msgid` is less than 0, events is ignored and revents is set to 0 in that entry on return from `poll()`.

In each `pollfd` structure, `poll()` clears the revents member except that where the application requested a report on a condition by setting one of the bits of events listed above, `poll()` sets the corresponding bit in revents if the requested condition is true. In addition, `poll()` sets the **POLLERR** flag in revents if the condition is true, even if the application did not set the corresponding bit in events.

The `poll()` function is not affected by the **O\_NONBLOCK** flag.

A file descriptor for a socket that is listening for connections will indicate that it is ready for reading, once connections are available. A file descriptor for a socket that is connecting asynchronously will indicate that it is ready for writing, once a connection has been established.

The following macros are provided to manipulate the *nmsgsfds* parameter and the return value from `poll()`:

Macro	Description
<code>_SET_FDS_MSGS(<i>nmsgsfds</i>, <i>nmsgs</i>, <i>nfds</i>)</code>	Sets the high-order halfword of <i>nmsgsfds</i> to <i>nmsgs</i> , and sets the low-order halfword of <i>nmsgsfds</i> to <i>nfds</i> .
<code>_NFDS(<i>n</i>)</code>	If the return value <i>n</i> from <code>poll()</code> is non-negative, returns the number of socket descriptors that meet the read, write, and exception criteria. A descriptor may be counted multiple times if it meets more than one given criterion.

`_NMSGs(n)` If the return value *n* from `poll()` is non-negative, returns the number of message queues that meet the read, write, and exception criteria. A message queue may be counted multiple times if it meets more than one given criterion.

## Returned Value

Upon successful completion, `poll()` returns a non-negative value. A positive value indicates the total number of events that were found to be ready among the message queues and the total number of events that were found to be ready among the file descriptors. The return value is similar to `nmsgsfds` in that the high-order 16 bits of the return value give the number associated with message queues, and the low-order 16 bits give the number associated with file descriptors. Should the number associated with message queues be greater than 32,767, only 32,767 will be reported. This is to ensure that the return value does not appear to be negative. Should the number associated with file descriptors be greater than 65,535, only 65,535 will be reported.

A value of 0 indicates that the call timed out and no file descriptors have been selected. Upon failure, `poll()` returns -1 and sets `errno` to indicate the error.

<code>EAGAIN</code>	The allocation of internal data structures failed, but a subsequent request may succeed.
<code>EINTR</code>	A signal was caught during <code>poll()</code> .
<code>EINVAL</code>	The <code>nmsgsfds</code> argument gives a number greater than the number of open files or the maximum number of allowable message queues.

## Related Information

- “`accept()` — Accept a New Connection on a Socket” on page 75
- “`connect()` — Connect a Socket” on page 214
- “`listen()` — Prepare the Server for Incoming Client Requests” on page 752
- “`msgctl()` — Message Control Operations” on page 845
- “`msgget()` — Get Message Queue” on page 847
- “`msgrcv()` — Message Receive Operation” on page 850
- “`msgsnd()` — Message Send Operations” on page 852
- “`read()` — Read From a File or Socket” on page 1080
- “`recv()` — Receive Data on a Socket” on page 1103
- “`select()` — Monitor Activity on Files/Sockets and Message Queues” on page 1159
- “`selectex()` — Monitor Activity on Files/Sockets and Message Queues” on page 1166
- “`send()` — Send Data on a Socket” on page 1178
- “`write()` — Write Data on a File or Socket” on page 1780

## popen() — Initiate a Pipe Stream to or from a Process

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	POSIX(ON)

### Format

```
#define _XOPEN_SOURCE
#include <stdio.h>
```

```
FILE *popen(const char *command, const char *mode);
```

### General Description

The `popen()` function executes the command specified by the string *command*. It creates a pipe between the calling program and the executed command, and returns a pointer to a stream that can be used to either read from or write to the pipe.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

The environment of the executed command will be as if a child process were created within the `popen()` call using `fork()`, and the child invoked the `sh` utility using the call:

```
execl("/bin/sh", "sh", "-c", command, (char *)0);
```

where shell path is an unspecified path name for the `sh` utility.

The `popen()` function ensures that any streams from previous `popen()` calls that remain open in the parent process are closed in the child process.

The *mode* argument to `popen()` is a string that specifies I/O mode:

1. If *mode* is **r**, file descriptor **STDOUT\_FILENO** will be the writable end of the pipe when the child process is started. The file descriptor *fileno(stream)* in the calling process, where *stream* is the stream pointer returned by `popen()`, will be the readable end of the pipe.
2. If *mode* is **w**, file descriptor **STDIN\_FILENO** will be the readable end of the pipe when the child process is started. The file descriptor *fileno(stream)* in the calling process, where *stream* is the stream pointer returned by `popen()`, will be the writable end of the pipe.
3. If *mode* is any other value, a null pointer is returned and `errno` is set to `EINVAL`.

After `popen()`, both the parent and the child process will be capable of executing independently before either terminates.

Because open files are shared, a mode *r* command can be used as an input filter and a mode *w* command as an output filter.



Buffered reading before opening an input filter (i.e. before `popen()`) may leave the standard input of that filter mispositioned. Similar problems with an output filter may be prevented by buffer flushing with `fflush()`.

A stream opened with `popen()` should be closed by `pclose()`.

The behavior of `popen()` is specified for values of *mode* of *r* and *w*. *mode* values of *rb* and *wb* are supported but are not portable.

If the shell command cannot be executed, the child termination status returned by `pclose()` will be as if the shell command terminated using `exit(127)` or `_exit(127)`.

If the application calls `waitpid()` with a *pid* argument greater than 0, and it still has a stream that was created with `popen()` open, it must ensure that *pid* does not refer to the process started by `popen()`

Threading Behavior: The `popen()` function cannot be called after the `pthread_create()` function has been called.

### Returned Value

On successful completion, `popen()` returns a pointer to an open stream that can be used to read or write to a pipe. Otherwise, it returns a null pointer with the `errno` set.

The following are the possible values of `errno`:

`EINVAL`      The *mode* argument is invalid.

The `popen()` function may also set *errno* values as described by `fork()` or `pipe()`.

### Related Information

- “stdio.h” on page 43
- “fflush() — Write Buffer to File” on page 385
- “fork() — Create a New Process” on page 422
- “pclose() — Close a Pipe Stream to or from a Process” on page 901
- “pipe() — Create an Unnamed Pipe” on page 907
- “system() — Execute a Command” on page 1488

## pow() — Raise to Power

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double pow(double x, double y);
```

### General Description

Calculates the value of  $x$  to the power of  $y$ .

### Returned Value

If  $y$  is 0, `pow()` returns the value 1. If  $x$  and  $y$  are negative, `pow()` sets `errno` to `EDOM` and returns `-HUGE_VAL`. If both  $x$  and  $y$  are 0, `pow()` sets `errno` to `EDOM` and returns 0. If  $x$  is negative and  $y$  is non-integral, `pow()` sets `errno` to `EDOM` and returns `-HUGE_VAL`. If the correct value is outside the range of representable values, plus or minus `HUGE_VAL` is returned according to the sign of the value and the value of `ERANGE` is stored in `errno`.

### Example CBC3BP05

```
/* CBC3BP05
   This example calculates the value of 2**3.
*/
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, y, z;

    x = 2.0;
    y = 3.0;
    z = pow(x,y);

    printf("%lf to the power of %lf is %lf\n", x, y, z);
}
```

### Output

```
2.000000 to the power of 3.000000 is 8.000000
```

### Related Information

- “`math.h`” on page 35
- “`exp()` — Calculate Exponential Function” on page 334
- “`log()` — Calculate Natural Logarithm” on page 762
- “`log10()` — Calculate Base 10 Logarithm” on page 767

**printf() — Format and Write Data**

The information for this function is included in “fprintf() - printf() - sprintf() — Format and Write Data” on page 436.

## pthread\_attr\_destroy() — Destroy the Thread Attributes Object

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_attr_destroy(pthread_attr_t *attr);
```

### General Description

Removes the definition of the thread attributes object. An error results if a thread attributes object is used after it has been destroyed.

*attr* is a pointer to a thread attribute object initialized by pthread\_attr\_init().

You can use a thread attribute object to manage the characteristics of threads in your application. It defines the set of values to be used for the thread during its creation. By establishing a thread attribute object, you can create many threads with the same set of characteristics, without defining those characteristics for each thread. You can define more than one thread attribute object.

If a thread attribute object is shared between threads, the application must provide the necessary synchronization because a thread attribute object is defined in the application's storage.

### Returned Value

If successful, pthread\_attr\_destroy() returns the value 0. If unsuccessful, it returns the value -1. There are no documented errno values for this function. Use perror() or strerror() to determine the cause of the error.

### Example

#### CBC3BP06

```
/* CBC3BP06 */
#define _OPEN_THREADS
#include <stdio.h>
#include <pthread.h>

void *thread1(void *arg) {
    pthread_exit(NULL);
}

int main()
{
    pthread_t      thid;
    pthread_attr_t attr;

    if (pthread_attr_init(&attr) == -1) {
        perror("error in pthread_attr_init");
        exit(1);
    }
}
```

```
if (pthread_create(&thid, &attr, thread1, NULL) == -1) {  
    perror("error in pthread_create");  
    exit(2);  
}  
  
if (pthread_detach(&thid) == -1) {  
    perror("error in pthread_detach");  
    exit(4);  
}  
  
if (pthread_attr_destroy(&attr) == -1) {  
    perror("error in pthread_attr_destroy");  
    exit(5);  
}  
exit(0);  
}
```

**Related Information**

- “pthread.h” on page 38
- “pthread\_attr\_init() — Initialize a Thread Attribute Object” on page 927
- “pthread\_create() — Create a Thread” on page 963

## pthread\_attr\_getdetachstate() — Get the Detach State Attribute

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_attr_getdetachstate(pthread_attr_t *attr);
```

### General Description

Returns the current value of the detachstate attribute for the thread attribute object, *attr*, that is created by `pthread_attr_init()`. The detachstate attribute values are:

- 0           Undetached. An undetached thread will keep its resources after termination.
- 1           Detached. A detached thread will have its resources automatically freed by the system at termination. Thus, you cannot get the thread's termination status (or wait for the thread to terminate) by using `pthread_join()`.

You can use a thread attribute object to manage the characteristics of threads in your application. It defines the set of values to be used for the thread during its creation. By establishing a thread attribute object, you can create many threads with the same set of characteristics, without defining those characteristics for each thread. You can define more than one thread attribute object.

### Returned Value

If successful, the detachstate is returned (0 or 1). If unsuccessful, it returns -1. There are no documented `errno`s for this function. Use `perror()` or `strerror()` to determine the cause of the error.

### Example

#### CBC3BP07

```
/* CBC3BP07 */
#define _OPEN_THREADS
#include <stdio.h>
#include <pthread.h>

int main()
{
    pthread_attr_t attr;
    char          typ[12];

    if (pthread_attr_init(&attr) == -1) {
        perror("error in pthread_attr_init");
        exit(1);
    }

    switch(pthread_attr_getdetachstate(&attr)) {
        default:
```

```

        perror("error in pthread_attr_getdetachstate()");
        exit(2);
    case 0:
        strcpy(typ, "undetached");
        break;
    case 1:
        strcpy(typ, "detached");
    }
    printf("The detach state is %s.\n", typ);

    if (pthread_attr_destroy(&attr) == -1) {
        perror("error in pthread_attr_destroy");
        exit(2);
    }
    exit(0);
}

```

### Output

The detach state is undetached.

### Related Information

- “pthread.h” on page 38
- “pthread\_attr\_init() — Initialize a Thread Attribute Object” on page 927
- “pthread\_attr\_setdetachstate() — Set the Detach State Attribute Object” on page 929
- “pthread\_create() — Create a Thread” on page 963

## pthread\_attr\_getstacksize() — Get the Thread Attribute Stacksize Object

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_attr_getstacksize(pthread_attr_t *attr, size_t *stacksize);
```

### General Description

Gets the value, in bytes, of the `stacksize` attribute for the thread attribute object, *attr*, that is created by `pthread_attr_init()`. This function returns the value in the variable pointed to by *stacksize*.

You can use a thread attribute object to manage the characteristics of threads in your application. It defines the set of values to be used for the thread during its creation. By establishing a thread attribute object, you can create many threads with the same set of characteristics, without defining those characteristics for each thread. You can define more than one thread attribute object.

### Returned Value

If successful, `pthread_attr_getstacksize()` returns the value 0 and stores the `stacksize` attribute value in *stacksize*. If unsuccessful, it returns the value -1. There are no documented errors for this function. Use `perror()` or `strerror()` to determine the cause of the error.

### Example CBC3BP08

```
/* CBC3BP08 */
#define _OPEN_THREADS
#include <stdio.h>
#include <pthread.h>

int main()
{
    pthread_attr_t attr;
    size_t size;

    if (pthread_attr_init(&attr) == -1) {
        perror("error in pthread_attr_init");
        exit(1);
    }

    if (pthread_attr_getstacksize(&attr, &size) == -1) {
        perror("error in pthread_attr_getstacksize()");
        exit(2);
    }
    printf("The stack size is %d.\n", (int) size);

    if (pthread_attr_destroy(&attr) == -1) {
```



```
        perror("error in pthread_attr_destroy");  
        exit(2);  
    }  
    exit(0);  
}
```

**Output**

The stack size is 524288.

**Related Information**

- “pthread.h” on page 38
- “pthread\_attr\_init() — Initialize a Thread Attribute Object” on page 927
- “pthread\_attr\_setstacksize() — Set the Stacksize Attribute Object” on page 931
- “pthread\_create() — Create a Thread” on page 963

## pthread\_attr\_getsynctype\_np() — Get Thread Sync Type

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_SYS
#include <pthread.h>
int pthread_attr_getsynctype_np(pthread_attr_t *attr);
```

### General Description

The `pthread_attr_getsynctype_np` function returns the current synctype setting of the *attr* thread attribute object.

The *synctype* can be set to one of the following symbolics, as defined in the `pthread.h` header file:

`__PTATSYNCHRONOUS`

Can only create as many threads as TCBs available (or as many threads are available, depending on which number is smaller).

`__PTATASYNCHRONOUS`

Allows threads to be queued, i.e. can create more threads than TCBs are available up to limit of how many threads are available. The queued threads will be released as TCBs become available.

### Returned Value

If successful, `pthread_attr_getsynctype_np` returns the synctype value of the thread attribute object.

If unsuccessful, `pthread_attr_getsynctype_np` returns -1. There are no documented `errno` values for this function. Use `perror()` or `strerror()` to determine cause of the error.

### Related Information

- “`pthread.h`” on page 38
- “`pthread_attr_init()` — Initialize a Thread Attribute Object” on page 927
- “`pthread_attr_setsynctype_np()` — Set Thread Sync Type” on page 933

## pthread\_attr\_getweight\_np() — Get Weight of Thread Attribute Object

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_attr_getweight_np(pthread_attr_t *attr);
```

### General Description

Obtains the current weight of the thread setting of the thread attributes object, *attr*. The symbols for weight are defined in the pthread.h include file. The following weights are supported:

**\_\_MEDIUM\_WEIGHT**

The executing task can be reused when the thread exits.

**\_\_HEAVY\_WEIGHT**

When this exits, the associated MVS task can no longer request threads to process.

You can use a thread attribute object to manage the characteristics of threads in your application. It defines the set of values to be used for the thread during its creation. By establishing a thread attribute object, you can create many threads with the same set of characteristics, without defining those characteristics for each and every thread. You can define more than one thread attribute object.

### Returned Value

If successful, pthread\_attr\_getweight\_np() returns the value of the weight of the thread attribute. If unsuccessful, it returns the value -1. There are no documented errors for this function. Use perror() or strerror() to determine the cause of the error.

### Example

#### CBC3BP09

```
/* CBC3BP09 */
#define _OPEN_THREADS

#include <stdio.h>
#include <pthread.h>

int main()
{
    pthread_attr_t attr;
    char          weight[12];

    if (pthread_attr_init(&attr) == -1) {
        perror("error in pthread_attr_init");
        exit(1);
    }
}
```

```
switch(pthread_attr_getweight_np(&attr)) {
    default:
        perror("error in pthread_attr_getweight_np()");
        exit(2);
    case __HEAVY_WEIGHT:
        strcpy(weight, "heavy");
        break;
    case __MEDIUM_WEIGHT:
        strcpy(weight, "medium");
}
printf("The thread weight is %s.\n", weight);

if (pthread_attr_destroy(&attr) == -1) {
    perror("error in pthread_attr_destroy");
    exit(2);
}
exit(0);
}
```

### Output

The thread weight is heavy.

### Related Information

- “pthread.h” on page 38
- “pthread\_attr\_init() — Initialize a Thread Attribute Object” on page 927
- “pthread\_attr\_setweight\_np() — Set Weight of Thread Attribute Object” on page 934
- “pthread\_create() — Create a Thread” on page 963

## pthread\_attr\_init() — Initialize a Thread Attribute Object

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_attr_init(pthread_attr_t *attr);
```

### General Description

Initializes *attr* with the default thread attributes, whose defaults are:

stacksize	Inherited from the STACK runtime option
detachstate	Undetached
synch	Synchronous
weight	Heavy

Using a thread attribute object, you can manage the characteristics of threads in your application. It defines the set of values to be used for the thread during its creation. By establishing a thread attribute object, you can create many threads with the same set of characteristics, without defining those characteristics for each thread. You can define more than one thread attribute object. All threads are of equal priority.

If a thread attribute object is shared between threads, the application must provide the necessary synchronization because a thread attribute object is defined in the application's storage.

### Returned Value

If successful, pthread\_attr\_init() returns the value 0. If unsuccessful, it returns the value -1 and sets errno to ENOMEM, which indicates that not enough memory is available to create the thread attribute object.

### Example

#### CBC3BP10

```
/* CBC3BP10 */
#define _OPEN_THREADS
#include <stdio.h>
#include <pthread.h>

void *thread1(void *arg)
{
    printf("hello from the thread\n");
    pthread_exit(NULL);
}

int main()
{
```

## pthread\_attr\_init

```
int          rc, stat;
pthread_attr_t attr;
pthread_t    thid;

rc = pthread_attr_init(&attr);
if (rc == -1) {
    perror("error in pthread_attr_init");
    exit(1);
}

rc = pthread_create(&thid, &attr, thread1, NULL);
if (rc == -1) {
    perror("error in pthread_create");
    exit(2);
}

rc = pthread_join(thid, (void *)&stat);
exit(0);
}
```

### Related Information

- “pthread.h” on page 38
- “pthread\_attr\_destroy() — Destroy the Thread Attributes Object” on page 918
- “pthread\_create() — Create a Thread” on page 963

## pthread\_attr\_setdetachstate() — Set the Detach State Attribute Object

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int *detachstate);
```

### General Description

Alters the current *detachstate* setting of a thread attributes object, which can be set to 0 or 1.

- 0 Causes all the threads created with *attr* to be in an undetached state. An undetached thread will keep its resources after termination.
- 1 Causes all the threads created with *attr* to be in a detached state. A detached thread will have its resources automatically freed by the system at termination. Thus, you cannot get the thread's termination status, or wait for the thread to terminate by using `pthread_join()`.

You can use a thread attribute object to manage the characteristics of threads in your application. It defines the set of values to be used for the thread during its creation. By establishing a thread attribute object, you can create many threads with the same set of characteristics, without defining those characteristics for each thread. You can define more than one thread attribute object.

### Returned Value

If successful, `pthread_attr_setdetachstate()` returns the value 0. If unsuccessful, it returns the value -1. There are no documented `errno`s for this function. Use `perror()` or `strerror()` to determine the cause of the error.

### Example

#### CBC3BP11

```
/* CBC3BP11 */
#define _OPEN_THREADS
#include <stdio.h>
#include <pthread.h>

void *thread1(void *arg)
void **stat;
{
    printf("hello from the thread\n");
    pthread_exit((void *)0);
}

int main()
{
    int          ds, rc, stat;
    size_t       sl;
    pthread_attr_t attr;
```

```
pthread_t      thid;

rc = pthread_attr_init(&attr);
if (rc == -1) {
    perror("error in pthread_attr_init");
    exit(1);
}

ds = 0;
rc = pthread_attr_setdetachstate(&attr, &ds);
if (rc == -1) {
    perror("error in pthread_attr_setdetachstate");
    exit(2);
}

rc = pthread_create(&thid, &attr, thread1, NULL);
if (rc == -1) {
    perror("error in pthread_create");
    exit(3);
}

rc = pthread_join(thid, stat);
exit(0);
}
```

### Related Information

- “pthread.h” on page 38
- “pthread\_attr\_getdetachstate() — Get the Detach State Attribute” on page 920
- “pthread\_attr\_init() — Initialize a Thread Attribute Object” on page 927
- “pthread\_create() — Create a Thread” on page 963



## pthread\_attr\_setstacksize() — Set the Stacksize Attribute Object

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

### General Description

Sets the stacksize, in bytes, for the thread attribute object, *attr*. *stacksize* is the initial stack size. Other stack characteristics, like stack increment size, are inherited from the STACK runtime option.

You can use a thread attribute object to manage the characteristics of threads in your application. It defines the set of values to be used for the thread during its creation. By establishing a thread attribute object, you can create many threads with the same set of characteristics, without defining those characteristics for each thread. You can define more than one thread attribute object.

### Returned Value

If successful, pthread\_attr\_setstacksize() returns the value 0. If unsuccessful, it returns the value -1. There are no documented errno's for this function. Use perror() or strerror() to determine the cause of the error.

### Example

#### CBC3BP12

```
/* CBC3BP12 */
#define _OPEN_THREADS
#include <stdio.h>
#include <pthread.h>

void *thread1(void *arg)
{
    printf("hello from the thread\n");
    pthread_exit(NULL);
}

int main()
{
    int          rc, stat;
    size_t       sl;
    pthread_attr_t attr;
    pthread_t     thid;

    rc = pthread_attr_init(&attr);
    if (rc == -1) {
        perror("error in pthread_attr_init");
        exit(1);
    }

    sl = 4096;
```

## pthread\_attr\_setstacksize

```
rc = pthread_attr_setstacksize(&attr, s1);
if (rc == -1) {
    perror("error in pthread_attr_setstacksize");
    exit(2);
}

rc = pthread_create(&thid, &attr, thread1, NULL);
if (rc == -1) {
    perror("error in pthread_create");
    exit(3);
}

rc = pthread_join(thid, (void *)&stat);
exit(0);
}
```

### Related Information

- “pthread.h” on page 38
- “pthread\_attr\_getstacksize() — Get the Thread Attribute Stacksize Object” on page 922
- “pthread\_attr\_init() — Initialize a Thread Attribute Object” on page 927
- “pthread\_create() — Create a Thread” on page 963

## pthread\_attr\_setsynctype\_np() — Set Thread Sync Type

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_SYS
#include <pthread.h>
int pthread_attr_setsynctype_np(pthread_attr_t *attr, int synctype);
```

### General Description

The pthread\_attr\_setsynctype\_np function allows you to alter the synctype setting of the *attr* thread attribute object.

The *synctype* can be set to one of the following symbolics, as defined in the pthread.h header file:

\_\_PTATSYNCHRONOUS

Can only create as many threads as TCBs available (or as many threads are available, depending on which number is smaller).

\_\_PTATASYNCHRONOUS

Allows threads to be queued, i.e. can create more threads than TCBs are available up to limit of how many threads are available. The queued threads will be released as TCBs become available. While threads are on the queue, they can still be affected by other pthread functions.

### Returned Value

If successful, pthread\_attr\_setsynctype\_np() returns 0.

If unsuccessful, pthread\_attr\_setsynctype\_np() returns -1. There are no documented errno values for this function. Use perror() or strerror() to determine cause of the error.

### Related Information

- “pthread.h” on page 38
- “pthread\_attr\_getsynctype\_np() — Get Thread Sync Type” on page 924
- “pthread\_attr\_init() — Initialize a Thread Attribute Object” on page 927
- “pthread\_attr\_setweight\_np() — Set Weight of Thread Attribute Object” on page 934

## pthread\_attr\_setweight\_np() — Set Weight of Thread Attribute Object

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_attr_setweight_np(pthread_attr_t *attr, int threadweight);
```

### General Description

Alter the current weight of the thread setting of the thread attribute object, *attr*.

*threadweight* can be set to one of the following two symbols for the weight of the thread, as defined in the pthread.h header file.

\_\_LIGHT\_WEIGHT

Not supported.

\_\_MEDIUM\_WEIGHT

Each thread runs on a task. Upon exiting, if another thread is not queued to run, the task waits for some other thread to issue a pthread\_create(), and the thread then runs on that task. The thread is assumed to cleanup all resources it used.

\_\_HEAVY\_WEIGHT

The task is attached on pthread\_create() and terminates upon a pthread\_exit(). Full MVS EOT resource cleanup occurs when exiting. When this exits, the associated MVS task can no longer request threads to process.

You can use a thread attribute object to manage the characteristics of threads in your application. It defines the set of values to be used for the thread during its creation. By establishing a thread attribute object, you can create many threads with the same set of characteristics, without defining those characteristics for each thread. You can define more than one thread attribute object.

### Returned Value

If successful, pthread\_attr\_setweight\_np() returns the value 0. If unsuccessful, it returns the value -1. There are no documented errno's for this function. Use perror() or strerror() to determine the cause of the error.

### Example

#### CBC3BP13

```
/* CBC3BP13 */
#define _OPEN_THREADS
#include <stdio.h>
#include <pthread.h>

void * thread1()
{
    printf("hello from the thread\n");
```

```

    pthread_exit((void *)0);
}

int main()
{
    int          rc, stat;
    pthread_attr_t attr;
    pthread_t     thid;

    rc = pthread_attr_init(&attr);
    if (rc == -1) {
        perror("error in pthread_attr_init");
        exit(1);
    }

    rc = pthread_attr_setweight_np(&attr, __MEDIUM_WEIGHT);
    if (rc == -1) {
        perror("error in pthread_attr_setweight_np");
        exit(2);
    }

    rc = pthread_create(&thid, &attr, thread1, NULL);
    if (rc == -1) {
        perror("error in pthread_create");
        exit(3);
    }

    rc = pthread_join(thid, (void *)&stat);
    exit(0);
}

```

### Related Information

- “pthread.h” on page 38
- “pthread\_attr\_getweight\_np() — Get Weight of Thread Attribute Object” on page 925
- “pthread\_attr\_init() — Initialize a Thread Attribute Object” on page 927
- “pthread\_create() — Create a Thread” on page 963

## pthread\_cancel() — Cancel a Thread

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_cancel(pthread_t thread);
```

### General Description

Requests that a thread be canceled. The thread to be canceled controls when this cancelation request is acted on through the cancelability state and type.

The cancelability states can be:

PTHREAD\_INTR\_DISABLE

The thread cannot be canceled.

PTHREAD\_INTR\_ENABLE

The thread can be canceled, but it is subject to type.

The cancelability types can be:

PTHREAD\_INTR\_CONTROLLED

The thread can be canceled, but only at specific points of execution:

- When waiting on a condition variable, which is `pthread_cond_wait()` or `pthread_cond_timedwait()`
- When waiting for the end of another thread, which is `pthread_join()`
- While waiting for an asynchronous signal, which is `sigwait()`
- Testing specifically for a cancel request, which is `pthread_testintr()`
- When suspended because of POSIX functions or one of the following C standard functions: `close()`, `fcntl()`, `open()`, `pause()`, `read()`, `tcdrain()`, `tcsetattr()`, `sigsuspend()`, `sigwait()`, `sleep()`, `wait()`, or `write()`

PTHREAD\_INTR\_ASYNCHRONOUS

The thread can be canceled at any time.

A thread that is joined on a thread that is canceled has a status of `-1` returned to it. For more information, refer to “`pthread_join()` — Wait for a Thread to End” on page 977.

*pthread\_t* is the data type used to uniquely identify a thread. It is returned by `pthread_create()` and used by the application in function calls that require a thread identifier.

### Special Behavior for C++

Destructors for automatic objects on the stack will be run when a thread is cancelled. The stack is unwound and the destructors are run in reverse order.

## Returned Value

If successful, `pthread_cancel()` returns the value 0. Success indicates that the `pthread_cancel()` request has been issued. The thread to be canceled may still execute because of its interruptibility state. If unsuccessful, `pthread_create()` returns the value -1 and sets `errno` to one of the following:

`EINVAL`      The specified thread is not valid.  
`ESRCH`        The specified thread does not refer to a currently existing thread.

## Example

### CBC3BP14

```
/* CBC3BP14 */
#define _OPEN_THREADS
#include <errno.h>
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int thstatus;

void * thread(void *arg)
{
    puts("thread has started. now sleeping");
    while (1)
        sleep(1);
}

main(int argc, char *argv[])
{
    pthread_t      thid;
    void           *status;

    if ( pthread_create(&thid, NULL, thread, NULL) != 0 ) {
        perror("pthread_create failed");
        exit(2);
    }

    if ( pthread_cancel(thid) == -1 ) {
        perror("pthread_cancel failed");
        exit(3);
    }

    if ( pthread_join(thid, &status) == -1 ) {
        perror("pthread_join failed");
        exit(4);
    }

    if ( status == (int *)-1 )
        puts("thread was cancelled");
    else
        puts("thread was not cancelled");

    exit(0);
}
```

### Output

```
thread has started. now sleeping  
thread was cancelled
```

### Related Information

- “pthread.h” on page 38
- “pthread\_cond\_timedwait() — Wait on a Condition Variable” on page 950
- “pthread\_cond\_wait() — Wait on a Condition Variable” on page 952
- “pthread\_exit() — Exit a Thread” on page 969
- “pthread\_join() — Wait for a Thread to End” on page 977
- “pthread\_setintr() — Set Thread's Cancelability State” on page 1035
- “pthread\_setintrtype() — Set Thread's Cancelability Type” on page 1038
- “pthread\_testintr() — Establish a Cancelability Point” on page 1047



## pthread\_cleanup\_pop() — Remove a Cleanup Handler

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
void pthread_cleanup_pop(int execute);
```

### General Description

Removes the specified *routine* in the last executed pthread\_cleanup\_push() statement from the top of the calling thread's cleanup stack.

The *execute* parameter specifies whether the cleanup routine that is popped should be run or just discarded. If the value is nonzero, the cleanup routine is executed.

pthread\_cleanup\_push() and pthread\_cleanup\_pop() must appear in pairs in the program within the same lexical scope, or undefined behavior will result.

When the thread ends, all pushed but not yet popped cleanup routines are popped from the cleanup stack and executed in last-in-first-out (LIFO) order. This occurs when the thread:

- Calls pthread\_exit()
- Does a return from the start routine (that gets controls as a result of a pthread\_create())
- Is canceled because of a pthread\_cancel()

### Returned Value

There is no return value. This function is used as a statement. If an error occurs while a pthread\_cleanup\_pop() statement is being processed, a termination condition is raised. There are no documented errno's for this function. Use perror() or strerror() to determine the cause of an error.

### Example CBC3BP15

```
/* CBC3BP15 */
#define _OPEN_THREADS

#include <pthread.h>
#include <stdio.h>

int iteration;

void noise_maker(void *arg) {
    printf("hello from noise_maker in iteration %d!\n", iteration);
}

void *thread(void *arg) {
    pthread_cleanup_push(noise_maker, NULL);
```

## pthread\_cleanup\_pop

```
pthread_cleanup_pop(iteration == 1 ? 0 : 1);
}

main() {
    pthread_t thid;

    for (iteration=1; iteration<=2; iteration++) {

        if (pthread_create(&thid, NULL, thread, NULL) != 0) {
            perror("pthread_create() error");
            exit(1);
        }

        if (pthread_detach(&thid) != 0) {
            perror("pthread_detach() error");
            exit(3);
        }
    }
}
```

### Output

hello from noise\_maker in iteration 2!

### Related Information

- “pthread.h” on page 38
- “pthread\_cancel() — Cancel a Thread” on page 936
- “pthread\_cleanup\_push() — Establish a Cleanup Handler” on page 941
- “pthread\_exit() — Exit a Thread” on page 969

## pthread\_cleanup\_push() — Establish a Cleanup Handler

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
void pthread_cleanup_push(void (*routine) (void *arg), void *arg);
```

### General Description

Pushes the specified *routine* onto the calling thread's cleanup stack. The cleanup handler is executed as a result of a `pthread_cleanup_pop()`, with a nonzero value for the *execute* parameter.

When the thread ends, all pushed but not yet popped cleanup routines are popped from the cleanup stack and executed in last-in-first-out (LIFO) order. This occurs when the thread:

- Calls `pthread_exit()`
- Does a return from the start routine
- Is canceled because of a `pthread_cancel()`

`pthread_cleanup_push()` and `pthread_cleanup_pop()` must appear in pairs and within the same lexical scope, or undefined behavior will result.

### Special Behavior for C++

Because C and C++ linkage conventions are incompatible, `pthread_cleanup_push()` cannot receive a C++ function pointer as the start routine function pointer. If you attempt to pass a C++ function pointer to `pthread_cleanup_push()`, the compiler will flag it as an error. You can pass a C or C++ function to `pthread_cleanup_push()` by declaring it as `extern "C"`.

### Returned Value

There is no return value. This function is used as a statement. If an error occurs while a `pthread_cleanup_push()` statement is being processed, a termination condition is raised. There are no documented `errno`s for this function. Use `perror()` or `strerror()` to determine the cause of an error.

### Example

#### CBC3BP16

```
/* CBC3BP16 */
#define _OPEN_THREADS

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int footprint=0;
```

```
void *thread(void *arg) {
    char *storage;

    if ((storage = (char*) malloc(80)) == NULL) {
        perror("malloc() failed");
        exit(6);
    }

    /* Plan to release storage even if thread doesn't exit normally */

    pthread_cleanup_push(free, storage);

    puts("thread has obtained storage and is waiting to be cancelled");
    footprint++;
    while (1)
        sleep(1);

    pthread_cleanup_pop(1);
}

main() {
    pthread_t thid;

    if (pthread_create(&thid, NULL, thread, NULL) != 0) {
        perror("pthread_create() error");
        exit(1);
    }

    while (footprint == 0)
        sleep(1);

    puts("IPT is cancelling thread");

    if (pthread_cancel(thid) != 0) {
        perror("pthread_cancel() error");
        exit(3);
    }

    if (pthread_join(thid, NULL) != 0) {
        perror("pthread_join() error");
        exit(4);
    }
}
```

### Output

```
thread has obtained storage and is waiting to be cancelled
IPT is cancelling thread
```

### Related Information

- “pthread.h” on page 38
- “pthread\_cancel() — Cancel a Thread” on page 936
- “pthread\_cleanup\_pop() — Remove a Cleanup Handler” on page 939
- “pthread\_exit() — Exit a Thread” on page 969

## pthread\_cond\_broadcast() — Broadcast a Condition

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

### General Description

Unblock all threads that are blocked on the specified condition variable, *cond*. If more than one thread is blocked, the order in which the threads are unblocked is unspecified.

pthread\_cond\_broadcast() has no effect if there are no threads currently blocked on *cond*.

### Returned Value

If successful, pthread\_cond\_broadcast() returns the value 0. If unsuccessful, it returns the value -1 and sets errno to EINVAL, which indicates that the value specified by *cond* does not refer to an initialized condition variable.

### Example

#### CBC3BP17

```
/* CBC3BP17 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

main() {
    pthread_cond_t cond;

    if (pthread_cond_init(&cond, NULL) != 0) {
        perror("pthread_cond_init() error");
        exit(1);
    }

    if (pthread_cond_broadcast(&cond) != 0) {
        perror("pthread_cond_broadcast() error");
        exit(2);
    }

    if (pthread_cond_destroy(&cond) != 0) {
        perror("pthread_cond_destroy() error");
        exit(3);
    }
}
```

### **Related Information**

- “pthread.h” on page 38
- “pthread\_cond\_init() — Initialize a Condition Variable” on page 947
- “pthread\_cond\_signal() — Signal a Condition” on page 948
- “pthread\_cond\_timedwait() — Wait on a Condition Variable” on page 950
- “pthread\_cond\_wait() — Wait on a Condition Variable” on page 952

## pthread\_cond\_destroy() — Destroy the Condition Variable Object

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

### General Description

Destroys the condition variable object specified by *cond*.

A condition variable object identifies a condition variable. Condition variables are used in conjunction with mutexes to protect shared resources.

### Returned Value

If successful, pthread\_cond\_destroy() returns the value 0. If unsuccessful, it returns the value -1 and sets errno to one of the following:

**EBUSY**      An attempt was made to destroy the object referenced by *cond* while it is referenced by another thread.

**EINVAL**      The value specified by *cond* is not valid.

### Example

#### CBC3BP18

```
/* CBC3BP18 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

main() {
    pthread_cond_t cond;

    if (pthread_cond_init(&cond, NULL) != 0) {
        perror("pthread_cond_init() error");
        exit(1);
    }

    if (pthread_cond_destroy(&cond) != 0) {
        perror("pthread_cond_destroy() error");
        exit(2);
    }
}
```

**Related Information**

- “pthread.h” on page 38
- “pthread\_cond\_broadcast() — Broadcast a Condition” on page 943
- “pthread\_cond\_init() — Initialize a Condition Variable” on page 947
- “pthread\_cond\_signal() — Signal a Condition” on page 948
- “pthread\_cond\_timedwait() — Wait on a Condition Variable” on page 950
- “pthread\_cond\_wait() — Wait on a Condition Variable” on page 952



## pthread\_cond\_init() — Initialize a Condition Variable

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *attr);
```

### General Description

Initializes the condition variable referenced by *cond* with attributes referenced by *attr*. If *attr* is NULL, the default condition variable attributes are used.

### Returned Value

If successful, pthread\_cond\_init() returns the value 0. If unsuccessful, it returns the value -1 and sets errno to ENOMEM, which indicates that there is not enough memory to initialize the condition variable.

### Example CBC3BP19

```
/* CBC3BP19 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

main() {
    pthread_cond_t cond;

    if (pthread_cond_init(&cond, NULL) != 0) {
        perror("pthread_cond_init() error");
        exit(1);
    }

    if (pthread_cond_destroy(&cond) != 0) {
        perror("pthread_cond_destroy() error");
        exit(2);
    }
}
```

### Related Information

- “pthread.h” on page 38
- “pthread\_condattr\_init() — Initialize a Condition Attribute Object” on page 959
- “pthread\_cond\_broadcast() — Broadcast a Condition” on page 943
- “pthread\_cond\_signal() — Signal a Condition” on page 948
- “pthread\_cond\_timedwait() — Wait on a Condition Variable” on page 950
- “pthread\_cond\_wait() — Wait on a Condition Variable” on page 952

## pthread\_cond\_signal() — Signal a Condition

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_cond_signal(pthread_cond_t *cond);
```

### General Description

Unblock at least one thread that is blocked on the specified condition variable, *cond*. If more than one thread is blocked, the order in which the threads are unblocked is unspecified.

pthread\_cond\_signal() will have no effect if there are no threads currently blocked on *cond*.

### Returned Value

If successful, pthread\_cond\_signal() returns the value 0. If unsuccessful, it returns the value -1 and sets errno to EINVAL, which indicates that the value specified by *cond* does not refer to an initialized condition variable.

### Example

#### CBC3BP20

```
/* CBC3BP20 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

main() {
    pthread_cond_t cond;

    if (pthread_cond_init(&cond, NULL) != 0) {
        perror("pthread_cond_init() error");
        exit(1);
    }

    if (pthread_cond_signal(&cond) != 0) {
        perror("pthread_cond_broadcast() error");
        exit(2);
    }

    if (pthread_cond_destroy(&cond) != 0) {
        perror("pthread_cond_destroy() error");
        exit(3);
    }
}
```

### **Related Information**

- “pthread.h” on page 38
- “pthread\_cond\_broadcast() — Broadcast a Condition” on page 943
- “pthread\_cond\_init() — Initialize a Condition Variable” on page 947
- “pthread\_cond\_timedwait() — Wait on a Condition Variable” on page 950
- “pthread\_cond\_wait() — Wait on a Condition Variable” on page 952

## pthread\_cond\_timedwait() — Wait on a Condition Variable

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                           const struct timespec *abstime);
```

### General Description

Allows a thread to wait on a condition variable until satisfied or until a specified time occurs. `pthread_cond_timedwait()` is the same as `pthread_cond_wait()` except it returns an error if the absolute time, specified by *abstime*, satisfies one of these conditions:

- Passes before *cond* is signaled or broadcasted
- Has already been passed at the time of the call

When such timeouts occur, `pthread_cond_timedwait()` reacquires the mutex, referenced by *mutex* (created by `pthread_mutex_init()`).

The two elements within the struct *timespec* are defined as follows:

tv_sec	The time to wait for the condition signal. It is expressed in seconds from midnight, January 1, 1970 UTC. The value specified must be greater than or equal to current calendar time expressed in seconds since midnight, January 1, 1970 UTC and less than 2,147,483,648 seconds.
tv_nsec	The time in nanoseconds to be added to tv_sec to determine when to stop waiting. The value specified must be greater than or equal to zero (0) and less than 1,000,000,000 (1,000 million).

### Returned Value

If successful, `pthread_cond_timedwait()` returns the value 0. If unsuccessful, it returns the value -1 and sets `errno` to one of the following:

EAGAIN	The time specified by <i>abstime</i> has passed.
EINVAL	Can be one of the following error conditions: <ul style="list-style-type: none"> <li>• The value specified by <i>cond</i> is not valid.</li> <li>• The value specified by <i>mutex</i> is not valid.</li> <li>• The value specified by <i>abstime</i> (tv_sec) is not valid.</li> <li>• The value specified by <i>abstime</i> (tv_nsec) is not valid.</li> <li>• Different mutexes were specified for concurrent operations on the same condition variable.</li> <li>• The mutex is not owned by the current thread.</li> </ul>

## Example

### CBC3BP21

```

/* CBC3BP21 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#include <errno.h>

main() {
    pthread_cond_t cond;
    pthread_mutex_t mutex;
    time_t T;
    struct timespec t;

    if (pthread_mutex_init(&mutex, NULL) != 0) {
        perror("pthread_mutex_init() error");
        exit(1);
    }

    if (pthread_cond_init(&cond, NULL) != 0) {
        perror("pthread_cond_init() error");
        exit(2);
    }

    if (pthread_mutex_lock(&mutex) != 0) {
        perror("pthread_mutex_lock() error");
        exit(3);
    }

    time(&T);
    t.tv_sec = T + 2;
    t.tv_nsec = 0;
    printf("starting timedwait at %s", ctime(&T));
    if (pthread_cond_timedwait(&cond, &mutex, &t) != 0)
        if (errno == EAGAIN)
            puts("wait timed out");
        else {
            perror("pthread_cond_timedwait() error");
            exit(4);
        }

    time(&T);
    printf("timedwait over at %s", ctime(&T));
}

```

## Output

```

starting timedwait at Fri Jun 16 10:44:00 1995
wait timed out
timedwait over at Fri Jun 16 10:44:02 1995

```

## Related Information

- “pthread.h” on page 38
- “pthread\_cond\_broadcast() — Broadcast a Condition” on page 943
- “pthread\_cond\_signal() — Signal a Condition” on page 948
- “pthread\_cond\_wait() — Wait on a Condition Variable” on page 952
- “pthread\_mutex\_init() — Initialize a Mutex Object” on page 988

## pthread\_cond\_wait() — Wait on a Condition Variable

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

### General Description

Blocks on a condition variable. It must be called with *mutex* locked by the calling thread, or undefined behavior will result. A mutex is locked using `pthread_mutex_lock()`.

*cond* is a condition variable that is shared by threads. To change it, a thread must hold the *mutex* associated with the condition variable. The `pthread_cond_wait()` function releases this *mutex* before suspending the thread and obtains it again before returning.

The `pthread_cond_wait()` function waits until a `pthread_cond_broadcast()` or a `pthread_cond_signal()` is received. For more information on these functions, refer to “`pthread_cond_broadcast()` — Broadcast a Condition” on page 943 and to “`pthread_cond_signal()` — Signal a Condition” on page 948.

### Returned Value

If successful, `pthread_cond_wait()` returns the value 0. If unsuccessful, it returns the value -1 and sets `errno` to `EINVAL`, which can indicate that:

- Different mutexes were specified for concurrent operations on the same condition variable.
- The mutex was not owned by the current thread.

### Example

#### CBC3BP22

```
/* CBC3BP22 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>

pthread_cond_t cond;
pthread_mutex_t mutex;

int footprint = 0;

void *thread(void *arg) {
    time_t T;

    if (pthread_mutex_lock(&mutex) != 0) {
        perror("pthread_mutex_lock() error");
```

```

        exit(6);
    }
    time(&T);
    printf("starting wait at %s", ctime(&T));
    footprint++;

    if (pthread_cond_wait(&cond, &mutex) != 0) {
        perror("pthread_cond_timedwait() error");
        exit(7);
    }
    time(&T);
    printf("wait over at %s", ctime(&T));
}

main() {
    pthread_t thid;
    time_t T;
    struct timespec t;

    if (pthread_mutex_init(&mutex, NULL) != 0) {
        perror("pthread_mutex_init() error");
        exit(1);
    }

    if (pthread_cond_init(&cond, NULL) != 0) {
        perror("pthread_cond_init() error");
        exit(2);
    }

    if (pthread_create(&thid, NULL, thread, NULL) != 0) {
        perror("pthread_create() error");
        exit(3);
    }

    while (footprint == 0)
        sleep(1);

    puts("IPT is about ready to release the thread");
    sleep(2);

    if (pthread_cond_signal(&cond) != 0) {
        perror("pthread_cond_signal() error");
        exit(4);
    }

    if (pthread_join(thid, NULL) != 0) {
        perror("pthread_join() error");
        exit(5);
    }
}

```

### Output

```

starting wait at Fri Jun 16 10:54:06 1995
IPT is about ready to release the thread
wait over at Fri Jun 16 10:54:09 1995

```

**Related Information**

- “pthread.h” on page 38
- “pthread\_cond\_broadcast() — Broadcast a Condition” on page 943
- “pthread\_cond\_signal() — Signal a Condition” on page 948
- “pthread\_cond\_timedwait() — Wait on a Condition Variable” on page 950



## pthread\_condattr\_destroy() — Destroy Condition Variable Attribute Object

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_condattr_destroy(pthread_condattr_t *attr);
```

### General Description

Destroys a condition attribute object. Condition-variable attribute objects are similar to mutex attribute objects because you can use them to manage the characteristics of condition variables in your application. They define the set of values to be used for the condition variable during its creation. However, there are currently no condition-attribute object characteristics that you can define.

pthread\_condattr\_init() is used to define a condition variable attribute object. pthread\_condattr\_destroy() is used to remove the definition of the condition variable attribute object. These functions are provided for portability purposes.

You can define a condition variable without using these functions by supplying a NULL parameter during the pthread\_cond\_init() call. For more details, refer to “pthread\_cond\_init() — Initialize a Condition Variable” on page 947.

### Returned Value

If successful, pthread\_condattr\_destroy() returns the value 0. If unsuccessful, it returns the value -1 and sets errno to EINVAL, which indicates that the value specified by attr is not valid.

### Example CBC3BP23

```
/* CBC3BP23 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

main() {
    pthread_condattr_t cond;

    if (pthread_condattr_init(&cond) != 0) {
        perror("pthread_condattr_init() error");
        exit(1);
    }

    if (pthread_condattr_destroy(&cond) != 0) {
        perror("pthread_condattr_destroy() error");
        exit(2);
    }
}
```

**Related Information**

- “pthread.h” on page 38
- “pthread\_condattr\_init() — Initialize a Condition Attribute Object” on page 959
- “pthread\_cond\_init() — Initialize a Condition Variable” on page 947
- “pthread\_mutex\_init() — Initialize a Mutex Object” on page 988

## pthread\_condattr\_getkind\_np() — Get Kind Attribute from a Condition Variable Attribute Object

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_condattr_getkind_np(pthread_condattr_t *attr, int *kind);
```

### General Description

Gets the attribute *kind* for the condition variable attribute object *attr*. Condition variable attribute objects are similar to mutex attribute objects. You can use them to manage the characteristics of condition variables in your application. They define the set of values for the condition variable during its creation.

The valid values for the attribute *kind* are:

**\_\_COND\_DEFAULT**

No defined attributes.

**\_\_COND\_NODEBUG**

State changes to this condition variable will *not* be reported to the debug interface, even though it is present.

### Returned Value

If successful, pthread\_condattr\_getkind\_np() returns the value 0. If unsuccessful, it returns the value -1 and sets errno to EINVAL, which indicates that the value specified for *attr* is not valid.

### Example

#### CBC3BP24

```
/* CBC3BP24 */
#pragma runopts(TEST(ALL))

#ifdef _OPEN_THREADS
#define _OPEN_THREADS
#endif

#include <stdio.h>
#include <pthread.h>

pthread_condattr_t attr;

int kind;

main() {
    if (pthread_condattr_init(&attr) == -1) {
        perror("pthread_condattr_init()");
        exit(1);
    }
}
```

```
if (pthread_condattr_setkind_np(ATTRIBUTE,; __COND_NODEBUG) == -1) {
    perror("pthread_condattr_setkind_np()");
    exit(1);
}

if (pthread_condattr_getkind_np(&attr, &kind) == -1) {
    exit(1);
}

switch(kind) {

    case __COND_DEFAULT:
        printf("\ncondition variable will have no defined attributes");
        break;

    case __COND_NODEBUG:
        printf("\ncondition variable will have nodebug attribute");
        break;

    default:
        printf("\nattribute kind value returned by \
pthread_condattr_getkind_np() unrecognized");

}

exit(0);
}
```

**Related Information**

- “pthread.h” on page 38
- “pthread\_condattr\_init() — Initialize a Condition Attribute Object” on page 959
- “pthread\_condattr\_setkind\_np() — Set Kind Attribute from a Condition Variable Attribute Object” on page 961

## pthread\_condattr\_init() — Initialize a Condition Attribute Object

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_condattr_init(pthread_condattr_t *attr);
```

### General Description

Establishes the default values for the condition variables that will be created. A condition attribute (*condattr*) object contains various condition variable characteristics. You can set up a template of these characteristics and then create a set of condition variables with similar characteristics.

Condition variable attribute objects are similar to mutex attribute objects. You can use them to manage the characteristics of condition variables in your application. They define the set of values to be used for the condition variable during its creation. However, there are currently no condition attribute object characteristics that can be defined.

`pthread_condattr_init()` is used to define a condition variable attribute object. `pthread_condattr_destroy()` is used to remove the definition of the condition variable attribute object. These functions are provided for portability purposes.

You can define a condition variable without using these functions by supplying a NULL parameter during the `pthread_cond_init()` call. For more details, refer to “`pthread_cond_init()` — Initialize a Condition Variable” on page 947.

### Returned Value

If successful, `pthread_condattr_init()` returns the value 0. If unsuccessful, it returns the value -1 and sets `errno` to `ENOMEM`, which indicates that there is not enough memory to initialize the condition variable attributes object.

### Example CBC3BP25

```
/* CBC3BP25 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

main() {
    pthread_condattr_t cond;

    if (pthread_condattr_init(&cond) != 0) {
        perror("pthread_condattr_init() error");
        exit(1);
    }

    if (pthread_condattr_destroy(&cond) != 0) {
```

## pthread\_condattr\_init

```
        perror("pthread_condattr_destroy() error");  
        exit(2);  
    }  
}
```

### Related Information

- “pthread.h” on page 38
- “pthread\_cond\_init() — Initialize a Condition Variable” on page 947
- “pthread\_mutex\_init() — Initialize a Mutex Object” on page 988

## pthread\_condattr\_setkind\_np() — Set Kind Attribute from a Condition Variable Attribute Object

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
int pthread_condattr_setkind_np(pthread_condattr_t *attr, int kind);
```

### General Description

Sets the attribute *kind* for the condition variable attribute object *attr*. Condition variable attribute objects are similar to mutex attribute objects. You can use them to manage the characteristics of condition variables in your application. They define the set of values to be used for the condition variable during its creation.

The valid values for the attribute *kind* are:

\_\_COND\_DEFAULT

No defined attributes.

\_\_COND\_NODEBUG

State changes to this condition variable will *not* be reported to the debug interface, even though it is present.

### Returned Value

If successful, pthread\_condattr\_setkind\_np() returns the value 0. If unsuccessful, it returns the value -1 and sets errno to EINVAL, which indicates the value specified for *attr* or *kind* is not valid.

### Example

#### CBC3BP26

```
/* CBC3BP26 */
#pragma runopts(TEST(ALL))

#ifdef _OPEN_THREADS
#define _OPEN_THREADS
#endif

#include <stdio.h>
#include <pthread.h>

pthread_condattr_t attr;

int kind;

main() {
    if (pthread_condattr_init(&attr) == -1) {
        perror("pthread_condattr_init()");
        exit(1);
    }

    if (pthread_condattr_setkind_np(ATTRIBUTE, __COND_NODEBUG) == -1) {
```

```
        perror("pthread_condattr_setkind_np()");
        exit(1);
    }

    if (pthread_condattr_getkind_np(&attr, &kind) == -1) {
        exit(1);
    }

    switch(kind) {

        case __COND_DEFAULT:
            printf("\ncondition variable will have no defined attributes");
            break;

        case __COND_NODEBUG:
            printf("\ncondition variable will have nodebug attribute");
            break;

        default:
            printf("\nattribute kind value returned by \
pthread_condattr_getkind_np() unrecognized");
    }

    exit(0);
}
```

**Related Information**

- “pthread.h” on page 38
- “pthread\_condattr\_init() — Initialize a Condition Attribute Object” on page 959
- “pthread\_condattr\_getkind\_np() — Get Kind Attribute from a Condition Variable Attribute Object” on page 957



## pthread\_create() — Create a Thread

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                  void * (*start_routine) (void *arg), void *arg);
```

### General Description

Creates a new thread within a process, with attributes defined by the thread attribute object, *attr*, that is created by `pthread_attr_init()`.

If *attr* is NULL, the default attributes are used. See “`pthread_attr_init()` — Initialize a Thread Attribute Object” on page 927 for a description of the thread attributes and their defaults. If the attributes specified by *attr* are changed later, the thread's attributes are not affected.

*pthread\_t* is the data type used to uniquely identify a thread. It is returned by `pthread_create()` and used by the application in function calls that require a thread identifier.

The thread is created running *start\_routine*, with *arg* as the only argument. If `pthread_create()` completes successfully, *thread* will contain the ID of the created thread. If it fails, no new thread is created, and the contents of the location referenced by *thread* are undefined.

System default for the thread limit in a process is set by MAXTHREADS in the BPXPRMxx parmlib member.

The maximum number of threads is dependent upon the size of the private area below 16M. `pthread_create()` inspects this address space prior to creating a new thread. A realistic limit is 200 to 400 threads.

### Special Behavior for C++

Because C and C++ linkage conventions are incompatible, `pthread_create()` cannot receive a C++ function pointer as the start routine function pointer. If you attempt to pass a C++ function pointer to `pthread_create()`, the compiler will flag it as an error. You can pass a C or C++ function to `pthread_create()` by declaring it as extern "C".

The started thread provides a boundary with respect to the scope of try-throw-catch processing. A throw done in the start routine or a function called by the start routine causes stack unwinding up to and including the start routine (or until caught). The stack unwinding will not go beyond the start routine back into the thread creator. If the exception is not caught, `terminate()` is called.

The exception stack (for try-throw-catch) are thread based. The throw of a condition, or re-throw of a condition by a thread does not affect exception processing on another thread, unless the condition is not caught.

## Returned Value

If successful, `pthread_create()` returns the value 0. If unsuccessful, it returns the value -1 and sets `errno` to one of the following:

<code>EAGAIN</code>	The system lacks the necessary resources to create another thread.
<code>EINVAL</code>	The value specified by <i>thread</i> is not valid.
<code>ENOMEM</code>	There is not enough memory to create the thread.

## Example

### CBC3BP27

```
/* CBC3BP27 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>
void *thread(void *arg) {
    char *ret;
    printf("thread() entered with argument '%s'\n", arg);
    if ((ret = (char*) malloc(20)) == NULL) {
        perror("malloc() error");
        exit(2);
    }
    strcpy(ret, "This is a test");
    pthread_exit(ret);
}

main() {
    pthread_t thid;
    void *ret;

    if (pthread_create(&thid, NULL, thread, "thread 1") != 0) {
        perror("pthread_create() error");
        exit(1);
    }

    if (pthread_join(thid, &ret) != 0) {
        perror("pthread_join() error");
        exit(3);
    }

    printf("thread exited with '%s'\n", ret);
}
```

## Output

```
thread() entered with argument 'thread 1'
thread exited with 'This is a test'
```

## Related Information

- “pthread.h” on page 38
- “pthread\_exit() — Exit a Thread” on page 969
- “pthread\_join() — Wait for a Thread to End” on page 977

## pthread\_detach() — Detach a Thread

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_detach(pthread_t *thread);
```

### General Description

Allows storage for the thread whose thread ID is in the location *thread* to be reclaimed when that thread ends. This storage is reclaimed on process exit, regardless of whether the thread was detached, and may include storage for *thread's* return value. If *thread* has not ended, pthread\_detach() will not cause it to end.

*pthread\_t* is the data type used to uniquely identify a thread. It is returned by pthread\_create() and used by the application in function calls that require a thread identifier.

### Returned Value

If successful, pthread\_detach() returns the value 0. If unsuccessful, it returns the value -1 and sets errno to one of the following:

**EINVAL**      The value specified by *thread* is not valid.

**ESRCH**        A value specified by *thread* refers to a thread that is already detached.

### Example

#### CBC3BP28

```
/* CBC3BP28 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

void *thread(void *arg) {
    char *ret;
    printf("thread() entered with argument '%s'\n", arg);
    if ((ret = (char*) malloc(20)) == NULL) {
        perror("malloc() error");
        exit(2);
    }
    strcpy(ret, "This is a test");
    pthread_exit(ret);
}

main() {
    pthread_t thid;
    void *ret;

    if (pthread_create(&thid, NULL, thread, "thread 1") != 0) {
        perror("pthread_create() error");
        exit(1);
    }
}
```

## pthread\_detach

```
    }  
  
    if (pthread_join(thid, &ret) != 0) {  
        perror("pthread_join() error");  
        exit(3);  
    }  
  
    printf("thread exited with '%s'\n", ret);  
}
```

### Output

```
thread() entered with argument 'thread 1'  
thread exited with 'This is a test'
```

### Related Information

- “pthread.h” on page 38
- “pthread\_join() — Wait for a Thread to End” on page 977

## pthread\_equal() — Compare Thread IDs

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

### General Description

Compares the thread IDs of *t1* and *t2*.

*pthread\_t* is the data type used to uniquely identify a thread. It is returned by `pthread_create()` and used by the application in function calls that require a thread identifier.

### Returned Value

If *t1* and *t2* are equal, `pthread_equal()` returns a positive value. Otherwise, the value 0 is returned. If *t1* or *t2* are not valid thread IDs, the behavior is undefined.

If `pthread_equal()` is unsuccessful, it returns the value `-1`. There are no documented errors for this function. Use `perror()` or `strerror()` to determine the cause of the error.

### Example

#### CBC3BP29

```
/* CBC3BP29 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

pthread_t thid, IPT;

void *thread(void *arg) {
    if (pthread_equal(IPT, thid))
        puts("the thread is the IPT...?");
    else
        puts("the thread is not the IPT");
}

main() {

    IPT = pthread_self();

    if (pthread_create(&thid, NULL, thread, NULL) != 0) {
        perror("pthread_create() error");
        exit(1);
    }

    if (pthread_join(thid, NULL) != 0) {
        perror("pthread_join() error");
        exit(3);
    }
}
```

```
}  
}
```

### Output

the thread is not the IPT

### Related Information

- “pthread.h” on page 38
- “pthread\_create() — Create a Thread” on page 963
- “pthread\_self() — Get the Caller” on page 1033

## pthread\_exit() — Exit a Thread

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
void pthread_exit(void *status);
```

### General Description

Ends the calling thread and makes *status* available to any thread that calls `pthread_join()` with the ending thread's thread ID.

As part of `pthread_exit()` processing, cleanup and destructor routines may be run:

- For details on the cleanup routines, refer to “`pthread_cleanup_pop()` — Remove a Cleanup Handler” on page 939 and “`pthread_cleanup_push()` — Establish a Cleanup Handler” on page 941.
- For details on the destructor routine, refer to “`pthread_key_create()` — Create Thread-Specific Data Key” on page 981.

### Special Behavior for C++

Destructors for automatic objects on the stack will be run when a thread is cancelled. The stack is unwound and the destructors are run in reverse order.

### Returned Value

This function cannot return to its caller. There are no documented errno's for this function. Use `perror()` or `strerror()` to determine the cause of the error.

### Example

#### CBC3BP30

```
/* CBC3BP30 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

void *thread(void *arg) {
    char *ret;

    if ((ret = (char*) malloc(20)) == NULL) {
        perror("malloc() error");
        exit(2);
    }
    strcpy(ret, "This is a test");
    pthread_exit(ret);
}

main() {
    pthread_t thid;
    void *ret;
```

## pthread\_exit

```
if (pthread_create(&thid, NULL, thread, NULL) != 0) {
    perror("pthread_create() error");
    exit(1);
}

if (pthread_join(thid, &ret) != 0) {
    perror("pthread_join() error");
    exit(3);
}

printf("thread exited with '%s'\n", ret);
}
```

### Output

thread exited with 'This is a test'

### Related Information

- “pthread.h” on page 38
- “pthread\_cleanup\_pop() — Remove a Cleanup Handler” on page 939
- “pthread\_cleanup\_push() — Establish a Cleanup Handler” on page 941
- “pthread\_create() — Create a Thread” on page 963
- “pthread\_exit() — Exit a Thread” on page 969
- “pthread\_join() — Wait for a Thread to End” on page 977
- “pthread\_key\_create() — Create Thread-Specific Data Key” on page 981



## pthread\_getspecific() — Get the Thread-Specific Value for a Key

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_getspecific(pthread_key_t key, void **value);
```

### General Description

Returns the thread-specific data associated with the specified *key* for the current thread. If no thread-specific data has been set for *key*, the null value is returned in *value*.

Many multithreaded applications require storage shared among threads, where each thread has its own unique value. A thread-specific data key is an identifier, created by a thread, for which each thread in the process can set a unique key *value*.

*pthread\_key\_t* is a storage area where the system places the key identifier. To create a key, a thread uses `pthread_key_create()`. This returns the key identifier into the storage area of type *pthread\_key\_t*. At this point, each of the threads in the application has the use of that key, and can set its own unique value by using `pthread_setspecific()`. A thread can get its own unique value using `pthread_getspecific()`.

### Returned Value

When unsuccessful, `pthread_getspecific()` sets `errno` to `EINVAL`, which indicates that the value for *key* is not valid.

### Example

#### CBC3BP31

```
/* CBC3BP31 */
#ifdef _OPEN_THREADS
#define _OPEN_THREADS
#endif

#include <stdio.h>
#include <errno.h>
#include <pthread.h>

#define threads 3
#define BUFFSZ 48
pthread_key_t key;

void *threadfunc(void *parm)
{
    int status;
    int value;
    int threadnum;
    int *tnum;
```

## pthread\_getspecific

```
void      *getvalue;
char      Buffer[BUFSZ];

tnum = parm;
threadnum = *tnum;

printf("Thread %d executing\n", threadnum);

if (!(value = malloc(sizeof(Buffer))))
    printf("Thread %d could not allocate storage, errno = %d\n",
           threadnum, errno);
status = pthread_setspecific(key, (void *) value);
if ( status < 0) {
    printf("pthread_setspecific failed, thread %d, errno %d",
           threadnum, errno);
    pthread_exit((void *)12);
}
printf("Thread %d setspecific value: %d\n", threadnum, value);

getvalue = 0;
status = pthread_getspecific(key, &getvalue);
if ( status < 0) {
    printf("pthread_getspecific failed, thread %d, errno %d",
           threadnum, errno);
    pthread_exit((void *)13);
}

if ((int)getvalue != value) {
    printf("getvalue not valid, getvalue=%d", (int)getvalue);
    pthread_exit((void *)68);
}

pthread_exit((void *)0);
}

void destr_fn(void *parm)
{
    printf("Destructor function invoked\n");
    if (free(parm))
        printf("unable to free storage, errno = %d\n", errno);
}

main() {
    int      getvalue;
    int      status;
    int      i;
    int      threadparm[threads];
    pthread_t threadid[threads];
    int      thread_stat[threads];

    if ((status = pthread_key_create(&key, destr_fn )) < 0) {
        printf("pthread_key_create failed, errno=%d", errno);
        exit(1);
    }

    /* create 3 threads, pass each its number */
    for (i=0; i<threads; i++) {
        threadparm[i] = i+1;
        status = pthread_create( &threadid[i],
                                NULL,
                                threadfunc,
                                (void *)&threadparm[i]);
    }
}
```

```

    if ( status < 0) {
        printf("pthread_create failed, errno=%d", errno);
        exit(2);
    }
}

for ( i=0; i<threads; i++) {
    status = pthread_join( threadid[i], (void *)&thread_stat[i]);
    if ( status < 0) {
        printf("pthread_join failed, thread %d, errno=%d\n", i+1, errno);
    }

    if (thread_stat[i] != 0) {
        printf("bad thread status, thread %d, status=%d\n", i+1,
            thread_stat[i]);
    }
}

exit(0);
}

```

### Related Information

- “pthread.h” on page 38
- “pthread\_getspecific\_d8\_np() — Get the Thread-Specific Value for a Key” on page 974
- “pthread\_key\_create() — Create Thread-Specific Data Key” on page 981
- “pthread\_setspecific() — Set the Thread-Specific Value for a Key” on page 1043

## pthread\_getspecific\_d8\_np() — Get the Thread-Specific Value for a Key

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
void *pthread_getspecific_d8_np(pthread_key_t key);
```

### General Description

Returns the thread-specific data associated with the specified *key* for the current thread. If no thread-specific data has been set for *key*, the null value is returned.

Many multithreaded applications require storage shared among threads, where each thread has its own unique value. A thread-specific data key is an identifier, created by a thread, for which each thread in the process can set a unique key *value*.

*pthread\_key\_t* is a storage area where the system places the key identifier. To create a key, a thread uses `pthread_key_create()`. This returns the key identifier into the storage area of type *pthread\_key\_t*. At this point, each of the threads in the application has the use of that key, and can set its own unique value by using `pthread_setspecific()`. A thread can get its own unique value using `pthread_getspecific_d8_np()` or `pthread_getspecific()`.

The only difference between `pthread_getspecific_d8_np()` and `pthread_getspecific()` is the syntax of the function.

### Returned Value

When successful, `pthread_getspecific_d8_np()` returns the thread-specific data value associated with *key*.

When unsuccessful, `pthread_getspecific_d8_np()` returns a NULL. Errno will also be set to EINVAL, when the value for *key* is not valid.

### Example

```
#ifndef _OPEN_THREADS
#define _OPEN_THREADS
#endif

#include <stdio.h>
#include <errno.h>
#include <pthread.h>

#define threads 3
#define BUFFSZ 48
pthread_key_t key;
```

```

void *threadfunc(void *);
void  destr_fn(void *);

main() {
    int          status;
    int          i;
    int          threadparm[threads];
    pthread_t     threadid[threads];
    int          thread_stat[threads];

    if ((status = pthread_key_create(&key, destr_fn )) < 0) {
        printf("pthread_key_create failed, errno=%d", errno);
        exit(1);
    }

    /* create 3 threads, pass each its number */
    for (i=0; i<threads; i++) {
        threadparm[i] = i+1;
        status = pthread_create( &threadid[i],
                                NULL,
                                threadfunc,
                                (void *)&threadparm[i]);

        if ( status < 0) {
            printf("pthread_create failed, errno=%d", errno);
            exit(2);
        }
    }

    for ( i=0; i<threads; i++) {
        status = pthread_join( threadid[i], (void *)&thread_stat[i]);
        if ( status < 0) {
            printf("pthread_join failed, thread %d, errno=%d\n", i+1, errno);
        }

        if (thread_stat[i] != 0) {
            printf("bad thread status, thread %d, status=%d\n", i+1,
                  thread_stat[i]);
        }
    }

    exit(0);
}

void *threadfunc(void *parm) {
    int          status;
    int          threadnum;
    int          *tnum;
    void         *getvalue;
    char         Buffer[BUFSZ];

    tnum = parm;
    threadnum = *tnum;

    printf("Thread %d executing\n", threadnum);

    if (!(value = malloc(sizeof(Buffer))))
        printf("Thread %d could not allocate storage, errno = %d\n",
              threadnum, errno);
    status = pthread_setspecific(key, (void *) value);
    if ( status < 0) {
        printf("pthread_setspecific failed, thread %d, errno %d",
              threadnum, errno);
    }
}

```

```
        pthread_exit((void *)12);
    }
    printf("Thread %d setspecific value: %d\n", threadnum, value);

    getvalue = pthread_getspecific_d8_np(key);
    if ( getvalue == NULL) {
        printf("pthread_getspecific_d8_np failed, thread %d, errno %d",
               threadnum, errno);
        pthread_exit((void *)13);
    }

    pthread_exit((void *)0);
}

void destr_fn(void *parm)
{
    printf("Destructor function invoked\n");
    if (free(parm))
        printf("unable to free storage, errno = %d\n", errno);
}
```

**Related Information**

- “pthread.h” on page 38
- “pthread\_key\_create() — Create Thread-Specific Data Key” on page 981
- “pthread\_setspecific() — Set the Thread-Specific Value for a Key” on page 1043
- “pthread\_getspecific() — Get the Thread-Specific Value for a Key” on page 971

## pthread\_join() — Wait for a Thread to End

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_join(pthread_t thread, void **status);
```

### General Description

Allows the calling thread to wait for the ending of the target *thread*.

*pthread\_t* is the data type used to uniquely identify a thread. It is returned by `pthread_create()` and used by the application in function calls that require a thread identifier.

*status* contains a pointer to the *status* argument passed by the ending thread as part of `pthread_exit()`. If the ending thread terminated with a return, *status* contains a pointer to the return value. If the thread was canceled, *status* can be set to the value `-1`.

### Returned Value

If successful, `pthread_join()` returns the value `0`. If unsuccessful, it returns the value `-1`, and sets `errno` to one of the following:

- `EINVAL`     The value specified by *thread* is not valid.
- `ESRCH`     The value specified by *thread* does not refer to an undetached thread.
- `EDEADLK`   A deadlock has been detected. This can occur if the target is directly or indirectly joined to the current thread.

### Notes:

1. When `pthread_join()` returns successfully, the target thread has been detached.
2. Multiple threads cannot use `pthread_join()` to wait for the same target thread to end. If a thread issues `pthread_join()` for a target thread after another thread has successfully issued `pthread_join()` for the same target thread, the second `pthread_join()` will be unsuccessful.
3. If the thread calling `pthread_join()` is canceled, the target thread is not detached.

### Example CBC3BP32

```
/* CBC3BP32 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>
```

```
void *thread(void *arg) {
    char *ret;
    printf("thread() entered with argument '%s'\n", arg);
    if ((ret = (char*) malloc(20)) == NULL) {
        perror("malloc() error");
        exit(2);
    }
    strcpy(ret, "This is a test");
    pthread_exit(ret);
}

main() {
    pthread_t thid;
    void *ret;

    if (pthread_create(&thid, NULL, thread, "thread 1") != 0) {
        perror("pthread_create() error");
        exit(1);
    }

    if (pthread_join(thid, &ret) != 0) {
        perror("pthread_join() error");
        exit(3);
    }

    printf("thread exited with '%s'\n", ret);
}
```

### Output

```
thread() entered with argument 'thread 1'
thread exited with 'This is a test'
```

### Related Information

- “pthread.h” on page 38
- “pthread\_create() — Create a Thread” on page 963
- “pthread\_cond\_wait() — Wait on a Condition Variable” on page 952
- “pthread\_detach() — Detach a Thread” on page 965



## pthread\_join\_d4\_np() — Wait for a Thread to End

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_SYS
#include <pthread.h>
```

```
int pthread_join_d4_np(pthread_t thread, void **status);
```

### General Description

Allows the calling thread to wait for the ending of the target *thread*.

*pthread\_t* is the data type used to uniquely identify a thread. It is returned by `pthread_create()` and used by the application in function calls that require a thread identifier.

*status* contains a pointer to the *status* argument passed by the ending thread as part of `pthread_exit()`. If the ending thread ended by a return, *status* contains a pointer to the return value. If the thread was canceled, *status* can be set to the value `-1`.

### Returned Value

If successful, `pthread_join_d4_np()` returns the value 0. If unsuccessful, it returns the value `-1`, and sets `errno` to one of the following:

- `EINVAL`     The value specified by *thread* is not valid.
- `ESRCH`     The value specified by *thread* does not refer to an undetached thread.
- `EDEADLK`   A deadlock has been detected. This can occur if the target is directly or indirectly joined to the current thread.

### Notes:

- When `pthread_join_d4_np()` returns successfully, the target thread has not been detached.
- Multiple threads can use `pthread_join_d4_np()` to wait for the same target thread to end.

### Example

#### CBC3BP33

```
/* CBC3BP33 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

void *thread(void *arg) {
    char *ret;
    printf("thread() entered with argument '%s'\n", arg);
    if ((ret = (char*) malloc(20)) == NULL) {
        perror("malloc() error");
```

```
        exit(2);
    }
    strcpy(ret, "This is a test");
    pthread_exit(ret);
}

main() {
    pthread_t thid;
    void *ret;

    if (pthread_create(&thid, NULL, thread, "thread 1") != 0) {
        perror("pthread_create() error");
        exit(1);
    }

    if (pthread_join_d4_np(thid, &ret) != 0) {
        perror("pthread_join_d4_np() error");
        exit(3);
    }

    printf("thread exited with '%s'\n", ret);

    if (pthread_detach(&thid) != 0) {
        perror("pthread_detach() error");
        exit(4);
    }
}
```

### Output

```
thread() entered with argument 'thread 1'
thread exited with 'This is a test'
```

### Related Information

- “pthread.h” on page 38
- “pthread\_create() — Create a Thread” on page 963
- “pthread\_cond\_wait() — Wait on a Condition Variable” on page 952
- “pthread\_detach() — Detach a Thread” on page 965

## pthread\_key\_create() — Create Thread-Specific Data Key

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_key_create(pthread_key_t *key, void (*destructor) (void *));
```

### General Description

Creates a key identifier, associated with *key*, and returns the key identifier into the storage area of type *pthread\_key\_t*. At this point, each of the threads in the application has the use of that key, and can set its own unique value by use of *pthread\_setspecific()*. A thread can get its own unique value using *pthread\_getspecific()*.

The *destructor* routine may be called when the thread ends. It is called when a non-null value has been set for the key for this thread, using *pthread\_setspecific()*, and the thread:

- Calls *pthread\_exit()*
- Does a return from the start routine
- Is canceled because of a *pthread\_cancel()* request.

When called, the destructor routine is passed the value bound to the key by the use of *pthread\_setspecific()*.

### Special Behavior for C++

Because C and C++ linkage conventions are incompatible, *pthread\_key\_create()* cannot receive a C++ function pointer as the start routine function pointer. If you attempt to pass a C++ function pointer to *pthread\_key\_create()*, the compiler will flag it as an error. You can pass a C or C++ function to *pthread\_key\_create()* by declaring it as extern "C".

### Returned Value

If successful, *pthread\_key\_create()* returns the value 0 and stores the newly created key identifier in *key*. If unsuccessful, it returns the value -1 and sets *errno* to one of the following:

- |        |   |
|--------|---|
| EAGAIN | There were not enough system resources to create another thread-specific data key, or the limit is exceeded for the total number of keys per process. |
| ENOMEM | There is not enough memory to create <i>key</i> .   |

## Example

### CBC3BP34

```

/* CBC3BP34 */
#ifdef _OPEN_THREADS
#define _OPEN_THREADS
#endif

#include <stdio.h>
#include <errno.h>
#include <pthread.h>

#define threads 3
#define BUFFSZ 48
pthread_key_t key;

void *threadfunc(void *parm)
{
    int status;
    int value;
    int threadnum;
    int *tnum;
    void *getvalue;
    char Buffer[BUFFSZ];

    tnum = parm;
    threadnum = *tnum;

    printf("Thread %d executing\n", threadnum);

    if (!(value = malloc(sizeof(Buffer))))
        printf("Thread %d could not allocate storage, errno = %d\n",
               threadnum, errno);
    status = pthread_setspecific(key, (void *) value);
    if (status < 0) {
        printf("pthread_setspecific failed, thread %d, errno %d",
               threadnum, errno);
        pthread_exit((void *)12);
    }
    printf("Thread %d setspecific value: %d\n", threadnum, value);

    getvalue = 0;
    status = pthread_getspecific(key, &getvalue);
    if (status < 0) {
        printf("pthread_getspecific failed, thread %d, errno %d",
               threadnum, errno);
        pthread_exit((void *)13);
    }

    if ((int)getvalue != value) {
        printf("getvalue not valid, getvalue=%d", (int)getvalue);
        pthread_exit((void *)68);
    }

    pthread_exit((void *)0);
}

void destr_fn(void *parm)
{
    printf("Destructor function invoked\n");
    if (free(parm))
        printf("unable to free storage, errno = %d\n", errno);
}

```

```

main() {
    int         getvalue;
    int         status;
    int         i;
    int         threadparm[threads];
    pthread_t    threadid[threads];
    int         thread_stat[threads];

    if ((status = pthread_key_create(&key, destr_fn )) < 0) {
        printf("pthread_key_create failed, errno=%d", errno);
        exit(1);
    }

    /* create 3 threads, pass each its number */
    for (i=0; i<threads; i++) {
        threadparm[i] = i+1;
        status = pthread_create( &threadid[i],
                                NULL,
                                threadfunc,
                                (void *)&threadparm[i]);

        if ( status < 0) {
            printf("pthread_create failed, errno=%d", errno);
            exit(2);
        }
    }

    for ( i=0; i<threads; i++) {
        status = pthread_join( threadid[i], (void *)&thread_stat[i]);
        if ( status < 0) {
            printf("pthread_join failed, thread %d, errno=%d\n", i+1, errno);
        }

        if (thread_stat[i] != 0) {
            printf("bad thread status, thread %d, status=%d\n", i+1,
                  thread_stat[i]);
        }
    }

    exit(0);
}

```

## Related Information

- “pthread.h” on page 38
- “pthread\_getspecific() — Get the Thread-Specific Value for a Key” on page 971
- “pthread\_getspecific\_d8\_np() — Get the Thread-Specific Value for a Key” on page 974
- “pthread\_setspecific() — Set the Thread-Specific Value for a Key” on page 1043

## pthread\_kill() — Send a Signal to a Thread

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <signal.h>
```

```
int pthread_kill(pthread_t thread, int sig);
```

### General Description

Directs a signal *sig* to the thread *thread*. The value of *sig* must be either 0 or one of the symbols defined in *signal.h*. (See Table 32 on page 1295 for a list of signals.) If *sig* is 0, *pthread\_kill()* performs error checking but does not send a signal.

*pthread\_t* is the data type used to uniquely identify a thread. It is returned by *pthread\_create()* and used by the application in function calls that require a thread identifier.

### Special Behavior for C++

If a thread is sent a signal using *pthread\_kill()* and that thread does not handle the signal, then destructors for local objects may not be executed.

### Returned Value

If successful, *pthread\_kill()* returns the value 0. If unsuccessful, it returns the value -1, sends no signal, and sets *errno* to *EINVAL*, which indicates one of the following:

- The thread ID specified by *thread* is not valid.
- The value of *sig* is incorrect or is not the number of a supported signal.

### Example CBC3BP35

```
/* CBC3BP35 */
#define _OPEN_THREADS

#include <errno.h>
#include <pthread.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void          *threadfunc(void *parm)
{
    int          threadnum;
    int          *tnum;
    sigset_t      set;

    tnum = parm;
    threadnum = *tnum;
```

```

printf("Thread %d executing\n", threadnum);
sigemptyset(&set);
if(sigaddset(&set, SIGUSR1) == -1) {
    perror("Sigaddset error");
    pthread_exit((void *)1);
}

if(sigwait(&set) != SIGUSR1) {
    perror("Sigwait error");
    pthread_exit((void *)2);
}

pthread_exit((void *)0);
}

main() {
    int          status;
    int          threadparm = 1;
    pthread_t    threadid;
    int          thread_stat;

    status = pthread_create( &threadid,
                            NULL,
                            threadfunc,
                            (void *)&threadparm);

    if ( status < 0 ) {
        perror("pthread_create failed");
        exit(1);
    }

    sleep(5);

    status = pthread_kill( threadid, SIGUSR1);
    if ( status < 0 )
        perror("pthread_kill failed");

    status = pthread_join( threadid, (void *)&thread_stat);
    if ( status < 0 )
        perror("pthread_join failed");

    exit(0);
}

```

## Related Information

- “pthread.h” on page 38
- “signal.h” on page 41
- “bsd\_signal() — BSD Version of signal()” on page 135
- “kill() — Send a Signal to a Process” on page 728
- “killpg() — Send a Signal to a Process Group” on page 731
- “pthread\_self() — Get the Caller” on page 1033
- “raise() — Raise Signal” on page 1074
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sighold() — Add a Signal to a Thread” on page 1322
- “sigignore() — Set Disposition to Ignore a Signal” on page 1323
- “signal() — Handle Interrupts” on page 1330
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “sigrelse() — Remove a Signal from a Thread” on page 1342
- “sigset() — Change a Signal Action and/or a Thread” on page 1343

## pthread\_mutex\_destroy() — Delete a Mutex Object

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

### General Description

Deletes a mutex object, which identifies a mutex. Mutexes are used to protect shared resources. *mutex* is set to an invalid value, but can be reinitialized using `pthread_mutex_init()`.

### Returned Value

If successful, `pthread_mutex_destroy()` returns the value 0. If unsuccessful, it returns the value -1 and sets `errno` to one of the following:

- EBUSY**      A request has detected an attempt to destroy the object referenced by *mutex* while it was locked or referenced by another thread (for example, while being used in a `pthread_cond_wait()` or `pthread_cond_timedwait()` function).
- EINVAL**      The value specified by *mutex* is not valid.

### Example

#### CBC3BP36

```
/* CBC3BP36 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

main() {
    pthread_mutex_t mutex;

    if (pthread_mutex_init(&mutex, NULL) != 0) {
        perror("pthread_mutex_init() error");
        exit(1);
    }

    if (pthread_mutex_destroy(&mutex) != 0) {
        perror("pthread_mutex_destroy() error");
        exit(2);
    }
}
```



**Related Information**

- “pthread.h” on page 38
- “pthread\_cond\_timedwait() — Wait on a Condition Variable” on page 950
- “pthread\_cond\_wait() — Wait on a Condition Variable” on page 952
- “pthread\_mutex\_init() — Initialize a Mutex Object” on page 988
- “pthread\_mutex\_lock() — Wait for a Lock on a Mutex Object” on page 990
- “pthread\_mutex\_trylock() — Attempt to Lock a Mutex Object” on page 992
- “pthread\_mutex\_unlock() — Unlock a Mutex Object” on page 994

## pthread\_mutex\_init() — Initialize a Mutex Object

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
```

### General Description

Creates a mutex, referenced by *mutex*, with attributes specified by *attr*. If *attr* is null, the default mutex attribute (NONRECURSIVE) is used.

### Returned Value

If successful, pthread\_mutex\_init() returns the value 0, and the state of the mutex becomes initialized and unlocked.

If unsuccessful, pthread\_mutex\_init() returns the value -1 and sets errno to one of the following:

- EAGAIN      The system lacked the necessary resources (other than memory) to initialize another mutex.
- EPERM      The caller does not have the privilege to perform the operation.
- EINVAL      The value specified by *attr* is not valid.
- ENOMEM      There is not enough memory to acquire a lock. This errno will only occur in the private path.

### Example

#### CBC3BP37

```
/* CBC3BP37 */
#ifdef _OPEN_THREADS
#define _OPEN_THREADS
#endif

#include <pthread.h>

main() {
    pthread_mutexattr_t attr;
    pthread_mutex_t mut;

    if (pthread_mutexattr_init(&attr) == -1) {
        perror("mutexattr_init error");
        exit(1);
    }

    if (pthread_mutex_init(&mut, &attr) == -1) {
        perror("mutex_init error");
        exit(2);
    }
}
```

```
    exit(0);  
}
```

### Related Information

- “pthread.h” on page 38
- “pthread\_cond\_init() — Initialize a Condition Variable” on page 947
- “pthread\_cond\_timedwait() — Wait on a Condition Variable” on page 950
- “pthread\_cond\_wait() — Wait on a Condition Variable” on page 952
- “pthread\_mutexattr\_init() — Initialize a Mutex Attribute Object” on page 1004
- “pthread\_mutex\_lock() — Wait for a Lock on a Mutex Object” on page 990
- “pthread\_mutex\_trylock() — Attempt to Lock a Mutex Object” on page 992
- “pthread\_mutex\_unlock() — Unlock a Mutex Object” on page 994

## pthread\_mutex\_lock() — Wait for a Lock on a Mutex Object

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

### General Description

Locks a mutex object, which identifies a mutex. Mutexes are used to protect shared resources. If the mutex is already locked by another thread, the thread waits for the mutex to become available. The thread that has locked a mutex becomes its current owner and remains the owner until the same thread has unlocked it.

When the mutex has the attribute of recursive the use of the lock may be different. When this kind of mutex is locked multiple times by the same thread, then a count is incremented and no waiting thread is posted. The owning thread must call pthread\_mutex\_unlock() the same number of times to decrement the count to zero.

**Note:** If a thread owns mutex at the time it is terminated then OS/390 UNIX will release those locks.

The mutex types are described below:

#### PTHREAD\_MUTEX\_NORMAL

A normal type mutex does not detect deadlock. That is, a thread attempting to relock this mutex without first unlocking it will deadlock. The mutex is either in a locked or unlocked state for a thread.

#### PTHREAD\_MUTEX\_ERRORCHECK

An errorcheck type mutex provides error checking. That is, a thread attempting to relock this mutex without first unlocking it will return with an error. The mutex is either in a locked or unlocked state for a thread. If a thread attempts to lock a mutex that it has already locked, it will return with an error. If a thread attempts to lock a mutex that another thread has already locked, it will return with an error. If a thread attempts to lock a mutex that is unlocked, it will return with an error.

#### PTHREAD\_MUTEX\_RECURSIVE

A recursive type mutex permits a thread to lock many times. That is, a thread attempting to relock this mutex without first unlocking will succeed. This type of mutex must be unlocked the same number of times it is locked before the mutex will be returned to an unlocked state. If locked, an error is returned.

#### PTHREAD\_MUTEX\_DEFAULT

The default type mutex is mapped to a normal type mutex which does not detect deadlock. That is, a thread attempting to relock this mutex without first unlocking it will deadlock. The mutex is either in a locked or unlocked state for a thread. The normal mutex is the default type mutex.

## Returned Value

If successful, `pthread_mutex_lock()` returns the value 0. If unsuccessful, it returns the value `-1` and sets `errno` to one of the following:

- |                |  |
|----------------|--|
| <b>EAGAIN</b>  | The mutex could not be acquired because the maximum number of recursive locks for mutex has been exceeded. This <code>errno</code> will only occur in the shared path. |
| <b>EINVAL</b>  | The value specified by <i>mutex</i> is not valid.  |
| <b>EDEADLK</b> | The current thread already owns the mutex, and the mutex has a <i>kind</i> attribute of <code>__MUTEX_NONRECURSIVE</code> .  |

## Example

### CBC3BP38

```
/* CBC3BP38 */
#ifdef _OPEN_THREADS
#define _OPEN_THREADS
#endif

#include <pthread.h>
#include <stdio.h>

main() {
    pthread_mutex_t mut;

    if (pthread_mutex_init(&mut, NULL) != 0) {
        perror("mutex_lock");
        exit(1);
    }

    if (pthread_mutex_lock(&mut) != 0) {
        perror("mutex_lock");
        exit(2);
    }

    puts("the mutex has been locked");
    exit(0);
}
```

## Related Information

- “pthread.h” on page 38
- “pthread\_cond\_init() — Initialize a Condition Variable” on page 947
- “pthread\_cond\_wait() — Wait on a Condition Variable” on page 952
- “pthread\_mutex\_destroy() — Delete a Mutex Object” on page 986
- “pthread\_mutex\_init() — Initialize a Mutex Object” on page 988

## pthread\_mutex\_trylock() — Attempt to Lock a Mutex Object

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

### General Description

Locks a mutex object, which identifies a mutex. Mutexes are used to protect shared resources. If pthread\_mutex\_trylock() is locked, it returns immediately.

For recursive mutexes, pthread\_mutex\_trylock() will effectively add to the count of the number of times pthread\_mutex\_unlock() must be called by the thread to release the mutex. (That is, it has the same behavior as a pthread\_mutex\_lock().)

### Returned Value

If successful, pthread\_mutex\_trylock() returns the value 0. If unsuccessful, it returns the value -1 and sets errno to one of the following:

- EAGAIN**      The mutex could not be acquired because the maximum number of recursive locks for mutex has been exceeded. This errno will only occur in the shared path.
- EBUSY**      *mutex* could not be acquired because it was already locked.
- EINVAL**      The value specified by *mutex* is not valid.

### Example CBC3BP40

```
/* CBC3BP40 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>
#include <errno.h>

pthread_mutex_t mutex;

void *thread(void *arg) {
    if (pthread_mutex_trylock(&mutex) != 0)
        if (errno == EBUSY)
            puts("thread was denied access to the mutex");
        else {
            perror("pthread_mutex_trylock() error");
            exit(1);
        }
    else puts("thread was granted the mutex");
}

main() {
    pthread_t thid;
```

```

if (pthread_mutex_init(&mutex, NULL) != 0) {
    perror("pthread_mutex_init() error");
    exit(2);
}

if (pthread_create(&thid, NULL, thread, NULL) != 0) {
    perror("pthread_create() error");
    exit(3);
}
if (pthread_mutex_trylock(&mutex) != 0)
    if (errno == EBUSY)
        puts("IPT was denied access to the mutex");
    else {
        perror("pthread_mutex_trylock() error");
        exit(4);
    }
else puts("IPT was granted the mutex");

if (pthread_join(thid, NULL) != 0) {
    perror("pthread_join() error");
    exit(5);
}
}

```

### Output

```

IPT was granted the mutex
thread was denied access to the mutex

```

### Related Information

- “pthread.h” on page 38
- “pthread\_mutex\_destroy() — Delete a Mutex Object” on page 986
- “pthread\_mutex\_init() — Initialize a Mutex Object” on page 988

## pthread\_mutex\_unlock() — Unlock a Mutex Object

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

### General Description

Releases a mutex object. If one or more threads are waiting to lock the mutex, `pthread_mutex_unlock()` causes one of those threads to return from `pthread_mutex_lock()` with the mutex object acquired. If no threads are waiting for the mutex, the mutex unlocks with no current owner.

When the mutex has the attribute of recursive the use of the lock may be different. When this kind of mutex is locked multiple times by the same thread, then unlock will decrement the count and no waiting thread is posted to continue running with the lock. If the count is decremented to zero, then the mutex is released and if any thread is waiting it is posted.

### Returned Value

If successful, `pthread_mutex_unlock()` returns the value 0. If unsuccessful, `pthread_mutex_unlock()` returns the value -1 and sets `errno` as follows:

`EINVAL`      The value specified by *mutex* is not valid  
`EPERM`        The current thread does not own the *mutex*

### Example

#### CBC3BP41

```
/* CBC3BP41 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>
#include <errno.h>

pthread_mutex_t mutex;

void *thread(void *arg) {
    if (pthread_mutex_lock(&mutex) != 0) {
        perror("pthread_mutex_lock() error");
        exit(1);
    }

    puts("thread was granted the mutex");

    if (pthread_mutex_unlock(&mutex) != 0) {
        perror("pthread_mutex_unlock() error");
        exit(2);
    }
}
```



```

main() {
    pthread_t thid;

    if (pthread_mutex_init(&mutex, NULL) != 0) {
        perror("pthread_mutex_init() error");
        exit(3);
    }

    if (pthread_create(&thid, NULL, thread, NULL) != 0) {
        perror("pthread_create() error");
        exit(4);
    }
    if (pthread_mutex_lock(&mutex) != 0) {
        perror("pthread_mutex_lock() error");
        exit(5);
    }

    puts("IPT was granted the mutex");

    if (pthread_mutex_unlock(&mutex) != 0) {
        perror("pthread_mutex_unlock() error");
        exit(6);
    }

    if (pthread_join(thid, NULL) != 0) {
        perror("pthread_join() error");
        exit(7);
    }
}

```

### Output

```

IPT was granted the mutex
thread was granted the mutex

```

### Related Information

- “pthread.h” on page 38
- “pthread\_mutex\_destroy() — Delete a Mutex Object” on page 986
- “pthread\_mutex\_init() — Initialize a Mutex Object” on page 988
- “pthread\_mutex\_lock() — Wait for a Lock on a Mutex Object” on page 990

## pthread\_mutexattr\_destroy() — Destroy a Mutex Attribute Object

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_mutexattr_destroy (pthread_mutexattr_t *attr);
```

### General Description

Destroys an initialized mutex attribute object. With a mutex attribute object, you can manage the characteristics of mutexes in your application. It defines the set of values to be used for the mutex during its creation. By establishing a mutex attribute object, you can create many mutexes with the same set of characteristics, without defining those characteristics for each mutex. `pthread_mutexattr_init()` is used to define a mutex attribute object.

### Returned Value

If successful, `pthread_mutexattr_destroy()` returns the value 0. If unsuccessful, it returns the value -1 and sets `errno` to `EINVAL`, which indicates that the value specified for `attr` is not valid.

### Example

#### CBC3BP42

```
/* CBC3BP42 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

main() {
    pthread_mutexattr_t attr;
    pthread_mutex_t mutex;

    if (pthread_mutexattr_init(&attr) != 0) {
        perror("pthread_mutex_attr_init() error");
        exit(1);
    }

    if (pthread_mutexattr_setkind_np(ATTRIBUTE, __MUTEX_RECURSIVE) != 0) {
        perror("pthread_mutex_attr_setkind_np() error");
        exit(2);
    }

    if (pthread_mutex_init(&mutex, &attr) != 0) {
        perror("pthread_mutex_init() error");
        exit(3);
    }

    if (pthread_mutexattr_destroy(&attr) != 0) {
        perror("pthread_mutex_attr_destroy() error");
        exit(4);
    }
}
```

```
}  
}
```

### **Related Information**

- “pthread.h” on page 38
- “pthread\_cond\_init() — Initialize a Condition Variable” on page 947
- “pthread\_create() — Create a Thread” on page 963
- “pthread\_mutexattr\_init() — Initialize a Mutex Attribute Object” on page 1004

## pthread\_mutexattr\_getkind\_np() — Get Kind from a Mutex Attribute Object

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_mutexattr_getkind_np(pthread_mutexattr_t *attr, int *kind);
```

### General Description

Gets the attribute *kind* from the mutex attribute object *attr*. With a mutex attribute object, you can manage the characteristics of mutexes in your application. It defines the set of values to be used for the mutex during its creation. By establishing a mutex attribute object, you can create many mutexes with the same set of characteristics, without defining those characteristics for each and every mutex.

The values for the attribute *kind* are:

#### \_\_MUTEX\_NONRECURSIVE

A nonrecursive mutex can be locked only once. That is, the mutex is either in a locked or unlocked state for a thread. If a thread attempts to lock a mutex that it has already locked, an error is returned.

#### \_\_MUTEX\_RECURSIVE

A recursive mutex can be locked more than once by the same thread. A count of the number of times the mutex has been locked is maintained. The mutex is unlocked when `pthread_mutex_unlock()` is performed an equal number of times.

#### \_\_MUTEX\_NONRECURSIVE + \_\_MUTEX\_NODEBUG

A nonrecursive mutex can be given an additional attribute, `NODEBUG`. This indicates that state changes to this mutex will *not* be reported to the debug interface, even if present.

#### \_\_MUTEX\_RECURSIVE + \_\_MUTEX\_NODEBUG

A recursive mutex can be given an additional attribute, `NODEBUG`. This indicates that state changes to this mutex will *not* be reported to the debug interface, even if present.

### Returned Value

If successful, `pthread_mutexattr_getkind_np()` returns the value 0. If unsuccessful, it returns the value -1 and sets `errno` `EINVAL`, which indicates that the value specified for *attr* is not valid.

## Example

### CBC3BP43

```

/* CBC3BP43 */
#pragma runopts(TEST(ALL))

#ifdef _OPEN_THREADS
#define _OPEN_THREADS
#endif

#include <stdio.h>
#include <pthread.h>

pthread_mutexattr_t attr;

int kind;

main() {
    if (pthread_mutexattr_init(&attr) == -1) {
        perror("pthread_mutexattr_init()");
        exit(1);
    }

    if (pthread_mutexattr_setkind_np(&attr, \
        __MUTEX_RECURSIVE + __MUTEX_NODEBUG) == -1 ) {

        perror("pthread_mutexattr_setkind_np()");
        exit(1);
    }

    if (pthread_mutexattr_getkind_np(&attr, &kind) == -1) {
        perror("pthread_mutexattr_getkind_np()");
        exit(1);
    }

    switch(kind) {

        case __MUTEX_NONRECURSIVE:
            printf("\nmutex will be nonrecursive");
            break;

        case __MUTEX_NONRECURSIVE+__MUTEX_NODEBUG:
            printf("\nmutex will be nonrecursive + nodebug");
            break;

        case __MUTEX_RECURSIVE:
            printf("\nmutex will be recursive");
            break;

        case __MUTEX_RECURSIVE+__MUTEX_NODEBUG:
            printf("\nmutex will be recursive + nodebug");
            break;

        default:
            printf("\nattribute kind value returned by \
pthread_mutexattr_getkind_np() unrecognized");
            exit(1);
    }
    exit(0);
}

```

## Output

a default mutex will be nonrecursive

**Related Information**

- “pthread.h” on page 38
- “pthread\_mutexattr\_init() — Initialize a Mutex Attribute Object” on page 1004
- “pthread\_mutexattr\_setkind\_np() — Set Kind for a Mutex Attribute Object” on page 1006

pthread\_mutexattr\_getpshared() — Get the Process-Shared Mutex Attribute

Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) OS/390 V2R7

Format

```
| #define _OPEN_THREADS
| #include <pthread.h>
|
| int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr,
|                                 int *pshared);
```

General Description

The pthread\_mutexattr\_getpshared() function gets the attribute *pshared* for the mutex attribute object *attr*. By using *attr* with the pthread\_mutexattr\_getpshared() function you can determine its *process-shared* value for a mutex.

The valid values for the attribute *pshared* are:

PTHREAD\_PROCESS\_SHARED

Permits a mutex to be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes.

PTHREAD\_PROCESS\_PRIVATE

A mutex can only be operated upon by threads created within the same process as the thread that initialized the mutex. When a new process is created by the parent process it will receive a different copy of the private mutex and this new mutex can only be used to serialize between threads in the child process. The default value of the attribute is PTHREAD\_PROCESS\_PRIVATE.

Returned Value

If successful, pthread\_mutexattr\_getpshared() returns 0.

If unsuccessful, pthread\_mutexattr\_getpshared() returns -1, and returns the value in errno. The following are the possible values of errno:

EINVAL      The value specified for *attr* is not valid.

Related Information

- “pthread.h” on page 38
- “pthread\_rwlockattr\_getpshared() — Get the Processed-Shared Read-Write Lock Attribute” on page 1025
- “pthread\_rwlockattr\_setpshared() — Set the Process-Shared Read-Write Lock Attribute” on page 1027
- “pthread\_mutexattr\_setpshared() — Set the Process-Shared Mutex Attribute” on page 1009

## pthread\_mutexattr\_gettype() — Get Type of Mutex Attribute Object

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) OS/390 V2R7

### Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_mutexattr_gettype(const pthread_mutexattr_t *attr, int *type);
```

### General Description

The `pthread_mutexattr_gettype()` function gets the attribute *type* from the mutex attribute object *attr*.

A mutex attribute object allows you to manage the characteristics of mutexes in your application. It defines the set of values to be used for the mutex during its creation. By establishing a mutex attribute object, you can create many mutexes with the same set of characteristics, without needing to define the characteristics for each and every mutex.

The values for the attribute *type* are:

#### PTHREAD\_MUTEX\_NORMAL

A normal type mutex does not detect deadlock. That is, a thread attempting to relock this mutex without first unlocking it will deadlock. The mutex is either in a locked or unlocked state for a thread.

#### PTHREAD\_MUTEX\_ERRORCHECK

An errorcheck type mutex provides error checking. That is, a thread attempting to relock this mutex without first unlocking it will return with an error. The mutex is either in a locked or unlocked state for a thread. If a thread attempts to lock a mutex that it has already locked, it will return with an error. If a thread attempts to lock a mutex that another thread has already locked, it will return with an error. If a thread attempts to lock a mutex that is unlocked, it will return with an error.

#### PTHREAD\_MUTEX\_RECURSIVE

A recursive type mutex permits a thread to lock many times. That is, a thread attempting to relock this mutex without first unlocking will succeed. This type of mutex must be unlocked the same number of times it is locked before the mutex will be returned to an unlocked state. If locked, an error is returned.

#### PTHREAD\_MUTEX\_DEFAULT

The default type mutex is mapped to a normal type mutex which does not detect deadlock. That is, a thread attempting to relock this mutex without first unlocking it will deadlock. The mutex is either in a locked or unlocked state for a thread. The normal mutex is the default type mutex.



**\_\_MUTEX\_NONRECURSIVE**

A nonrecursive mutex can be locked only once. That is, the mutex is either in a locked or unlocked state for a thread. If a thread attempts to lock a mutex that it has already locked, an error is returned.

**\_\_MUTEX\_RECURSIVE**

A recursive mutex can be locked more than once by the same thread. A count of the number of times the mutex has been locked is maintained. The mutex is unlocked when pthread\_mutex\_unlock() is performed an equal number of times.

**\_\_MUTEX\_NONRECURSIVE + \_\_MUTEX\_NODEBUG**

A nonrecursive mutex can be given an additional attribute, NODEBUG. This indicates that state changes to this mutex will *not* be reported to the debug interface, even if present.

**\_\_MUTEX\_RECURSIVE + \_\_MUTEX\_NODEBUG**

A recursive mutex can be given an additional attribute, NODEBUG. This indicates that state changes to this mutex will *not* be reported to the debug interface, even if present.

**Returned Value**

If successful, pthread\_mutexattr\_gettype() returns 0.

If unsuccessful, pthread\_mutexattr\_gettype() returns -1, and returns the value in errno. The following are the possible values of errno:

EINVAL      The value specified for *attr* is not valid.

**Related Information**

- “pthread.h” on page 38
- “pthread\_mutexattr\_settype() — Set Type of Mutex Attribute Object” on page 1011

## pthread\_mutexattr\_init() — Initialize a Mutex Attribute Object

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

### General Description

Initializes a mutex attribute object. With a mutex attribute object, you can manage the characteristics of mutexes in your application. It defines the set of values to be used for the mutex during its creation. By establishing a mutex attribute object, you can create many mutexes with the same set of characteristics, without defining those characteristics for each and every mutex.

For a valid mutex attribute, refer to “pthread\_mutexattr\_setkind\_np() — Set Kind for a Mutex Attribute Object” on page 1006.

### Returned Value

If successful, pthread\_mutexattr\_init() returns the value 0. If unsuccessful, it returns the value -1 and sets errno to ENOMEM, which indicates that there is not enough memory to initialize *attr*.

### Example

#### CBC3BP44

```
/* CBC3BP44 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

main() {
    pthread_mutexattr_t attr;
    pthread_mutex_t mutex;

    if (pthread_mutexattr_init(&attr) != 0) {
        perror("pthread_mutex_attr_init() error");
        exit(1);
    }

    if (pthread_mutexattr_setkind_np(ATTRIBUTE, __MUTEX_RECURSIVE) != 0) {
        perror("pthread_mutex_attr_setkind_np() error");
        exit(2);
    }

    if (pthread_mutex_init(&mutex, &attr) != 0) {
        perror("pthread_mutex_init() error");
        exit(3);
    }

    if (pthread_mutexattr_destroy(&attr) != 0) {
        perror("pthread_mutex_attr_destroy() error");
    }
}
```

```
        exit(4);  
    }  
}
```

**Related Information**

- “pthread.h” on page 38
- “pthread\_cond\_init() — Initialize a Condition Variable” on page 947
- “pthread\_create() — Create a Thread” on page 963
- “pthread\_mutex\_init() — Initialize a Mutex Object” on page 988

## pthread\_mutexattr\_setkind\_np() — Set Kind for a Mutex Attribute Object

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_mutexattr_setkind_np(pthread_mutexattr_t *attr, int kind);
```

### General Description

Sets the attribute *kind* for the mutex attribute object *attr*. With a mutex attribute object, you can manage the characteristics of mutexes in your application. It defines the set of values to be used for the mutex during its creation. By establishing a mutex attribute object, you can create many mutexes with the same set of characteristics, without defining those characteristics for each and every mutex.

The valid values for the attribute *kind* are:

**\_\_MUTEX\_NONRECURSIVE**

A nonrecursive mutex can be locked only once. That is, the mutex is either in a locked or unlocked state for a thread. If a thread attempts to lock a mutex that it has already locked, an error is returned.

**\_\_MUTEX\_RECURSIVE**

A recursive mutex can be locked more than once by the same thread. A count of the number of times the mutex has been locked is maintained. The mutex is unlocked when an equal number of pthread\_mutex\_unlock() functions are performed.

**\_\_MUTEX\_NONRECURSIVE + \_\_MUTEX\_NODEBUG**

A nonrecursive mutex can be given an additional attribute, NODEBUG. This indicates that state changes to this mutex will *not* be reported to the debug interface, even though it is present.

**\_\_MUTEX\_RECURSIVE + \_\_MUTEX\_NODEBUG**

A recursive mutex can be given an additional attribute, NODEBUG. This indicates that state changes to this mutex will *not* be reported to the debug interface, even though it is present.

### Returned Value

If successful, pthread\_mutexattr\_setkind\_np() returns the value 0. If unsuccessful, it returns the value -1 and sets errno to EINVAL, which indicates that the value specified for *attr* or *kind* is not valid.

## Example

### CBC3BP45

```

/* CBC3BP45 */
#pragma runopts(TEST(ALL))

#ifdef _OPEN_THREADS
#define _OPEN_THREADS
#endif

#include <stdio.h>
#include <pthread.h>

pthread_mutexattr_t attr;

int kind;

main() {
    if (pthread_mutexattr_init(&attr) == -1) {
        perror("pthread_mutexattr_init()");
        exit(1);
    }

    if (pthread_mutexattr_setkind_np(&attr, \
        __MUTEX_RECURSIVE + __MUTEX_NODEBUG) == -1 ) {
        perror("pthread_mutexattr_setkind_np()");
        exit(1);
    }

    if (pthread_mutexattr_getkind_np(&attr, &kind) == -1) {
        perror("pthread_mutexattr_getkind_np()");
        exit(1);
    }

    switch(kind) {

        case __MUTEX_NONRECURSIVE:
            printf("\nmutex will be nonrecursive");
            break;

        case __MUTEX_NONRECURSIVE+__MUTEX_NODEBUG:
            printf("\nmutex will be nonrecursive + nodebug");
            break;

        case __MUTEX_RECURSIVE:
            printf("\nmutex will be recursive");
            break;

        case __MUTEX_RECURSIVE+__MUTEX_NODEBUG:
            printf("\nmutex will be recursive + nodebug");
            break;

        default:
            printf("\nattribute kind value returned by \
pthread_mutexattr_getkind_np() unrecognized");
            exit(1);
    }

    exit(0);
}

```

**Related Information**

- “pthread.h” on page 38
- “pthread\_mutexattr\_init() — Initialize a Mutex Attribute Object” on page 1004
- “pthread\_mutexattr\_getkind\_np() — Get Kind from a Mutex Attribute Object” on page 998

## pthread\_mutexattr\_setpshared() — Set the Process-Shared Mutex Attribute

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) OS/390 V2R7

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);
```

### General Description

The `pthread_mutexattr_setpshared()` function sets the attribute *pshared* for the mutex attribute object *attr*.

A mutex attribute object allows you to manage the characteristics of mutexes in your application. It defines the set of values to be used for a mutex during its creation. By establishing a mutex attribute object, you can create many mutexes with the same set of characteristics, without needing to define the characteristics for each and every mutex. By using *attr* with the `pthread_mutexattr_setpshared()` function you can define its *process-shared* value for a mutex.

The valid values for the attribute *pshared* are:

#### PTHREAD\_PROCESS\_SHARED

Permits a mutex to be operated upon by any thread that has access to the memory where the mutex is allocated, even if the mutex is allocated in memory that is shared by multiple processes.

#### PTHREAD\_PROCESS\_PRIVATE

A mutex can only be operated upon by threads created within the same process as the thread that initialized the mutex; if threads of differing processes attempt to operate on such a mutex, only the process to initialize the mutex will succeed. When a new process is created by the parent process it will receive a different copy of the private mutex and this new mutex can only be used to serialize between threads in the child process. The default value of the attribute is `PTHREAD_PROCESS_PRIVATE`.

### Returned Value

If successful, `pthread_mutexattr_setpshared()` returns 0.

If unsuccessful, `pthread_mutexattr_setpshared()` returns `-1`, and returns the value in `errno`. The following are the possible values of `errno`:

**EINVAL**      The value specified for *attr* or *pshared* is not valid.

**Related Information**

- “pthread.h” on page 38
- “pthread\_mutexattr\_getpshared() — Get the Process-Shared Mutex Attribute” on page 1001
- “pthread\_rwlockattr\_getpshared() — Get the Process-Shared Read-Write Lock Attribute” on page 1025
- “pthread\_rwlockattr\_setpshared() — Set the Process-Shared Read-Write Lock Attribute” on page 1027



## pthread\_mutexattr\_settype() — Set Type of Mutex Attribute Object

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) OS/390 V2R7

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int type);
```

### General Description

The `pthread_mutexattr_settype()` function sets the attribute *type* from the mutex attribute object *attr*.

A mutex attribute object allows you to manage the characteristics of mutexes in your application. It defines the set of values to be used for the mutex during its creation. By establishing a mutex attribute object, you can create many mutexes with the same set of characteristics, without needing to define the characteristics for each and every mutex.

The values for the attribute *type* are:

#### PTHREAD\_MUTEX\_NORMAL

A normal type mutex does not detect deadlock. That is, a thread attempting to relock this mutex without first unlocking it will deadlock. The mutex is either in a locked or unlocked state for a thread.

#### PTHREAD\_MUTEX\_ERRORCHECK

An errorcheck type mutex provides error checking. That is, a thread attempting to relock this mutex without first unlocking it will return with an error. The mutex is either in a locked or unlocked state for a thread. If a thread attempts to lock a mutex that it has already locked, it will return with an error. If a thread attempts to lock a mutex that another thread has already locked, it will return with an error. If a thread attempts to lock a mutex that is unlocked, it will return with an error.

#### PTHREAD\_MUTEX\_RECURSIVE

A recursive type mutex permits a thread to lock many times. That is, a thread attempting to relock this mutex without first unlocking will succeed. This type of mutex must be unlocked the same number of times it is locked before the mutex will be returned to an unlocked state. If locked, an error is returned.

#### PTHREAD\_MUTEX\_DEFAULT

The default type mutex is mapped to a normal type mutex which does not detect deadlock. That is, a thread attempting to relock this mutex without first unlocking it will deadlock. The mutex is either in a locked or unlocked state for a thread. The normal mutex is the default type mutex.

### \_\_MUTEX\_NONRECURSIVE

A nonrecursive mutex can be locked only once. That is, the mutex is either in a locked or unlocked state for a thread. If a thread attempts to lock a mutex that it has already locked, an error is returned.

### \_\_MUTEX\_RECURSIVE

A recursive mutex can be locked more than once by the same thread. A count of the number of times the mutex has been locked is maintained. The mutex is unlocked when `pthread_mutex_unlock()` is performed an equal number of times.

### \_\_MUTEX\_NONRECURSIVE + \_\_MUTEX\_NODEBUG

A nonrecursive mutex can be given an additional attribute, `NODEBUG`. This indicates that state changes to this mutex will *not* be reported to the debug interface, even if present.

### \_\_MUTEX\_RECURSIVE + \_\_MUTEX\_NODEBUG

A recursive mutex can be given an additional attribute, `NODEBUG`. This indicates that state changes to this mutex will *not* be reported to the debug interface, even if present.

## Returned Value

If successful, `pthread_mutexattr_settype()` returns 0.

If unsuccessful, `pthread_mutexattr_settype()` returns -1, and returns the value in `errno`. The following are the possible values of `errno`:

`EINVAL`      The value specified for *attr* is not valid.

## Related Information

- “pthread.h” on page 38
- “pthread\_mutexattr\_gettype() — Get Type of Mutex Attribute Object” on page 1002

## pthread\_once() — Invoke a Function Once

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
pthread_once_t once_control = PTHREAD_ONCE_INIT;

int pthread_once(pthread_once_t *once_control, void(*init_routine) ());
```

### General Description

Establishes a function that will be executed only once in a given process. You may have each thread call the function, but only the first call causes the function to run. This is true even if called simultaneously by multiple threads. For example, a mutex or a thread-specific data key must be created exactly once. Calling `pthread_once()` prevents the code that creates a mutex or thread-specific data from being called by multiple threads. Without this routine, the execution must be serialized so that only one thread performs the initialization. Other threads that reach the same point in the code are delayed until the first thread is finished.

`pthread_once()` is used in conjunction with a *once\_control* variable of the type `pthread_once_t`. This variable is a data type that you initialize to the `PTHREAD_ONCE_INIT` constant. It is then passed as a parameter on the `pthread_once()` function call.

*init\_routine* is a normal function. It can be invoked directly outside of `pthread_once()`. In addition, it is the *once\_control* variable that determines if the *init\_routine* has been invoked. Calling `pthread_once()` with the same routine but with different *once\_control* variables, will result in the routine being called twice, once for each *once\_control* variable.

### Returned Value

If successful, `pthread_once()` returns the value 0. If unsuccessful, it returns the value -1. There are no documented `errno`s for this function. Use `perror()` or `strerror()` to determine the cause of the error.

### Example

#### CBC3BP46

```
/* CBC3BP46 */
#ifdef _OPEN_THREADS
#define _OPEN_THREADS
#endif

#include <stdio.h>
#include <errno.h>
#include <pthread.h>

#define threads 3
```

## pthread\_once

```
int          once_counter=0;
pthread_once_t once_control = PTHREAD_ONCE_INIT;

void once_fn(void) {
    puts("in once_fn");
    once_counter++;
}

void          *threadfunc(void *parm) {
    int          status;
    int          threadnum;
    int          *tnum;

    tnum = parm;
    threadnum = *tnum;

    printf("Thread %d executing\n", threadnum);

    status = pthread_once(&once_control, once_fn);
    if ( status < 0)
        printf("pthread_once failed, thread %d, errno=%d\n", threadnum, errno);

    pthread_exit((void *)0);
}

main() {
    int          status;
    int          i;
    int          threadparm[threads];
    pthread_t     threadid[threads];
    int          thread_stat[threads];

    for (i=0; i<threads; i++) {
        threadparm[i] = i+1;
        status = pthread_create( &threadid[i],
                                NULL,
                                threadfunc,
                                (void *)&threadparm[i]);

        if ( status < 0) {
            printf("pthread_create failed, errno=%d", errno);
            exit(2);
        }
    }

    for ( i=0; i<threads; i++) {
        status = pthread_join( threadid[i], (void *)&thread_stat[i]);
        if ( status < 0)
            printf("pthread_join failed, thread %d, errno=%d\n", i+1, errno);

        if (thread_stat[i] != 0)
            printf("bad thread status, thread %d, status=%d\n", i+1,
                    thread_stat[i]);
    }

    if (once_counter != 1)
        printf("once_fn did not get control once, counter=%d", once_counter);
    exit(0);
}
```

### Output

Thread 1 executing  
in once\_fn  
Thread 2 executing  
Thread 3 executing

**Related Information**

- “pthread.h” on page 38

## pthread\_rwlock\_destroy() — Destroy a Read-Write Lock Object

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) OS/390 V2R7

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

### General Description

The `pthread_rwlock_destroy()` function deletes a read-write lock object, which is identified by `rwlock` and releases any resources used by this read-write lock object. Read-write locks are used to protect shared resources.

**Note:** `rwlock` is set to an invalid value by `pthread_rwlock_destroy()` but can be reinitialized using `pthread_rwlock_init()`.

### Returned Value

If successful, `pthread_rwlock_destroy()` returns 0.

If unsuccessful, `pthread_rwlock_destroy()` returns -1, and returns the value in `errno`. The following are the possible value(s) of `errno`:

**EBUSY**      An attempt was made to destroy the object referenced by `rwlock` while it is locked or referenced as part of a wait on a condition variable.

**EINVAL**      The value specified by `rwlock` is not valid.

### Related Information

- “pthread.h” on page 38
- “pthread\_rwlock\_init() — Initialize a Read-Write Lock Object” on page 1017
- “pthread\_rwlock\_rdlock() — Wait for a Lock on a Read-Write Lock Object” on page 1018
- “pthread\_rwlock\_wrlock() — Wait for a Lock on a Read-Write Lock Object for Writing” on page 1023
- “pthread\_rwlock\_tryrdlock() — Attempt to Lock a Read-Write Lock Object for Reading” on page 1020
- “pthread\_rwlock\_trywrlock() — Attempt to Lock a Read-Write Lock Object for Writing” on page 1021
- “pthread\_rwlock\_unlock() — Unlock a Read-Write Lock Object” on page 1022

## pthread\_rwlock\_init() — Initialize a Read-Write Lock Object

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) OS/390 V2R7

### Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_rwlock_init(pthread_rwlock_t *rwlock,
                        pthread_rwlockattr_t *attr);
pthread_rwlock_t rwlock=PTHREAD_RWLOCK_INITIALIZER;
```

### General Description

The `pthread_rwlock_init()` function creates a read-write lock, referenced by *rwlock*, with attributes specified by *attr*. If *attr* is NULL, the default read-write lock attribute (PTHREAD\_PROCESS\_PRIVATE) is used. Once initialized, the lock can be used any number of times without being re-initialized. Upon successful initialization, the state of the read-write lock becomes initialized and unlocked.

In cases where default read-write lock attributes are appropriate, the macro PTHREAD\_RWLOCK\_INITIALIZER can be used to initialize read-write locks that are statically allocated. The effect is equivalent to dynamic initialization by a call to *pthread\_rwlock\_init()* with parameter *attr* specified as NULL, except that no error checking is done.

### Returned Value

If successful, `pthread_rwlock_init()` returns 0, and the state of the read-write lock becomes initialized and unlocked.

If unsuccessful, `pthread_rwlock_init()` returns -1, and returns the value in `errno`. The following are the possible value(s) of `errno`:

- EAGAIN     The system lacked necessary resources (other than memory) to initialize another read-write lock.
- ENOMEM    There is not enough memory to initialize the read-write lock.
- EPERM     The caller does not have the privilege to perform the operation.
- EINVAL    The value specified by *attr* is not valid.

### Related Information

- “pthread.h” on page 38

## pthread\_rwlock\_rdlock() — Wait for a Lock on a Read-Write Lock Object

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) OS/390 V2R7

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
```

### General Description

The `pthread_rwlock_rdlock()` function applies a read lock to the read-write lock referenced by `rwlock`. The calling thread acquires the read lock if a writer does not hold the lock and there are no writers blocked on the lock. In OS/390 UNIX, the calling thread does not acquire the lock when a writer does not hold the lock and there are writers waiting for the lock unless the thread already held `rwlock` for read. It will block and wait until there are no writers holding or waiting for the read-write lock. If a writer holds the lock, the calling thread will not acquire the read lock. If the read lock is not acquired, the calling thread blocks (that is, it does not return from the `pthread_rwlock_rdlock()` call) until it can acquire the lock.

A thread may hold multiple concurrent read locks on `rwlock` (that is successfully call the `pthread_rwlock_rdlock()` function *n* times). If so, the thread must perform matching unlocks (that is, it must call the `pthread_rwlock_unlock()` function *n* times). Read-write locks are used to protect shared resources.

**Note:** If a thread owns locks at the time it is terminated then OS/390 UNIX will release those locks.

### Returned Value

If successful, `pthread_rwlock_rdlock()` returns 0.

If unsuccessful, `pthread_rwlock_rdlock()` returns -1, and returns the value in `errno`. The following are the possible values of `errno`:

- EINVAL** The value specified by `rwlock` is not valid.
- EDEADLK** The current thread already owns the read-write lock for writing.
- EAGAIN** The read lock could not be acquired because the maximum number of read locks for `rwlock` has been exceeded. This `errno` will only occur in the shared path.
- ENOMEM** There is not enough memory to acquire a lock. This `errno` will only occur in the private path.



### **Related Information**

- “pthread.h” on page 38
- “pthread\_rwlock\_destroy() — Destroy a Read-Write Lock Object” on page 1016
- “pthread\_rwlock\_init() — Initialize a Read-Write Lock Object” on page 1017

## pthread\_rwlock\_tryrdlock() — Attempt to Lock a Read-Write Lock Object for Reading

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) OS/390 V2R7

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

### General Description

The `pthread_rwlock_tryrdlock()` function applies a read lock as in the `pthread_rwlock_rdlock()` function with the exception that the function fails if any thread holds a write lock on *rwlock* or there are writers blocked on *rwlock* unless the thread already held *rwlock* for read. Read-write locks are used to protect shared resources.

If the read-write lock identified by *rwlock* is locked, `pthread_rwlock_tryrdlock()` returns immediately.

When there are only read locks on the read-write lock, `pthread_rwlock_tryrdlock()` will effectively add to the count of the number of times `pthread_rwlock_unlock()` must be called by the thread to release the mutex (that is, it has the same behavior as a `pthread_rwlock_rdlock()` function).

### Returned Value

If successful, `pthread_rwlock_tryrdlock()` returns 0.

If unsuccessful, `pthread_rwlock_tryrdlock()` returns -1, and returns the value in `errno`. The following are the possible values of `errno`:

- |        |   |
|--------|---|
| EBUSY  | <i>rwlock</i> could not be acquired because it was already locked.  |
| EINVAL | The value specified by <i>rwlock</i> is not valid.  |
| EAGAIN | The read lock could not be acquired because the maximum number of read locks for <i>rwlock</i> has been exceeded. This <code>errno</code> will only occur in the shared path. |
| ENOMEM | There is not enough memory to acquire a lock. This <code>errno</code> will only occur in the private path.  |

### Related Information

- “pthread.h” on page 38
- “pthread\_rwlock\_destroy() — Destroy a Read-Write Lock Object” on page 1016
- “pthread\_rwlock\_init() — Initialize a Read-Write Lock Object” on page 1017

## pthread\_rwlock\_trywrlock() — Attempt to Lock a Read-Write Lock Object for Writing

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) OS/390 V2R7

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

### General Description

The `pthread_rwlock_trywrlock()` function applies a write lock as in the `pthread_rwlock_wrlock()` function with the exception that the function fails if any thread holds either a read lock or a write lock on *rwlock*. Read-write locks are used to protect shared resources.

If the read-write lock identified by *rwlock* is locked, `pthread_rwlock_trywrlock()` returns immediately.

### Returned Value

If successful, `pthread_rwlock_trywrlock()` returns 0.

If unsuccessful, `pthread_rwlock_trywrlock()` returns -1, and returns the value in `errno`. The following are the possible values of `errno`:

**EBUSY**      *rwlock* could not be acquired because it was already locked.

**EINVAL**     The value specified by *rwlock* is not valid.

**ENOMEM**    There is not enough memory to acquire a lock. This `errno` will only occur in the private path.

### Related Information

- “pthread.h” on page 38
- “pthread\_rwlock\_destroy() — Destroy a Read-Write Lock Object” on page 1016
- “pthread\_rwlock\_init() — Initialize a Read-Write Lock Object” on page 1017

## pthread\_rwlock\_unlock() — Unlock a Read-Write Lock Object

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) OS/390 V2R7

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

### General Description

The `pthread_rwlock_unlock()` function releases a read-write lock object. If one or more threads are waiting to lock the `rwlock`, `pthread_rwlock_unlock()` causes one or more of these threads to return from the `pthread_rwlock_rdlock()` or the `pthread_rwlock_wrlock()` call with the read-write lock object acquired. If there are multiple threads blocked on `rwlock` for both read locks and write locks, OS/390 UNIX will give the read-write lock to the next waiting call whether it is a read or a write request even when there is a writer blocked waiting for the lock. If no threads are waiting for the `rwlock`, the `rwlock` unlocks with no current owner.

### Returned Value

If successful, `pthread_rwlock_unlock()` returns 0.

If unsuccessful, `pthread_rwlock_unlock()` returns -1, and returns the value in `errno`. The following are the possible values of `errno`:

- `EINVAL`     The value specified for `rwlock` is not valid.
- `EPERM`     The current thread does not own the read\_write lock object.
- `ENOMEM`    There is not enough memory during the unlock process. This `errno` will only occur in the private path.

### Related Information

- “pthread.h” on page 38

## pthread\_rwlock\_wrlock() — Wait for a Lock on a Read-Write Lock Object for Writing

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) OS/390 V2R7

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rlock);
```

### General Description

The `pthread_rwlock_wrlock()` function applies a write lock to the read-write lock referenced by *rlock*. The calling thread acquires the write lock if no other thread (reader or writer) holds the read-write lock *rlock*. Otherwise, the thread blocks (that is, does not return from the `pthread_rwlock_wrlock()` call) until it can acquire the lock. In OS/390 UNIX the calling thread does not acquire the lock when a writer does not hold the lock and there are writers waiting for the lock. It will block and wait until there are no writers holding or waiting for the read-write lock. If the thread already holds read-write lock for either read or write then a deadlock error will be returned.

**Note:** If a thread owns locks at the time it is terminated then OS/390 UNIX will release those locks.

### Returned Value

If successful, `pthread_rwlock_wrlock()` returns 0.

If unsuccessful, `pthread_rwlock_wrlock()` returns -1, and returns the value in `errno`. The following are the possible values of `errno`:

- EINVAL**      The value specified by *rlock* is not valid.
- EDEADLK**    The current thread already owns the read-write lock for writing or reading.
- ENOMEM**    There is not enough memory to acquire a lock. This error will only occur in the private path.

### Related Information

- “pthread.h” on page 38
- “pthread\_rwlock\_destroy() — Destroy a Read-Write Lock Object” on page 1016
- “pthread\_rwlock\_init() — Initialize a Read-Write Lock Object” on page 1017

## pthread\_rwlockattr\_destroy() — Destroy a Read-Write Lock Attribute Object

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) OS/390 V2R7

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_rwlockattr_destroy (pthread_rwlockattr_t *attr);
```

### General Description

The `pthread_rwlockattr_destroy()` function destroys an initialized rwlock attribute object.

After a read-write lock attributes object has been used to initialize one or more read-write locks any function affecting the attributes object (including destruction) does not affect any previously initialized read-write locks.

The `pthread_rwlockattr_destroy()` function destroys a read-write lock attributes object. Subsequent use of the object will cause an error until the object is re-initialized by another call to `pthread_rwlockattr_init()`.

### Returned Value

If successful, `pthread_rwlockattr_destroy()` returns 0.

If unsuccessful, `pthread_rwlockattr_destroy()` returns -1, and returns the value in `errno`. The following are the possible value(s) of `errno`:

`EINVAL`      The value specified for *attr* is not valid.

### Related Information

- “pthread.h” on page 38
- “pthread\_rwlockattr\_init() — Initialize a Read-Write Lock Attribute Object” on page 1026

# pthread\_rwlockattr\_getpshared() — Get the Processed-Shared Read-Write Lock Attribute

## Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) OS/390 V2R7

## Format

```
| #define _OPEN_THREADS
| #include <pthread.h>
|
| int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *attr,
|                                   int *pshared);
```

## General Description

The pthread\_rwlockattr\_getpshared() function gets the attribute *pshared* for the read-write lock attribute object *attr*. By using *attr* with the pthread\_rwlockattr\_getpshared() function you can determine its *process-shared* value for a read-write lock.

The valid values for the attribute *pshared* are:

### PTHREAD\_PROCESS\_SHARED

Permits a read-write lock to be operated upon by any thread that has access to the memory where the read-write lock is allocated, even if the read-write lock is allocated in memory that is shared by multiple processes.

### PTHREAD\_PROCESS\_PRIVATE

A read-write lock can only be operated upon by threads created within the same process as the thread that initialized the read-write lock. When a new process is created by the parent process it will receive a different copy of the private read-write lock and this new read-write lock can only be used to serialize between threads in the child process. The default value of the attributed is PTHREAD\_PROCESS\_PRIVATE.

## Returned Value

If successful, pthread\_rwlockattr\_getpshared() returns 0.

If unsuccessful, pthread\_rwlockattr\_getpshared() returns -1, and returns the value in errno. The following are the possible values of errno:

EINVAL      The value specified for *attr* is not valid.

## Related Information

- “pthread.h” on page 38
- “pthread\_rwlock\_init() — Initialize a Read-Write Lock Object” on page 1017
- “pthread\_rwlockattr\_setpshared() — Set the Process-Shared Read-Write Lock Attribute” on page 1027

## pthread\_rwlockattr\_init() — Initialize a Read-Write Lock Attribute Object

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) OS/390 V2R7

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
```

### General Description

The `pthread_rwlockattr_init()` function initializes a read-write lock attribute object. A read-write lock attribute object allows you to manage the characteristics of read-write locks in your application. It defines the set of values to be used for the read-write lock during its creation. By establishing a read-write lock attribute object, you can create many read-write locks with the same set of characteristics, without needing to define the characteristics for each and every read-write lock.

For a valid read-write lock attribute, refer to “`pthread_rwlockattr_setpshared()` — Set the Process-Shared Read-Write Lock Attribute” on page 1027.

If `pthread_rwlockattr_init()` is called specifying an already initialized read-write lock attributes object the request is rejected and the current lock attributes object is unchanged.

### Returned Value

If successful, `pthread_rwlockattr_init()` returns 0.

If unsuccessful, `pthread_rwlockattr_init()` returns `-1`, and returns the value in `errno`. The following are the possible value(s) of `errno`:

**ENOMEM** There is not enough memory to initialize *attr*.

### Related Information

- “`pthread.h`” on page 38
- “`pthread_rwlock_init()` — Initialize a Read-Write Lock Object” on page 1017



## pthread\_rwlockattr\_setpshared() — Set the Process-Shared Read-Write Lock Attribute

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) OS/390 V2R7

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);
```

### General Description

The `pthread_rwlockattr_setpshared()` function sets the attribute *pshared* for the read-write lock attribute object *attr*.

A read-write lock attribute object allows you to manage the characteristics of read-write locks in your application. It defines the set of values to be used for a read-write lock during its creation. By establishing a read-write lock attribute object, you can create many read-write locks with the same set of characteristics, without needing to define those characteristics for each and every read-write lock. By using *attr* with the `pthread_rwlockattr_setpshared()` function you can define its *process-shared* value for a read-write lock.

The valid values for the attribute *pshared* are:

#### PTHREAD\_PROCESS\_SHARED

Permits a read-write lock to be operated upon by any thread that has access to the memory where the read-write lock is allocated, even if the read-write lock is allocated in memory that is shared by multiple processes.

#### PTHREAD\_PROCESS\_PRIVATE

A read-write lock can only be operated upon by threads created within the same process as the thread that initialized the read-write lock. When a new process is created by the parent process it will receive a different copy of the private read-write lock and this new read-write lock can only be used to serialize between threads in the child process. The default value of the attributed is `PTHREAD_PROCESS_PRIVATE`.

### Returned Value

If successful, `pthread_rwlockattr_setpshared()` returns 0.

If unsuccessful, `pthread_rwlockattr_setpshared()` returns -1, and returns the value in `errno`. The following are the possible values of `errno`:

**EINVAL**      The value specified for *attr* or *pshared* is not valid.

**Related Information**

- “pthread.h” on page 38
- “pthread\_rwlock\_init() — Initialize a Read-Write Lock Object” on page 1017
- “pthread\_rwlockattr\_getpshared() — Get the Processed-Shared Read-Write Lock Attribute” on page 1025

## pthread\_security\_np() — Create or Delete Thread Level Security

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_SYS 1
#include <pthread.h>
int pthread_security_np(int function_code,
                       int identity_type,
                       size_t identity_length,
                       void *identity,
                       char *password,
                       int options);
```

### General Description

pthread\_security\_np() creates or deletes a thread-level security environment for the calling thread.

The function supports the following parameters:

Parameter	Description
<i>function_code</i>	Specify one of the following:  <b>__CREATE_SECURITY_ENV</b> Create a thread-level security environment for the calling thread. If a thread-level security environment already exists, it is deleted before a new one is created.  <b>__DELETE_SECURITY_ENV</b> Delete a thread-level security environment for the calling thread if one exists.
<i>identity_type</i>	Specifies the format of the user identity in the argument, 'identity'. It can have one of the following values:  <b>__USERID_IDENTITY</b> User identity in the form of a character string (1 to 8 bytes in length)  <b>__CERTIFICATE_IDENTITY</b> User identity in the form of a '___certificate_t'.  A '___certificate_t' is a structure containing the following elements:  <b>___cert_type</b> The type of security certificate. Setting value <b>__CERT_X509</b> , for example, indicates the certificate is an X.509 security certificate.  <b>___userid</b> An output field in the <b>___certificate</b> structure that will be filled with the userid associated with the certificate. This output will be up to 8 characters long and null terminated.

<code>__cert_length</code>	The length in bytes of the security certificate.
<code>__cert_ptr</code>	A pointer to the start of the security certificate.
<code>identity_length</code>	Specifies the length of the <i>identity</i> parameter. If <i>identity_type</i> is <code>__USERID_IDENTITY</code> , <i>identity_length</i> is the length of the user identity character string. If <i>identity_type</i> is <code>__CERTIFICATE_IDENTITY</code> , <i>identity_length</i> is the length of the <code>__certificate</code> structure.
<code>identity</code>	Specifies the user identity according to the <i>identity</i> paramter.
<code>password</code>	Specifies a user password or pass ticket.
<code>options</code>	Specifies options used to tailor the request. <i>options</i> must be set to 0.

This function is intended to be used by servers which process requests from multiple clients. By creating and building a thread-level security environment for the client, a server can process many client requests without the overhead of issuing *fork/setuid/exec*.

In order to be able to invoke this service, the calling process must have appropriate privileges. There are two methods for assigning privileges to a server using this function:

1. If the installation has defined the BPX.SERVER FACILITY class profile within RACF, then the userid assigned to the server must be permitted to this profile. The following example shows the PERMIT command assuming the userid of the server is SERVERID:

```
PERMIT BPX.SERVER CLASS(FACILITY) ID(SERVERID) ACCESS(READ)
```

In addition to having access to the BPX.SERVER profile, the server's executable load modules will need to reside in PADS (Program Access to Data Sets) define libraries. If your load modules normally reside in HFS files, you need to turn on the sticky bit and link edit your load modules into an MVS load library which is PADS defined. If you are creating a server which exploits this function, the documentation of your server should tell the system programmer to read the section in the *OS/390 UNIX System Services Planning* on defining BPX.SERVER. This book also contains further information on making your programs PADS defined.

2. If the installation has chosen not to define BPX.SERVER, then the user id of the server needs to be assigned a UID of 0. This gives the server superuser authority, which is sufficient authority to invoke this function when BPX.SERVER is not defined. In this environment, your load modules do not have to be PADS defined.

When a thread level security environment is established, the other C functions get broken into two categories:

- a. Services which are used to access data in the file system base the permission checks on the thread level security. So a function like `open()` will only work if the identity of the user in the task security environment has permission to the file.
- b. Services which are process oriented base the permission checking on the security identity of the process. Functions like `kill` will only work if the process has permission to send the signal to the target process. The IPC

functions of shared memory, message queues and semaphores are also accessed with the security environment of the process.

If a thread with a thread level security environment issues a `spawn()` function call, the new process will have the identity of the process, not the thread.

Access to most MVS resources is based on the security identity of the thread.

The specification of a password is optional. The following are some examples of when a server would want to create a thread level security environment without a password:

- a. Some servers allow access to a system with a userid known as ANONYMOUS. The ANONYMOUS userid is defined to the system with access to data available to the general public. It is up to the installation to define and manage an ANONYMOUS userid so that integrity is not compromised.
- b. Some servers are connected to global security servers. If other services are used to authenticate a user, then it is not necessary to provide a password to this service. It is up to the application and the installation to define what level of user authentication is acceptable.

There is no support in DBX to allow debugging in this environment.

This service cannot be called from the initial thread.

## Returned Value

If successful, 0 is returned. Otherwise, a -1 is returned and `errno` is set to indicate the error.

If any of the following conditions occurs, the `pthread_security_np()` function sets `errno` to the corresponding value:

EINVAL	One of the following errors was detected: <ul style="list-style-type: none"> <li>Function_code specified is undefined.</li> <li>Identity_Type specified is undefined.</li> <li>Identity_Length specified was not valid for the Identity_Type.</li> <li>Password_Length specified was not in the range 0 to 8.</li> <li>An undefined option flag was set.</li> </ul>
EPERM	The process does not have appropriate privileges to set a thread level security environment. The caller is not permitted to the BPX.SERVER FACILITY class profile or BPX.SERVER is not defined and the caller is not a superuser. No password is provided and the caller is not defined as a surrogate of the passed userid.
ESRCH	The userid provided as input is not defined to RACF or does not have an OMVS segment defined.
EACCESS	The password provided is not valid for the passed userid.
EMVSEXPIRE	The password provided has expired.
EMVSSAF2ERR	The SAF call to the security product incurred an error.
EMVSSAFEXTRERR	The SAF call to the security product incurred an error.

- EMVSERR     An MVS environmental or internal error occurred.
- pthread\_security\_np() was called from the initial thread.
  - pthread\_security\_np() was called from a task that is being debugged using the ptrace() service.
  - An MVS internal error occurred

### Related Information

- “getlogin() — Get the User Login Name” on page 550

## pthread\_self() — Get the Caller

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
pthread_t pthread_self(void);
```

### General Description

Returns the thread ID of the calling thread.

### Returned Value

There are no documented errno's for this function. Use perror() or strerror() to determine the cause of the error.

### Example

#### CBC3BP47

```
/* CBC3BP47 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>

pthread_t thid, IPT;

void *thread(void *arg) {
    if (pthread_equal(IPT, pthread_self()))
        puts("the thread is the IPT...?");
    else
        puts("the thread is not the IPT");

    if (pthread_equal(thid, pthread_self()))
        puts("the thread is the one created by the IPT");
    else
        puts("the thread is not the one created by the IPT...?");
}

main() {
    IPT = pthread_self();
    if (pthread_create(&thid, NULL, thread, NULL) != 0) {
        perror("pthread_create() error");
        exit(1);
    }

    if (pthread_join(thid, NULL) != 0) {
        perror("pthread_join() error");
        exit(3);
    }
}
```

### Output

the thread is not the IPT  
the thread is the one created by the IPT

### Related Information

- “pthread.h” on page 38
- “pthread\_create() — Create a Thread” on page 963
- “pthread\_equal() — Compare Thread IDs” on page 967



## pthread\_setintr() — Set Thread's Cancelability State

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_setintr(int state);
```

### General Description

Controls whether the thread accepts a cancel request that was produced by a call to pthread\_cancel(). The cancelability states can be:

PTHREAD\_INTR\_DISABLE

The thread cannot be canceled.

PTHREAD\_INTR\_ENABLE

The thread can be canceled, but it is subject to type. The cancelability types can be found in "pthread\_setintrtype() — Set Thread's Cancelability Type" on page 1038.

### Returned Value

If successful, pthread\_setintr() returns the previous state. If unsuccessful, it returns the value -1 and sets errno to EINVAL, which indicates that state is an invalid value.

### Example

#### CBC3BP48

```
/* CBC3BP48 */
#define _OPEN_THREADS
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <errno.h>
#include <unistd.h>

int thstatus;
char state[60] = "enable/controlled - initial default";

void * thfunc(void *voidptr) {
    int rc;
    char *parmptr;

    parmptr = voidptr;
    printf("parm = %s.\n", parmptr);

    strcpy(state, "disable/controlled");
    if ( pthread_setintrtype(PTHREAD_INTR_CONTROLLED ) == -1 ) {
        printf("set controlled failed. %s\n", strerror(errno));
        thstatus = 103;
        pthread_exit(&thstatus);
    }
}
```

```

if ( pthread_setintr(PTHREAD_INTR_ENABLE) == -1 ) {
    printf("set enable failed. %s\n", strerror(errno));
    thstatus = 104;
    pthread_exit(&thstatus);
}

strcpy(state, "enable/controlled");
strcat(state, " - pthread_testintr");

while (1) {
    pthread_testintr();
    sleep(1);
}

thstatus = 100;
pthread_exit(&thstatus);
}

main(int argc, char *argv[]) {
    int          rc;
    pthread_attr_t attrarea;
    pthread_t     thid;
    char          parm[] = "abcdefghijklmnopqrstuvwxyz";
    int           *statptr;

    if ( pthread_attr_init(&attrarea) == -1 ) {
        printf("pthread_attr_init failed. %s\n", strerror(errno));
        exit(1);
    }

    if ( pthread_create(&thid, &attrarea, thfunc, (void *)&parm) == -1 ) {
        printf("pthread_create failed. %s\n", strerror(errno));
        exit(2);
    }

    sleep(5);
    if ( pthread_cancel(thid) == -1 ) {
        printf("pthread_cancel failed. %s\n", strerror(errno));
        exit(3);
    }

    if ( pthread_join(thid, (void **)&statptr) == -1 ) {
        printf("pthread_join failed. %s\n", strerror(errno));
        exit(4);
    }

    if ( statptr == (int *)-1 )
        printf("thread was cancelled. state = %s.\n", state);
    else
        printf("thread was not cancelled. thstatus = %d.\n", *statptr);
    exit(0);
}

```

**Output**

```

parm = abcdefghijklmnopqrstuvwxyz.
thread was cancelled. state = enable/controlled - pthread_testintr.

```

**Related Information**

- “pthread.h” on page 38
- “pthread\_cancel() — Cancel a Thread” on page 936
- “pthread\_setintrtype() — Set Thread's Cancelability Type” on page 1038
- “pthread\_testintr() — Establish a Cancelability Point” on page 1047

## pthread\_setintrtype() — Set Thread's Cancelability Type

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_setintrtype(int type);
```

### General Description

Controls when a cancel request is acted on. The cancelability types can be:

PTHREAD\_INTR\_ASYNCHRONOUS

The thread can be canceled at any time.

PTHREAD\_INTR\_CONTROLLED

The thread can be canceled, but only at specific points of execution. These are:

- When waiting on a condition variable, which is `pthread_cond_wait()` or `pthread_cond_timedwait()`
- When waiting for the end of another thread, which is `pthread_join()`
- While waiting for an asynchronous signal, which is `sigwait()`
- When setting the calling thread's cancelability state, which is `pthread_setintr()`
- Testing specifically for a cancel request, which is `pthread_testintr()`
- When suspended because of POSIX functions or one of the following C standard functions: `close()`, `fcntl()`, `open()`, `pause()`, `read()`, `tcdrain()`, `tcsetattr()`, `sigsuspend()`, `sigwait()`, `sleep()`, `wait()`, or `write()`

### Returned Value

If successful, `pthread_setintrtype()` returns the previous type. If unsuccessful, it returns the value `-1` and sets `errno` to `EINVAL`, which indicates that *type* is an invalid value.

### Example

#### CBC3BP50

```
/* CBC3BP50 */
#define _OPEN_THREADS
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <errno.h>
#include <unistd.h>

int thstatus;
char state[60] = "enable/controlled - initial default";
```

```

void * thfunc(void *voidptr)
{
    int rc;
    char *parmptr;

    parmptr = voidptr;
    printf("parm = %s.\n", parmptr);
    if ( pthread_setintrtype(PTHREAD_INTR_CONTROLLED ) == -1 ) {
        printf("set controlled failed. %s\n", strerror(errno));
        thstatus = 103;
        pthread_exit(&thstatus);
    }
    strcpy(state, "disable/controlled");

    if ( pthread_setintr(PTHREAD_INTR_ENABLE) == -1 ) {
        printf("set enable failed. %s\n", strerror(errno));
        thstatus = 104;
        pthread_exit(&thstatus);
    }
    strcpy(state, "enable/controlled");

    strcat(state, " - pthread_testintr");

    while(1) {
        pthread_testintr();
        sleep(1);
    }

    thstatus = 100;
    pthread_exit(&thstatus);
}

main(int argc, char *argv[]) {
    int rc;
    pthread_attr_t attrarea;
    pthread_t thid;
    char parm[] = "abcdefghijklmnopqrstuvwxyz";
    int *statp;

    if ( pthread_attr_init(&attrarea) == -1 ) {
        printf("pthread_attr_init failed. %s\n", strerror(errno));
        exit(1);
    }

    if ( pthread_create(&thid, &attrarea, thfunc, (void *)&parm) == -1 ) {
        printf("pthread_create failed. %s\n", strerror(errno));
        exit(2);
    }

    sleep(5);

    if ( pthread_cancel(thid) == -1 ) {
        printf("pthread_cancel failed. %s\n", strerror(errno));
        exit(3);
    }

    if ( pthread_join(thid, (void **)&statp) == -1 ) {
        printf("pthread_join failed. %s\n", strerror(errno));
        exit(4);
    }

    if ( statp == (int *)-1 )
        printf("thread was cancelled. state = %s.\n", state);
    else

```

## pthread\_setintrtype

```
    printf("thread was not cancelled. thstatus = %d.\n", *statptr);  
    exit(0);  
}
```

### Output

```
parm = abcdefghijklmnopqrstuvwxyz.  
thread was cancelled. state = enable/controlled - pthread_testintr.
```

### Related Information

- “pthread.h” on page 38
- “pthread\_cancel() — Cancel a Thread” on page 936
- “pthread\_setintr() — Set Thread's Cancelability State” on page 1035
- “pthread\_testintr() — Establish a Cancelability Point” on page 1047

## pthread\_set\_limit\_np() — Set Task and Thread Limits

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_SYS
#include <pthread.h>
int pthread_set_limit_np(int action, int maxthreadtasks, int maxthreads);
```

### General Description

The pthread\_set\_limit\_np() function allows you to control how many tasks and threads can be created for a process. On a single call, you can specify that you want to update either the maximum number of tasks, the maximum number of threads, or both. The maximum number of tasks and threads is dependent upon the size of the private area below 16M. A realistic limit is 200 to 400 tasks and threads.

The *action* can be set to one of the following symbolics, as defined in the pthread.h header file:

\_\_STL\_MAX\_TASKS

Specify this action when only updating the maximum number of tasks.

\_\_STL\_MAX\_THREADS

Specify this action when only updating the maximum number of threads.

\_\_STL\_SET\_BOTH

Specify this action when updating both the maximum number of tasks and the maximum number of threads at the same time.

For more information on the allowable values for maxthreadtasks and maxthreads, see the BPX1STL function in *OS/390 UNIX System Services Programming: Assembler Callable Services Reference*.

### Returned Value

If successful, pthread\_set\_limit\_np() returns 0.

If unsuccessful, pthread\_set\_limit\_np() returns -1.

For more information regarding return values and reason codes, see the BPX1STL function in *OS/390 UNIX System Services Programming: Assembler Callable Services Reference*.

**Related Information**

- “pthread.h” on page 38
- “pthread\_attr\_setweight\_np() — Set Weight of Thread Attribute Object” on page 934
- “pthread\_attr\_setsyncctype\_np() — Set Thread Sync Type” on page 933



## pthread\_setspecific() — Set the Thread-Specific Value for a Key

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_setspecific(pthread_key_t key, void *value);
```

### General Description

Associates a thread-specific value, *value*, with a key identifier, *key*.

Many multithreaded applications require storage shared among threads but a unique value for each thread. A thread-specific data key is an identifier, created by a thread, for which each thread in the process can set a unique key *value*.

*pthread\_key\_t* is a storage area where the system places the key identifier. To create a key, a thread uses `pthread_key_create()`. This returns the key identifier into the storage area of type *pthread\_key\_t*. At this point, each of the threads in the application has the use of that key, and can set its own unique value by use of `pthread_setspecific()`. A thread can get its own unique value using `pthread_getspecific()`.

### Returned Value

If successful, `pthread_setspecific()` returns the value 0. If unsuccessful, it returns the value -1 and sets `errno` to `EINVAL`, which indicates that the key identifier *key* is not valid.

### Example

#### CBC3BP51

```
/* CBC3BP51 */
#define _OPEN_THREADS

#include <stdio.h>
#include <errno.h>
#include <pthread.h>

#define threads 3
#define BUFFSZ 48
pthread_key_t key;

void *threadfunc(void *parm)
{
    int status;
    int value;
    int threadnum;
    int *tnum;
    void *getvalue;
    char Buffer[BUFFSZ];

    tnum = parm;
```

```

threadnum = *tnum;

printf("Thread %d executing\n", threadnum);

if (!(value = malloc(sizeof(Buffer))))
    printf("Thread %d could not allocate storage, errno = %d\n",
          threadnum, errno);
status = pthread_setspecific(key, (void *) value);
if ( status < 0) {
    printf("pthread_setspecific failed, thread %d, errno %d",
          threadnum, errno);
    pthread_exit((void *)12);
}
printf("Thread %d setspecific value: %d\n", threadnum, value);

getvalue = 0;
status = pthread_getspecific(key, &getvalue);
if ( status < 0) {
    printf("pthread_getspecific failed, thread %d, errno %d",
          threadnum, errno);
    pthread_exit((void *)13);
}

if ((int)getvalue != value) {
    printf("getvalue not valid, getvalue=%d", (int)getvalue);
    pthread_exit((void *)68);
}

pthread_exit((void *)0);
}

void destr_fn(void *parm)
{
    printf("Destructor function invoked\n");
    if (free(parm))
        printf("unable to free storage, errno = %d\n", errno);
}

main() {
    int      getvalue;
    int      status;
    int      i;
    int      threadparm[threads];
    pthread_t threadid[threads];
    int      thread_stat[threads];

    if ((status = pthread_key_create(&key, destr_fn )) < 0) {
        printf("pthread_key_create failed, errno=%d", errno);
        exit(1);
    }

    for (i=0; i<threads; i++) {
        threadparm[i] = i+1;
        status = pthread_create( &threadid[i],
                                NULL,
                                threadfunc,
                                (void *)&threadparm[i]);

        if ( status < 0) {
            printf("pthread_create failed, errno=%d", errno);
            exit(2);
        }
    }
}

```

```

for ( i=0; i<threads; i++) {
    status = pthread_join( threadid[i],
        (void *)&thread_stat[i]);
    if ( status < 0) {
        printf("pthread_join failed, thread %d, errno=%d\n", i+1, errno);
    }

    if (thread_stat[i] != 0) {
        printf("bad thread status, thread %d, status=%d\n", i+1,
            thread_stat[i]);
    }
}
exit(0);
}

```

### Related Information

- “pthread.h” on page 38
- “pthread\_getspecific() — Get the Thread-Specific Value for a Key” on page 971
- “pthread\_getspecific\_d8\_np() — Get the Thread-Specific Value for a Key” on page 974
- “pthread\_key\_create() — Create Thread-Specific Data Key” on page 981

## pthread\_tag\_np() — Set and Query Thread Tag Data

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
int pthread_tag_np(const char * newtag, char * oldtag);
```

### General Description

The pthread\_tag\_np() function is used to set and query the contents of the calling thread's tag data.

The parameters supported are:

**newtag** Specifies the new tag data to be set for the callers thread. The length of the new tag data must be in the range of 0 to 65 bytes. If the length is zero (NULL string) the caller's thread tag data will be cleared.

**oldtag** Specifies the string where pthread\_tag\_np() returns the old (current) tag data for the caller's thread. Tag data can be up to 66 bytes (including the trailing NULL).

### Returned Value

The pthread\_tag\_np() function returns 0 if successful. Otherwise, it returns -1 and sets errno to indicate the error.

errno values defined for pthread\_tag\_np().

#### errno      Meaning

**EINVAL** The length of the newtag string is not within allowable range (0 to 65 bytes).

**EFAULT** One of the following errors was detected:

- All or part of the newtag string is not addressable by the caller.
- All or part of the oldtag string is not addressable by the caller.

**EMVSERR** An MVS environmental or internal error has occurred.

### Related Information

- "pthread.h" on page 38
- "pthread\_create() — Create a Thread" on page 963

## pthread\_testintr() — Establish a Cancelability Point

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
void pthread_testintr(void);
```

### General Description

Allows the thread to solicit cancel requests at specific points within the current thread. You must have the cancelability state set to enabled (PTHREAD\_INTR\_ENABLE) for this function to have any effect.

### Returned Value

There is no return value. There are no documented errno's for this function. Use perror() or strerror() to determine the cause of the error.

### Example

#### CBC3BP52

```
/* CBC3BP52 */
#define _OPEN_THREADS
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <errno.h>
#include <unistd.h>

int thstatus;
char state[60] = "enable/controlled - initial default";

void * thfunc(void *voidptr)
{
    int rc;
    char *pamptr;

    pamptr = voidptr;
    printf("parm = %s.\n", pamptr);
    if ( pthread_setintrtype(PTHREAD_INTR_CONTROLLED) == -1 ) {
        printf("set controlled failed. %s\n", strerror(errno));
        thstatus = 103;
        pthread_exit(&thstatus);
    }
    strcpy(state, "disable/controlled");

    if ( pthread_setintr(PTHREAD_INTR_ENABLE) == -1 ) {
        printf("set enable failed. %s\n", strerror(errno));
        thstatus = 104;
        pthread_exit(&thstatus);
    }
    strcpy(state, "enable/controlled");

    strcat(state, " - pthread_testintr");
```

```

while(1) {
    pthread_testintr();
    sleep(1);
}

thstatus = 100;
pthread_exit(&thstatus);
}

main(int argc, char *argv[]) {
    int          rc;
    pthread_attr_t attrarea;
    pthread_t     thid;
    char         parm[] = "abcdefghijklmnopqrstuvwxyz";
    int          *statptr;

    if ( pthread_attr_init(&attrarea) == -1 ) {
        printf("pthread_attr_init failed. %s\n", strerror(errno));
        exit(1);
    }

    if ( pthread_create(&thid, &attrarea, thfunc, (void *)&parm) == -1 ) {
        printf("pthread_create failed. %s\n", strerror(errno));
        exit(2);
    }

    sleep(5);

    if ( pthread_cancel(thid) == -1 ) {
        printf("pthread_cancel failed. %s\n", strerror(errno));
        exit(3);
    }

    if ( pthread_join(thid, (void **)&statptr) == -1 ) {
        printf("pthread_join failed. %s\n", strerror(errno));
        exit(4);
    }

    if ( statptr == (int *)-1 )
        printf("thread was cancelled. state = %s.\n", state);
    else
        printf("thread was not cancelled. thstatus = %d.\n", *statptr);

    exit(0);
}

```

**Output**

parm = abcdefghijklmnopqrstuvwxyz.  
thread was cancelled. state = enable/controlled - pthread\_testintr.

**Related Information**

- “pthread.h” on page 38
- “pthread\_cancel() — Cancel a Thread” on page 936
- “pthread\_setintr() — Set Thread's Cancelability State” on page 1035
- “pthread\_setintrtype() — Set Thread's Cancelability Type” on page 1038

## pthread\_yield() — Release the Processor to Other Threads

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON) MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <pthread.h>
```

```
void pthread_yield(NULL);
```

### General Description

Allows a thread to give up control of a processor so that another thread may have the opportunity to run.

The parameter to the function must be NULL, because non-NULL values are reserved.

### Returned Value

There is no return value. There are no documented errno values for this function.

### Example CBC3BP53

```
/* CBC3BP53 */
#define _OPEN_THREADS
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

void *thread(void *arg) {

    /* A simple loop with only puts() would allow a thread to write several
    lines in a row.
    With pthread_yield(), each thread gives another thread a chance before
    it writes its next line */

    while (1) {
        puts((char*) arg);
        pthread_yield(NULL);
    }
}

main() {
    pthread_t t1, t2, t3;

    if (pthread_create(&t1, NULL, thread, "thread 1") != 0) {
        perror("pthread_create() error");
        exit(1);
    }

    if (pthread_create(&t2, NULL, thread, "thread 2") != 0) {
        perror("pthread_create() error");
        exit(2);
    }
}
```

## pthread\_yield

```
if (pthread_create(&t3, NULL, thread, "thread 3") != 0) {  
    perror("pthread_create() error");  
    exit(3);  
}  
  
sleep(1);  
  
exit(0); /* this will tear all threads down */  
}
```

### Output

```
thread 1  
thread 3  
thread 2  
thread 1  
thread 3  
thread 2  
thread 1  
thread 3  
thread 2  
thread 1  
thread 3
```

### Related Information

- “pthread.h” on page 38



## ptsname() — Get Name of the Slave Pseudoterminal Device

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#include <stdlib.h>
char *ptsname(int fildes);
```

### General Description

The `ptsname()` function returns the name of the slave pseudoterminal device associated with a master pseudoterminal device. The *fildes* argument is a file descriptor that refers to the master device. `ptsname()` returns a pointer to a string containing the path name of the corresponding slave device.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

Upon successful completion, `ptsname()` returns a pointer to a string which is the name of the pseudoterminal slave device. Upon failure, `ptsname()` returns a null pointer. This could occur if *fildes* is an invalid file descriptor or if the slave device name does not exist in the file system.

No errors are defined.

### Related Information

- “`grantpt()` — Grant Access to the Slave Pseudoterminal Device” on page 642
- “`open()` — Open a File” on page 872
- “`ttyname()` — Get the Name of a Terminal” on page 1632
- “`unlockpt()` — Unlock a Pseudoterminal Master/Slave Pair” on page 1662

## putc() - putchar() — Write a Character

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#define _ALL_SOURCE_NO_THREADS
```

```
#include <stdio.h>
```

```
int putc(int c, FILE *stream);
int putchar(int c);
```

### General Description

Converts *c* to unsigned char and then writes *c* to the output *stream* at the current position. The putchar() function is identical to:

```
putc(c, stdout);
```

These functions are also available as macros in the OS/390 C/C++ product. For performance purposes, it is recommended that the macro forms rather than the functional forms be used.

By default, if the stdio.h header file is included, the macro is invoked. Therefore, the stream argument expression should never be an expression with side effects.

The actual function can be accessed using one of the following methods:

- For C only: do *not* include stdio.h.
- Specify #undef, for example, #undef putc.
- Surround the function name by parentheses, for example: (putc)('a').

In a multithread application, in the presence of the feature test macro, \_OPEN\_THREADS, these macros are in an #undef status because they are not thread-safe.

putc() and putchar() are not supported for files opened with type=record.

putc() and putchar() have the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

If the application is not multi-threaded, then setting the \_ALL\_SOURCE\_NO\_THREADS feature test macro may improve performance of the application, because it allows use of the inline version of this function.

## Returned Value

The `putc()` and `putchar()` functions return the character written. A returned value of EOF indicates an error.

## Example CBC3BP54

```
/* CBC3BP54
   This example writes the contents of a buffer to a data stream.
   The body of the "for" statement is null because the example carries out
   the writing operation in the test expression.
*/
#include <stdio.h>
#include <string.h>
#define LENGTH 80

int main(void)
{
    FILE *stream = stdout;
    int i, ch;
    char buffer[LENGTH + 1] = "Hello world";

    /* This could be replaced by using the fwrite routine */
    for ( i = 0;
          (i < strlen(buffer)) && ((ch = putc(buffer[i], stream)) != EOF);
          ++i);
}
```

## Output

Hello world

## Related Information

- “stdio.h” on page 43
- “getc() - getchar() — Read a Character” on page 499
- “fputc() — Write a Character” on page 448
- “fwrite() — Write Items” on page 495

putenv() — Change or Add an Environment Variable

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>

int putenv(const char *envvar);
```

General Description

Adds a new environment variable or changes the value of an existing one. The argument *envvar* is a pointer to a null-terminated character string that should be of the form:

*name=value*

Where:

1. The first part, *name*, is a character string that represents the name of the environment variable. It is this part of the environment variable that putenv() will use when it searches the array of environment variable to determine whether to add or change this environment variable.
2. The second part, =, is a separator character (since the equal-sign is used as a separator character it cannot appear in the *name*).
3. The third part, *value*, is a null-terminated character string that represents the value that the environment variable, *name*, will be set to.

putenv() is a simplified form of setenv() and is equivalent to

setenv(*name*, *value*, 1)

**Note:** On this implementation, the storage used to define the environment variable, pointed to by *envvar* is not added to the array of environment variables. Instead the system copies the string into system allocated storage. However, a portable program should assume that the system will use the user's storage. Therefore, the storage used to define this environment variable should not be part of the caller's automatic storage.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

Special Behavior for POSIX C

You can use the external variable *\*\*environ* (defined as `extern char **environ`) to access the array of pointers to environment variables.

## Returned Value

If successful, `putenv()` returns a zero.

If unsuccessful, `putenv()` returns `-1`, and returns the error value in `errno`. The following are the possible values of `errno`:

`ENOMEM`    Insufficient memory was available.

## Special Behavior for OS/390 UNIX Services

`EINVAL`    The environment variable pointed to by the argument *envvar* does not follow the prescribed format. The equal-sign (=) separating the environment variable name from the value was not found.

## Related Information

- "C/370 Environmental Variables" in the *OS/390 C/C++ Programming Guide*
- "stdlib.h" on page 45
- "clearenv() — Clear Environment Variables" on page 184
- "\_\_\_getenv() — Get an Environment Variable" on page 517
- "getenv() — Get Value of Environment Variables" on page 515
- "setenv() — Add, Delete, and Change Environment Variables" on page 1215

## putmsg() - putpmsg() — Send a Message on a STREAM

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stropts.h>

int putmsg(int fildev, const struct strbuf *ctlptr,
           const struct strbuf *dataptr, int flags);

int putpmsg(int fildev, const struct strbuf *ctlptr,
            const struct strbuf *dataptr, int band, int flags);
```

### General Description

The `putmsg()` function creates a message from a process buffer(s) and sends the message to a STREAMS file. The message may contain either a data part, a control part, or both. The data and control parts are distinguished by placement in separate buffers, as described below. The semantics of each part is defined by the STREAMS module that receives the message.

The `putpmsg()` function does the same thing as `putmsg()`, but the process can send messages in different priority bands. Except where noted, all requirements on `putmsg()` also pertain to `putpmsg()`.

The *fildev* argument specifies a file descriptor referencing an open STREAM. The *ctlptr* and *dataptr* arguments each point to a **strbuf** structure.

The *ctlptr* argument points to the structure describing the control part, if any, to be included in the message. The **buf** member in the **strbuf** structure points to the buffer where the control information resides, and the **len** member indicates the number of bytes to be sent. The **maxlen** member is not used by `putmsg()`. In a similar manner, the argument *dataptr* specifies the data, if any, to be included in the message. The *flags* argument indicates what type of message should be sent and is described further below.

To send the data part of a message, *dataptr* must not be a null pointer and the **len** member of *dataptr* must be 0 or greater. To send the control part of a message, the corresponding values must be set for *ctlptr*. No data (control) part will be sent if either *dataptr* (*ctlptr*) is a null pointer or the **len** member of *dataptr* (*ctlptr*) is set to -1.

For `putmsg()`, if a control part is specified and *flags* is set to `RS_HIPRI`, a high priority message is sent. If no control part is specified, and *flags* is set to `RS_HIPRI`, `putmsg()` fails and sets `errno` to `EINVAL`. If *flags* is set to 0, a normal message (priority band equal to 0) is sent. If a control part and data part are not specified and *flags* is set to 0, no message is sent and 0 is returned.

The STREAM head guarantees that the control part of a message generated by `putmsg()` is at least 64 bytes in length.

For `putpmsg()`, the flags are different. The *flags* argument is a bitmask with the following mutually-exclusive flags defined: `MSG_HIPRI` and `MSG_BAND`. If *flags* is set to 0, `putpmsg()` fails and sets `errno` to `EINVAL`. If a control part is specified and *flags* is set to `MSG_HIPRI` and *band* is set to 0, a high-priority message is sent. If *flags* is set to `MSG_HIPRI` and either no control part is specified or *band* is set to a nonzero value, `putpmsg()` fails and sets `errno` to `EINVAL`. If *flags* is set to `MSG_BAND`, then a message is sent in the priority band specified by *band*. If a control part and data part are not specified and *flags* is set to `MSG_BAND`, no message is sent and 0 is returned.

The `putpmsg()` function blocks if the STREAM write queue is full due to internal flow control conditions, with the following exceptions:

- For high-priority messages, `putpmsg()` does not block on this condition and continues processing the message.
- For other messages, `putpmsg()` does not block but fails when the write queue is full and `O_NONBLOCK` is set.

The `putpmsg()` function also blocks, unless prevented by lack of internal resources, while waiting for the availability of message blocks in the STREAM, regardless of priority or whether `O_NONBLOCK` has been specified. No partial message is sent.

The following symbolic constants are defined under `_XOPEN_SOURCE_EXTENDED 1` in `<stropts.h>`.

<code>MSG_ANY</code>	Receive any message.
<code>MSG_BAND</code>	Receive message from specified band.
<code>MSG_HIPRI</code>	Send/Receive high priority message.
<code>MORECTL</code>	More control information is left in message.
<code>MOREDATA</code>	More data is left in message.

## Returned Value

If successful, `putmsg()` and `putpmsg()` return 0.

If unsuccessful, they return -1, and return the error value in `errno`.

**Note:** OS/390 UNIX services do not supply any STREAMS devices or pseudodevices. It is impossible for `putmsg()` and `putpmsg()` to send a message on a STREAM. It will always return -1 with `errno` set to indicate the failure. See “`open()` — Open a File” on page 872 for more information.

The following are the possible values of `errno`:

<code>EAGAIN</code>	A non-priority message was specified, the <code>O_NONBLOCK</code> flag is set, and the STREAM write queue is full due to internal flow control conditions; or buffers could not be allocated for the message that was to be created.
<code>EBADF</code>	<i>fildev</i> is not a valid file descriptor open for writing.
<code>EINTR</code>	A signal was caught during <code>putmsg()</code> .
<code>EINVAL</code>	An undefined value is specified in <i>flags</i> , or <i>flags</i> is set to <code>RS_HIPRI</code> or <code>MSG_HIPRI</code> and no control part is supplied, or the STREAM or multiplexer referenced by <i>fildev</i> is linked (directly or indirectly) down-

	stream from a multiplexer, or <i>flags</i> is set to MSG_HIPRI and <i>band</i> is nonzero (for <code>putpmsg()</code> only).
ENOSR	Buffers could not be allocated for the message that was to be created due to insufficient STREAMS memory resources.
ENOSTR	A STREAM is not associated with <i>fildev</i> .
ENXIO	A hang-up condition was generated downstream for the specified STREAM.
EPIPE or EIO	The <i>fildev</i> argument refers to a STREAMS-based pipe and the other end of the pipe is closed. A SIGPIPE signal is generated for the calling process.
ERANGE	The size of the data part of the message does not fall within the range specified by the maximum and minimum packet sizes of the topmost STREAM module. This value is also returned if the control part of the message is larger than the maximum configured size of the control part of a message, or if the data part of a message is larger than the maximum configured size of the data part of a message.

In addition, `putmsg()` and `putpmsg()` will fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of `errno` does not reflect the result of `putmsg()` or `putpmsg()` but reflects the prior error.

### Related Information

- “stropts.h” on page 46
- “`getmsg()` - `getpmsg()` — Receive Next Message from a STREAMS File” on page 555
- “`poll()` — Monitor Activity on File Descriptors and Message Queues” on page 910
- “`read()` — Read From a File or Socket” on page 1080
- “`readv()` — Read Data on a File or Socket and Store in a Set of Buffers” on page 1093
- “`write()` — Write Data on a File or Socket” on page 1780
- “`writv()` — Write Data on a File or Socket Socket from an Array” on page 1785



## puts() — Write a String

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
int puts(const char *string);
```

### General Description

Writes the string pointed to by *string* to the stream pointed to by `stdout`, and appends the new-line character to the output. The terminating null character is not written.

If `stdout` points to the text stream, and the output string is longer than the length of the stream's record, the output is *wrapped*. That is, the record is filled with the output characters, the last character of the record is set to a new-line character, and the remaining output characters are written to the next record. Such wrapping is repeated until the remaining output characters fit into the record. Please note that the new-line character is appended to the last portion of the output string. If the output string is shorter than the record, the remaining characters of the record are filled with blanks—if `stdout` is opened in a text mode—or with null characters if the `stdout` is opened in binary mode.

The `puts()` function is not supported for files opened with `type=record`.

`puts()` has the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

### Returned Value

Returns EOF if an error occurs; otherwise, `puts()` returns the number of bytes written. However, new-line characters used to wrap the data are not counted.

If a system write-error occurs, the write stops at the point of failure.

After truncation, `puts()` does not count the truncated characters, but returns the actual number of bytes written.

### Example

#### CBC3BP55

```
/* CBC3BP55
   This example writes "Hello World" to stdout.
*/
#include <stdio.h>

int main(void)
{
```

## puts

```
if ( puts("Hello World") == EOF )  
    printf( "Error in puts\n" );  
}
```

### Output

Hello World

### Related Information

- “stdio.h” on page 43
- “fputs() — Write a String” on page 450
- “gets() — Read a String” on page 595

## pututxline() — Write Entry to utmpx Database

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <utmpx.h>
```

```
struct utmpx *pututxline(const struct utmpx *utmpx);
```

### General Description

The `pututxline()` function writes out the structure into the utmpx database, when the calling process has appropriate privileges. The `pututxline()` function uses `getutxid()` to search for a record that satisfies the request. If the `getutxid()` search succeeds, then the entry is replaced. Otherwise, a new entry is made at the end of the database. If the utmpx database does not already exist, then `pututxline()` creates the utmpx database with file permissions 0644.

If the `ut_type` field in the entry being added is `EMPTY`, it is always placed at the start of the utmpx database. For this reason, `pututxline()` should not be used to place `EMPTY` entries in the utmpx database.

The functions `getutxent()`, `getutxid()`, and `getutxline()` obtain a shared lock on the database. The function `pututxline()` releases the shared lock, obtains an exclusive lock while writing to the database, releases the exclusive lock, and obtains the shared lock on the database. This lock is obtained using the `fcntl()` command **F\_SETLKW** on the first record in the database.

Because the `pututxline()` function processes thread specific data the `pututxline()` function can be used safely from a multi-threaded application. If multiple threads in the same process open the database, then each thread opens the database with a different file descriptor. The thread's database file descriptor is closed when the calling thread terminates or the `endutxent()` function is called by the calling thread.

The name of the database file defaults to `/etc/utmpx`. To process a different database file name use the `__utmpxname()` function.

`pututxline()` is not supported when all of the following conditions are true:

- The security environment for the current address space has the trusted attribute.
- Either the effective UID is different than the real UID, or the effective GID is different than the real GID.
- The effective UID is not 0
- The utmpx file is not writable by normal (non-trusted) processes with the current effective UID and GID.
- `pututxline()` is called after `getutxline()`, `getutxid()`, or `getutxent()`, with no intervening calls to `endutxent()` or `__utmpxname()`.

For all entries that match a request, the `ut_type` member indicates the type of the entry. Other members of the entry will contain meaningful data based on the value of the `ut_type` member as follows:

<b>EMPTY</b>	No other members have meaningful data.
<b>BOOT_TIME</b>	<code>ut_tv</code> is meaningful.
<b>__RUN_LVL</b>	<code>ut_tv</code> and <code>ut_line</code> are meaningful
<b>OLD_TIME</b>	<code>ut_tv</code> is meaningful.
<b>NEW_TIME</b>	<code>ut_tv</code> is meaningful.
<b>USER_PROCESS</b>	<code>ut_id</code> , <code>ut_user</code> (login name of the user), <code>ut_line</code> , <code>ut_pid</code> , and <code>ut_tv</code> are meaningful.
<b>INIT_PROCESS</b>	<code>ut_id</code> , <code>ut_pid</code> , and <code>ut_tv</code> are meaningful.
<b>LOGIN_PROCESS</b>	<code>ut_id</code> , <code>ut_user</code> (implementation-specific name of the login process), <code>ut_pid</code> , and <code>ut_tv</code> are meaningful.
<b>DEAD_PROCESS</b>	<code>ut_id</code> , <code>ut_pid</code> , and <code>ut_tv</code> are meaningful.

### Returned Value

If successful, `pututxline()` returns a pointer to a `utmpx` structure containing a copy of the entry written to the database. Otherwise a null pointer is returned.

The `pututxline()` function may fail if the process does not have appropriate privileges.

### Related Information

- “`endutxent()` — Close the `utmpx` Database” on page 313
- “`getutxent()` — Read Next Entry in `utmpx` Database” on page 620
- “`getutxid()` — Search by ID `utmpx` Database” on page 622
- “`getutxline()` — Search by Line `utmpx` Database” on page 624
- “`setutxent()` — Reset to Start of `utmpx` Database” on page 1282
- “`__utmpxname()` — Change the `utmpx` Database Name” on page 1669

## putw() — Put a Machine Word on a Stream

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <stdio.h>
```

```
int putw(int w, FILE *stream);
```

### General Description

The `putw()` function writes the word *w* to the output *stream* (at the position at which the file offset, if defined, is pointing). The size of the word is the size of a type `int`, and varies from machine to machine. The `putw()` function neither assumes nor causes special alignment in the file. The *st\_ctime* and *st\_mtime* fields of the file will be marked for update between the successful execution of `putw()` and the next successful call to `fflush()` or `fclose()` on the same stream or a call to `exit()` or `abort()`.

### Returned Value

If successful, `putw()` returns 0.

If unsuccessful, a nonzero value is returned, the error indicators for *stream* are set, and `errno` is set to indicate the error. Refer to “`fread()` — Read Items” on page 456 for `errno` values.

### Related Information

- “`stdio.h`” on page 43
- “`fopen()` — Open a File” on page 417
- “`fwrite()` — Write Items” on page 495
- “`getw()` — Get a Machine Word from a Stream” on page 626

## putwc() — Output a Wide Character

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

#### Non-XPG4

```
#include <stdio.h>
#include <wchar.h>

wint_t putwc(wchar_t wc, FILE *stream);
```

#### XPG4

```
#define _XOPEN_SOURCE
#include <stdio.h>
#include <wchar.h>

wint_t putwc(wint_t wc, FILE *stream);
```

#### XPG4 and MSE

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <stdio.h>
#include <wchar.h>

wint_t putwc(wchar_t wc, FILE *stream);
```

### General Description

The `putwc()` function is equivalent to the `fputwc()` function, except that if it is implemented as a macro, it may evaluate *stream* more than once. Therefore, the argument should never be an expression with side effects. The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you use a non-wide-oriented function with `putwc()`, undefined results can occur.

`putwc()` has the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

### Special Behavior for XPG4

If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the `wchar` header, then the compiler assumes that your program is using the XPG4 variety of the `putwc()` function, unless you also define the `_MSE_PROTOS` feature test macro. Please see Table 3 on page 18 for a list of XPG4 and other feature test macros.

The prototype for the XPG4 variety of the `putwc()` function is:

```
wint_t putwc(wint_t wc, FILE *stream);
```

The difference between this variety and the MSE variety of the `putwc()` function is that the first parameter has type `wint_t` rather than type `wchar_t`.

## Returned Value

Returns the wide character written. If a write error occurs, the error indicator for the stream is set and `WEOF` is returned. If an encoding error occurs when converting from a wide character to a multibyte character, the value of the macro `EILSEQ` is stored in `errno` and `WEOF` is returned.

## Example CBC3BP56

```
/* CBC3BP56 */
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
    FILE    *stream;
    wchar_t *wcs = L"This test string should not cause a WEOF condition";
    int      i;
    int      rc;

    if ((stream = fopen("myfile.dat", "w")) == NULL) {
        printf("Unable to open file\n");
        exit(1);
    }

    for (i=0; wcs[i] != L'\0'; i++) {
        errno = 0;
        if ((rc = putwc(wcs[i], stream)) == WEOF) {
            printf("Unable to putwc() the wide character.\n");
            printf("wcs[%d] = 0x%X\n", i, wcs[i]);
            if (errno == EILSEQ)
                printf("An invalid wide character was encountered.\n");
            exit(1);
        }
    }

    fclose(stream);
}
```

## Related Information

- “`stdio.h`” on page 43
- “`wchar.h`” on page 54
- “`fputwc()` — Output a Wide-Character” on page 452

## putwchar() — Output a Wide Character to Standard Output

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

#### Non-XPG4

```
#include <stdio.h>
#include <wchar.h>

wint_t putwchar(wchar_t wc);
```

#### XPG4

```
#define _XOPEN_SOURCE
#include <stdio.h>
#include <wchar.h>

wint_t putwchar(wint_t wc);
```

#### XPG4 and MSE

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <stdio.h>
#include <wchar.h>

wint_t putwchar(wchar_t wc);
```

### General Description

The `putwchar()` function is equivalent to: `putc()(wc stdout)`.

The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you use a non-wide-oriented function with `putwchar()`, undefined results can occur.

`putwchar()` has the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

You may not use `putc()` or `putwchar()` with files opened as `type=record`.

### Special Behavior for XPG4

If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the `wchar` header, then the compiler assumes that your program is using the XPG4 variety of the `putwchar()` function, unless you also define the `_MSE_PROTOS` feature test macro. Please see Table 3 on page 18 for a list of XPG4 and other feature test macros.



The prototype for the XPG4 variety of the `putwchar()` function is:

```
wint_t putwchar(wint_t wc);
```

The difference between this variety and the MSE variety of the `putwchar()` function is that its parameter has type `wint_t` rather than type `wchar_t`.

## Returned Value

Returns the wide character written. If a write error occurs, the error indicator for the stream is set and `WEOF` is returned. If an encoding error occurs when converting from a wide character to a multibyte character, the value of the macro `EILSEQ` is stored in `errno` and `WEOF` is returned.

## Example

### CBC3BP57

```
/* CBC3BP57 */
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
    wchar_t *wcs = L"This test string should not cause a WEOF condition";
    int i;
    int rc;

    for (i=0; wcs[i] != L'\0'; i++) {
        errno = 0;
        if ((rc = putwchar(wcs[i])) == WEOF) {
            printf("Unable to putwchar() the wide character.\n");
            printf("wcs[%d] = 0x%X\n", i, wcs[i]);
            if (errno == EILSEQ)
                printf("An invalid wide character was encountered.\n");
            exit(1);
        }
    }
}
```

## Related Information

- “`wchar.h`” on page 54
- “`fputwc()` — Output a Wide-Character” on page 452

## qsort() — Sort Array

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
void qsort(void *base, size_t num, size_t width,
           int(*compare)(const void *element1, const void *element2));
```

### General Description

Sorts an array of *num* elements, each of *width* bytes in size. The *base* pointer is a pointer to the array to be sorted. The `qsort()` function overwrites the contents of the array with the sorted elements.

The *compare* pointer points to a function, which you supply, that compares two array elements and returns an integer value specifying their relationship. The `qsort()` function calls the `compare()` function one or more times during the sort, passing pointers to two array elements on each call. The function must compare the elements, and then it returns one of the following values:

Value	Meaning
Less than 0	<i>element1</i> less than <i>element2</i>
0	<i>element1</i> equal to <i>element2</i>
Greater than 0	<i>element1</i> greater than <i>element2</i>

The sorted array elements are stored in increasing order, as defined by your comparison function. You can sort in reverse order by reversing the sense of “greater than” and “less than” in the comparison function. If two elements are equal, their order in the sorted array is unspecified.

### Special Behavior for C++

C++ and C linkage conventions are incompatible, and therefore `qsort()` cannot receive C++ function pointers. If you attempt to pass a C++ function pointer to `qsort()`, the compiler will flag it as an error. To use the C++ `qsort()` function, you must ensure that the `compare()` function has C linkage, by declaring it as `extern "C"`.

### Returned Value

There is no returned value.

## Example

### CBC3BQ01

```

/* CBC3BQ01
   This example sorts the arguments (argv) in ascending sequence, based on
   the ASCII value of each character and string, and using the comparison
   function compare() supplied in the example.
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Declaration of compare() as a function */
#ifdef __cplusplus
extern "C" int compare(const void *, const void *);
#else
int compare(const void *, const void *); /* macro is automatically */
#endif                                /* defined by the C++/MVS compiler */

int main (int argc, char *argv[ ])
{
    int i;
    argv++;
    argc--;
    qsort((char *)argv, argc, sizeof(char *), compare);
    for (i = 0; i < argc; ++i)
        printf("%s\n", argv[i]);
    return 0;
}

int compare (const void *arg1, const void *arg2)
{
    /* Compare all of both strings */
    return(strcmp(*(char **)arg1, *(char **)arg2));
}

```

## Output

If the program is passed the arguments:

Does, this, really, sort, the, arguments, correctly?

then expect the following output.

```

arguments
correctly?
really
sort
the
this
Does

```

## Related Information

- “stdlib.h” on page 45
- “bsearch() — Search Arrays” on page 137

## QueryMetrics() — Query WLM System Information

### Standards

Standards / Extensions	C or C++	Dependencies
	both	

### Format

```
#include <sys/___wlm.h>
int QueryMetrics(struct sysi *sysi_ptr,
                 int *anslen);
```

### General Description

The QueryMetrics() function provides the ability for an application to query WLM system information.

- \*sysi\_ptr**      Points to a buffer that the service is to return the WLM system information. The data returned is in the format of the structure sysi.
- \*anslen**        Points to an integer data field that contains the length of the buffer to return the WLM system information into.

### Returned Value

Upon successful completion QueryMetrics() returns a zero. If the function is unsuccessful, -1 is returned and errno is set to indicate the error. If the returned errno and \_\_errno2() indicate the supplied buffer is too small, the anslen argument is updated to contain the length required to hold WLM system information. Possible values of errno are:

- EFAULT**        An argument of this function contained an address that was not accessible to the caller.
- EINVAL**        An argument of this function contained an incorrect value.
- EMVSWLMERROR**  
The WLM query system information failed. Use \_\_errno2() to obtain the WLM service reason code for the failure.
- EPERM**        The calling thread's address space is not permitted to the BPX.WLMSEVER Facility class. The caller's address space must be permitted to the BPX.WLMSEVER Facility class if it is defined. If BPX.WLMSEVER is not defined, the calling process is not defined as a superuser (UID=0).
- EMVSSAF2ERR**  
An error occurred in the security product.

### Related Information

- “sys/\_\_\_wlm.h” on page 51
- “CheckSchEnv() — Check WLM Scheduling Environment” on page 172
- “ConnectServer() — Connect to WLM as a Server Manager” on page 219
- “ConnectWorkMgr() — Connect to WLM as a Work Manager” on page 221
- “ContinueWorkUnit() — Continue WLM Work Unit” on page 225
- “CreateWorkUnit() — Create WLM Work Unit” on page 236
- “DeleteWorkUnit() — Delete a WLM Work Unit” on page 272
- “DisconnectServer() — Disconnect from WLM Server” on page 276
- “JoinWorkUnit() — Join a WLM Work Unit” on page 723

- “LeaveWorkUnit() — Leave a WLM Work Unit” on page 742
- “QuerySchEnv() — Query WLM Scheduling Environment” on page 1072

## QuerySchEnv() — Query WLM Scheduling Environment

### Standards

Standards / Extensions	C or C++	Dependencies
	both	

### Format

```
#include <sys/_wlm.h>
int QuerySchEnv(struct sethdr *sysi_ptr,
                int *anslen);
```

### General Description

The QuerySchEnv() function provides the ability for an application to query WLM scheduling environment.

- \*sysi\_ptr**      Points to a buffer that the service is to return the WLM system information. The data returned is in the format of the structure sysi.
- \*anslen**        Points to an integer data field that contains the length of the buffer to return the WLM system information into.

### Returned Value

Upon successful completion QuerySchEnv() returns a zero. If the function is unsuccessful, -1 is returned and errno is set to indicate the error. If the returned errno and \_\_errno2() indicate the supplied buffer is too small, the anslen argument is updated to contain the length required to hold WLM system information. Possible values of errno are:

- EFAULT**        An argument of this function contained an address that was not accessible to the caller.
- EINVAL**        An argument of this function contained an incorrect value.
- EMVSWLMERROR**  
The WLM query scheduling environment failed. Use \_\_errno2() to obtain the WLM service reason code for the failure.
- EPERM**        The calling thread's address space is not permitted to the BPX.WLMSEVER Facility class. The caller's address space must be permitted to the BPX.WLMSEVER Facility class if it is defined. If BPX.WLMSEVER is not defined, the calling process is not defined as a superuser (UID=0).
- EMVSSAF2ERR**  
An error occurred in the security product.

### Related Information

- “sys/\_wlm.h” on page 51
- “CheckSchEnv() — Check WLM Scheduling Environment” on page 172
- “ConnectServer() — Connect to WLM as a Server Manager” on page 219
- “ConnectWorkMgr() — Connect to WLM as a Work Manager” on page 221
- “ContinueWorkUnit() — Continue WLM Work Unit” on page 225
- “CreateWorkUnit() — Create WLM Work Unit” on page 236
- “DeleteWorkUnit() — Delete a WLM Work Unit” on page 272
- “DisconnectServer() — Disconnect from WLM Server” on page 276
- “JoinWorkUnit() — Join a WLM Work Unit” on page 723

- “LeaveWorkUnit() — Leave a WLM Work Unit” on page 742
- “QueryMetrics() — Query WLM System Information” on page 1070

## raise() — Raise Signal

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <signal.h>
```

```
int raise(int sig);
```

### General Description

Sends the signal *sig* to the program that issued `raise()`. See Table 33 on page 1332 for the list of signals supported.

You can use `signal()` to specify how a signal will be handled when `raise()` is invoked.

In C++ only, the use of `signal()` and `raise()` with `try()`, `catch()`, or `throw()` is undefined. The use of `signal()` and `raise()` with destructors is also undefined.

### Special Behavior for Posix

To obtain access to the special Posix behavior for `raise()`, the POSIX runtime option must be set ON, and the version of MVS must be 4.3 or higher.

The `raise()` function sends the signal, *sig*, to the process that issued the `raise()`. If the signal is not blocked, it is delivered to the sender before `raise()` returns. See Table 32 on page 1295 in the description of the `sigaction()` function for the list of signals supported.

You can use `signal()` or `sigaction()` to specify how a signal will be handled when `raise()` is invoked.

### Special Behavior for XPG4.2

To obtain access to the special Posix behavior for `raise()`, the POSIX runtime option must be set ON, and the version of MVS must be 4.3 or higher.

Several other functions are available to the XPG4.2 application for affecting the behavior of a signal:

```
bsd_signal()
sigignore()
sigset()
```

### Special Behavior for C++

The behavior when mixing signal-handling with C++ exception handling is undefined. Also, the use of signal-handling with constructors and destructors is undefined.



## Returned Value

The returned value is 0 if successful; nonzero if unsuccessful.

## Special Behavior for XPG4:

`raise()` returns the error value in `errno`. The following are the possible values of `errno`:

`EINVAL`      The value of the *sig* argument is an invalid signal number.

## Example CBC3BR01

```
/* CBC3BR01
   This example establishes a signal handler called sig_hand for the signal SIGUSR1.
   The signal handler is called whenever the SIGUSR1 signal is raised and
   will ignore the first nine occurrences of the signal.
   On the tenth raised signal, it exits the program with an error code of 10.
   Note that the signal handler must be reestablished each time it is called.
*/
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

void sig_hand(int); /* declaration of sig_hand() as a function */
int i;

int main(void)
{
    signal(SIGUSR1, sig_hand); /* set up handler for SIGUSR1 */
    for (i=0; i<10; ++i)
        raise(SIGUSR1);          /* signal SIGUSR1 is raised */
    /* sig_hand() is called */

#ifdef __cplusplus
    extern "C" {
#endif
    void sig_hand(int);
#ifdef __cplusplus
    }
#endif

    {
        static int count = 0;          /* initialized only once */

        count++;
        if (count == 10) { /* ignore the first 9 occurrences of this signal */
            printf("reached 10th signal\n");
            exit(10);
        }
        else
            signal(SIGUSR1, sig_hand); /* set up the handler again */
    }
}
```

## **Related Information**

- “signal.h” on page 41
- “kill() — Send a Signal to a Process” on page 728
- “bsd\_signal() — BSD Version of signal()” on page 135
- “pthread\_kill() — Send a Signal to a Thread” on page 984
- “killpg() — Send a Signal to a Process Group” on page 731
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sighold() — Add a Signal to a Thread” on page 1322
- “sigignore() — Set Disposition to Ignore a Signal” on page 1323
- “signal() — Handle Interrupts” on page 1330
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “sigset() — Change a Signal Action and/or a Thread” on page 1343

## rand() — Generate Random Number

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
int rand(void);
```

### General Description

Generates a pseudorandom integer in the range 0 to `RAND_MAX`. Use the `srand()` function before calling `rand()` to set a seed for the random number generator. If you do not make a call to `srand()`, the default seed is 1.

### Returned Value

Returns the calculated value.

### Example CBC3BR02

```
/* CBC3BR02
   This example prints the first 10 random numbers generated.
*/
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int x;

    for (x = 1; x <= 10; x++)
        printf("iteration %d, rand=%d\n", x, rand());
}
```

### Output

```
iteration 1, rand=16838
iteration 2, rand=5758
iteration 3, rand=10113
iteration 4, rand=17515
iteration 5, rand=31051
iteration 6, rand=5627
iteration 7, rand=23010
iteration 8, rand=7419
iteration 9, rand=16212
iteration 10, rand=4086
```

**Related Information**

- “stdlib.h” on page 45
- “srand() — Set Seed for rand() Function” on page 1398

## random() — A Better Random-Number Generator

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>
```

```
long random(void);
```

### General Description

The `random()` function uses a nonlinear additive feedback random-number generator employing a default state array size of 31 long integers to return successive pseudorandom numbers in the range from 0 to  $2^{31}-1$ . The period of this random-number generator is approximately  $16 \times (2^{31}-1)$ . The size of the state array determines the period of the random-number generator. Increasing the state array size increases the period.

With 256 bytes of state information, the period of the random-number generator is greater than  $2^{69}$ .

Like `rand()`, `random()` produces by default a sequence of numbers that can be duplicated by calling `srandom()` with 1 as the seed. The state information for the random functions is maintained on a per-thread basis. For example, calls to `srandom()` in one thread will have no effect on the numbers generated by calls to `random()` in another thread.

### Returned Value

`random()` returns the generated pseudorandom number.

### Related Information

- “`stdlib.h`” on page 45
- “`drand48()` — Pseudo-random Number Generator” on page 286
- “`initstate()` — Initialize Generator for `Random()`” on page 670
- “`setstate()` — Change Generator for `random()`” on page 1275
- “`srandom()` — Use Seed to Initialize Generator for `random()`” on page 1400

## read() — Read From a File or Socket

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
ssize_t read(int fs, void *buf, size_t N);
```

### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
```

```
ssize_t read(int fs, void *buf, ssize_t N);
```

### Berkeley Sockets

```
#define _OE_SOCKETS
#include <unistd.h>
```

```
ssize_t read(int socket, void *buf, ssize_t N);
```

### General Description

From the file indicated by the file descriptor *fs*, the read() function reads *N* bytes of input into the memory area indicated by *buf*. A successful read() updates the access time for the file.

If *fs* refers to a regular file or any other type of file on which the process can seek, read() begins reading at the file offset associated with *fs*. If successful, read() changes the file offset by the number of bytes read. *N* should not be greater than INT\_MAX (defined in the limits.h header file).

If *fs* refers to a file on which the process cannot seek, read() begins reading at the current position. There is no file offset associated with such a file.

If *fs* refers to a socket, read() is equivalent to recv() with no flags set.

#### Parameter      Description

<i>fs</i>	The file or socket descriptor.
<i>buf</i>	The pointer to the buffer that receives the data.
<i>N</i>	The length in bytes of the buffer pointed to by the <i>buf</i> parameter.

### Behavior for Sockets

The read() call reads data on a socket with descriptor *fs* and stores it in a buffer. The read() call applies only to connected sockets. This call returns up to *N* bytes of data. If there are fewer bytes available than requested, the call returns the number currently available. If data is not available for the socket *fs*, and the socket is in

blocking mode, the `read()` call blocks the caller until data arrives. If data is not available, and the socket is in nonblocking mode, `read()` returns a `-1` and sets the error code to `EWOULDBLOCK`. See “`ioctl()` — Control Device” on page 672 or “`fcntl()` — Control Open File Descriptors” on page 350 for a description of how to set non-blocking mode.

For datagram sockets, this call returns the entire datagram that was sent, provided that the datagram fits into the specified buffer. Excess datagram data is discarded. Stream sockets act like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and application A sends 1000 bytes, each call to this function can return 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been received.

For sockets that are defined as `AF_INET` and `SOCK_DGRAM` type sockets, bulk mode I/O will be supported only after the socket has been connected and the `setsockopt()` or `sock_do_bulkmode()` function is issued to set a socket for bulk mode use.

### Behavior for Streams

A `read()` from a STREAMS file can read data in three different modes: byte-stream mode, message-nondiscard mode, and message-discard mode. The default is byte-stream mode. This can be changed using the `I_SRDOPT` `ioctl()` request, and can be tested with the `I_GRDOPT` `ioctl()`. In byte-stream mode, `read()` retrieves data from the STREAM until as many bytes as were requested are transferred, or until there is no more data to be retrieved. Byte-stream mode ignores message boundaries.

In STREAMS message-nondiscard mode, `read()` retrieves data until as many bytes as were requested are transferred, or until a message boundary is reached. If `read()` does not retrieve all the data in a message, the remaining data is left on the STREAM, and can be retrieved by the next `read()` call. Message-discard mode also retrieves data until as many bytes as were requested are transferred, or a message boundary is reached. However, unread data remaining in a message after the `read()` returns is discarded, and is not available for a subsequent `read()`, `readv()` or `getmsg()` call.

How `read()` handles zero-byte STREAMS messages is determined by the current read mode setting. In byte-stream mode, `read()` accepts data until it has read `N` bytes, or until there is no more data to read, or until a zero-byte message block is encountered. The `read()` function then returns the number of bytes read, and places the zero-byte message back on the STREAM to be retrieved by the next `read()`, `readv()` or `getmsg()`. In message-nondiscard mode or message-discard mode, a zero-byte message returns 0 and the message is removed from the STREAM. When a zero-byte message is read as the first message on a STREAM, the message is removed from the STREAM and 0 is returned, regardless of the read mode.

A `read()` from a STREAMS file returns the data in the message at the front of the STREAM head read queue, regardless of the priority band of the message.

By default, STREAMS are in control-normal mode, in which a `read()` from a STREAMS file can only process messages that contain a data part but do not contain a control part. The `read()` fails if a message containing a control part is

encountered at the STREAM head. This default action can be changed by placing the STREAM in either control-data mode or control-discard mode with the `I_SRDOPT` `ioctl()` command. In control-data mode, `read()` converts any control part to data and passes it to the application before passing any data part originally present in the same message. In control-discard mode, `read()` discards message control parts but returns to the process any data part in the message.

In addition, `read()` and `readv()` will fail if the STREAM head had processed an asynchronous error before the call. In this case, the value of `errno` does not reflect the result of `read()` or `readv()` but reflects the prior error. If a hang-up occurs on the STREAM being read, `read()` continues to operate normally until the STREAM head read queue is empty. Thereafter, it returns 0.

### Returned Value

If successful, `read()` returns the number of bytes actually read and placed in *buf*. This number is less than or equal to *N*. It is less than *N* only if:

- `read()` reached the end of the file before reading the requested number of bytes.
- `read()` was interrupted by a signal.
- In POSIX C programs only, the file is a pipe, FIFO special file, or a character special file that has fewer than *N* bytes immediately available for reading. (See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information about using POSIX support.)

In POSIX C programs only, if `read()` is interrupted by a signal, the effect is one of the following:

- If `read()` has not read any data yet, it returns the value `-1` and sets `errno` to `EINTR`.
- If `read()` has successfully read some data, it returns the number of bytes it read before it was interrupted.

If the starting position for the read operation is at the end of the file or beyond, `read()` returns the value 0.

In POSIX C programs, if `read()` attempts to read from an empty pipe or a FIFO special file, it has one of the following results:

- If no process has the pipe open for writing, `read()` returns the value 0 to indicate the end of the file.
- If some process has the pipe open for writing and `O_NONBLOCK` is set to 1, `read()` returns the value `-1` and sets `errno` to `EAGAIN`.
- If some process has the pipe open for writing and `O_NONBLOCK` is set to 0, `read()` blocks (that is, does not return) until some data is written, or the pipe is closed by all other processes that have the pipe open for writing.

With other files that support nonblocking read operations (for example, character special files), a similar principle applies:

- If data is available, `read()` reads the data immediately.
- If no data is available and `O_NONBLOCK` is set to 1, `read()` returns the value `-1` and sets `errno` to `EAGAIN`.



- If no data is available and `O_NONBLOCK` is set to 0, `read()` blocks until some data becomes available.

`read()` causes the signal `SIGTTIN` to be sent when all these conditions exist:

- The process is attempting to read from its controlling terminal.
- The process is running in a background process group.
- The `SIGTTIN` signal is not blocked or ignored.
- The process group of the process is not orphaned.

If `read()` is reading a regular file and encounters a part of the file that has not been written (but before the end of the file), `read()` places 0 bytes into *buf* in place of the unwritten bytes.

If the number of bytes of input that you want to read is 0, `read()` simply returns a value of 0 without attempting any other action.

If the connection is broken on a stream socket, but data is available, then `read()` reads the data and gives no error on the first read operation. If the connection is broken on a stream socket, but no data is available, then `read()` returns 0 bytes as EOF on the first read operation.

**Note:** OS/390 UNIX services do not supply any STREAMS devices or pseudodevices. It is impossible for `read()` to read any data from a STREAMS-based file indicated by *fs*. It will always return -1 with `errno` set to `EBADF`. `EINVAL` will never be set because there are no multiplexing STREAMS drivers. See “`open()` — Open a File” on page 872 for more information.

If `read()` fails, it returns the value -1 and sets `errno` to one of the following:

<code>EAGAIN</code>	<code>O_NONBLOCK</code> is set to 1, but data was not available for reading.
<code>EBADF</code>	<i>fs</i> is not a valid file or socket descriptor.
<code>ECONNRESET</code>	A connection was forcibly closed by a peer.
<code>EFAULT</code>	Using the <i>buf</i> and <i>N</i> parameters would result in an attempt to access memory outside the caller's address space.
<code>EINTR</code>	<code>read()</code> was interrupted by a signal that was caught before any data was available.
<code>EINVAL</code>	<i>N</i> contains a value that is less than 0, or the request is invalid or not supported, or the STREAM or multiplexer referenced by <i>fs</i> is linked (directly or indirectly) downstream from a multiplexer.
<code>EIO</code>	The process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the <code>SIGTTIN</code> signal or the process group of the process is orphaned. For sockets, an I/O error occurred.
<code>ENOBUFS</code>	Insufficient system resources are available to complete the call.
<code>ENOTCONN</code>	A receive was attempted on a connection-oriented socket that is not connected.
<code>ETIMEDOUT</code>	The connection timed out during connection establishment, or due to a transmission timeout on active connection.

**EWouldBlock**

The socket is in nonblocking mode and data is not available to read.

### Example

#### CBC3BR03

```
/* CBC3BR03
   This example opens a file and reads input.
*/
#define _POSIX_SOURCE
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

main() {
    int ret, fd;
    char buf[1024];

    system("ls -l / > ls.output");

    if ((fd = open("ls.output", O_RDONLY)) < 0)
        perror("open() error");
    else {
        while ((ret = read(fd, buf, sizeof(buf)-1)) > 0) {
            buf[ret] = 0x00;
            printf("block read: \n<%s>\n", buf);
        }
        close(fd);
    }

    unlink("ls.output");
}
```

**Output**

```
block read:
<total 0
drwxr-xr-x  3 USER1  SYS1          0 Apr 16 07:59 bin
drwxr-xr-x  2 USER1  SYS1          0 Apr  6 10:20 dev
drwxr-xr-x  4 USER1  SYS1          0 Apr 16 07:59 etc
drwxr-xr-x  2 USER1  SYS1          0 Apr  6 10:15 lib
drwxrwxrwx  2 USER1  SYS1          0 Apr 16 07:55 tmp
drwxr-xr-x  2 USER1  SYS1          0 Apr  6 10:15 u
drwxr-xr-x  6 USER1  SYS1          0 Apr  6 10:15 usr
>
```

**Related Information**

- “limits.h” on page 32
- “unistd.h” on page 53
- “close() — Close a File” on page 191
- “connect() — Connect a Socket” on page 214
- “creat() — Create a New File or Rewrite an Existing One” on page 233
- “dup() — Duplicate an Open File Descriptor” on page 288
- “fcntl() — Control Open File Descriptors” on page 350
- “fread() — Read Items” on page 456
- “getsockopt() — Get the Options Associated with a Socket” on page 606
- “ioctl() — Control Device” on page 672
- “lseek() — Change the Offset of a File” on page 776
- “open() — Open a File” on page 872

- “pipe() — Create an Unnamed Pipe” on page 907
- “readv() — Read Data on a File or Socket and Store in a Set of Buffers” on page 1093
- “recv() — Receive Data on a Socket” on page 1103
- “recvfrom() — Receive Messages on a Socket” on page 1106
- “recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers” on page 1110
- “select() — Monitor Activity on Files/Sockets and Message Queues” on page 1159
- “selectex() — Monitor Activity on Files/Sockets and Message Queues” on page 1166
- “send() — Send Data on a Socket” on page 1178
- “sendmsg() — Send Messages on a Socket” on page 1185
- “sendto() — Send Data on a Socket” on page 1190
- “setsockopt() — Set Options Associated with a Socket” on page 1270
- “socket() — Create a Socket” on page 1371
- “write() — Write Data on a File or Socket” on page 1780
- “writev() — Write Data on a File or Socket Socket from an Array” on page 1785

readdir() — Read an Entry from a Directory

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

Format

```
#define _POSIX_SOURCE
#include <dirent.h>
```

```
struct dirent *readdir(DIR *dir);
```

General Description

Returns a pointer to a `dirent` structure describing the next directory entry in the directory stream associated with *dir*.

A call to `readdir()` overwrites data produced by a previous call to `readdir()` or `__readdir2()` on the same directory stream. Calls for different directory streams do not overwrite each other's data.

Each call to `readdir()` updates the `st_atime` (access time) field for the directory.

A `dirent` structure contains the character pointer *d\_name*, which points to a string that gives the name of a file in the directory. This string ends in a terminating null, and has a maximum of `NAME_MAX` characters.

Save the data from `readdir()`, if required, before calling `closedir()`, because `closedir()` frees the data.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

Special Behavior for XPG4

If entries for dot or dot-dot exist, one entry will be returned for dot and one entry will be returned for dot-dot; otherwise they will not be returned.

After a call to `fork()`, either the parent or child (but not both) may continue processing the directory stream using `__readdir2()`, `readdir()`, `rewinddir()`, or `seekdir()`. If both the parent and child processes use these functions, the result is undefined.

Special Behavior for XPG4.2

If the entry names a symbolic link, the value of `d_ino` member in *dirent* structure is unspecified.

## Returned Value

If successful, `readdir()` returns a pointer to a `dirent` structure describing the next directory entry in the directory stream. When `readdir()` reaches the end of the directory stream, it returns a NULL pointer.

If unsuccessful, `readdir()` returns a NULL pointer and sets `errno` to one of the following:

`EBADF`            *dir* does not yield an open directory stream.  
`EINVAL`           The buffer was too small to contain any directories.

## Special Behavior for XPG4.2

`ENOENT`          The current position of the directory stream is invalid.

## Example CBC3BR04

```
/* CBC3BR04
   This example reads the contents of a root directory.
   */
#define _POSIX_SOURCE
#include <dirent.h>
#include <errno.h>
#include <sys/types.h>
#include <stdio.h>

main() {
    DIR *dir;
    struct dirent *entry;

    if ((dir = opendir("/")) == NULL)
        perror("opendir() error");
    else {
        puts("contents of root:");
        while ((entry = readdir(dir)) != NULL)
            printf(" %s\n", entry->d_name);
        closedir(dir);
    }
}
```

## Output

```
contents of root:
.
..
bin
dev
etc
lib
tmp
u
usr
```

## Related Information

- “dirent.h” on page 24
- “stdio.h” on page 43
- “sys/types.h” on page 49
- “closedir() — Close a Directory” on page 194
- “opendir() — Open a Directory” on page 877
- “\_\_opendir2() — Open a Directory” on page 880
- “\_\_readdir2() — Read Directory Entry and Get File Information” on page 1089
- “rewinddir() — Reposition a Directory Stream to the Beginning” on page 1141
- “seekdir() — Set Position of Directory Stream” on page 1158
- “telldir() — Current Location of Directory Stream” on page 1557

## \_\_readdir2() — Read Directory Entry and Get File Information

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R6

### Format

```
#define _OPEN_SYS_DIR_EXT
#include <dirent.h>
```

```
struct dirent *__readdir2(DIR *dir, struct stat *info);
```

### General Description

The `__readdir2()` function returns a pointer to a `dirent` structure describing the next directory entry in the directory stream associated with *dir*.

A `dirent` structure contains the character pointer *d\_name*, which points to a string that gives the name of a file in the directory. This string ends in a terminating null, and has a maximum of `NAME_MAX` characters.

The *info* argument points to an area of storage that will be filled in with information about the file *d\_name*. This information is returned in a `stat` structure defined in the `sys/stat.h` header file. The format of this structure is described in the section about the `lstat()` function. If *info* is `NULL`, no `stat` information is passed back.

If entries for dot or dot-dot exist, one entry will be returned for dot and one entry will be returned for dot-dot; otherwise they will not be returned.

A call to `__readdir2()` overwrites data produced by a previous call to `__readdir2()` or `readdir()` on the same directory stream. Calls for different directory streams do not overwrite each other's data.

Save the `dirent` data from `__readdir2()`, if required, before calling `closedir()`, because `closedir()` frees the `dirent` data.

The `__readdir2()` function may buffer several directory entries per actual read operation. `__readdir2()` updates the `st_atime` (access time) field of the directory each time the directory is actually read.

After a call to `fork()`, either the parent or child (but not both) may continue processing the directory stream using `__readdir2()`, `readdir()`, `rewinddir()` or `seekdir()`. If both the parent and child processes use these functions, the result is undefined.

If the entry names a symbolic link, the value of `d_ino` member in *dirent* structure is unspecified.

Unpredictable results can occur if `closedir()` is used to close the the directory stream before `__readdir2()` is called. Unpredictable results can also occur if files are added to or removed from the directory after it is opened and before `__readdir2()` is called.

The output from this function is similar to a combination of `readdir()` and `lstat()`. In some cases, certain information in the output `stat` structure differs from what `lstat()` would return. Also, the `d_extra` field in *dir* is always `NULL` for `__readdir2()`.

### Returned Value

If successful, `__readdir2()` returns a pointer to a `dirent` structure describing the next directory entry in the directory stream. When `__readdir2()` reaches the end of the directory stream, it returns a `NULL` pointer.

If unsuccessful, `__readdir2()` returns a `NULL` pointer and sets `errno` to one of the following:

<code>EBADF</code>	<i>dir</i> does not yield an open directory stream.
<code>EINVAL</code>	The buffer was too small to contain any directories.
<code>ELOOP</code>	A loop exists in symbolic links. This error occurs if the number of symbolic links in a file name in the directory is greater than <code>POSIX_SYMLLOOP</code> .
<code>ENOENT</code>	The current position of the directory stream is invalid.

### Related Information

- “`dirent.h`” on page 24
- “`stdio.h`” on page 43
- “`sys/types.h`” on page 49
- “`closedir()` — Close a Directory” on page 194
- “`opendir()` — Open a Directory” on page 877
- “`__opendir2()` — Open a Directory” on page 880
- “`readdir()` — Read an Entry from a Directory” on page 1086
- “`rewinddir()` — Reposition a Directory Stream to the Beginning” on page 1141
- “`seekdir()` — Set Position of Directory Stream” on page 1158
- “`tellldir()` — Current Location of Directory Stream” on page 1557



## readlink() — Read the Value of a Symbolic Link

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX1_SOURCE 2
#include <unistd.h>
```

```
int readlink(const char *path, char *buf, size_t bufsiz);
```

### General Description

Places the contents of the symbolic link *path* in the buffer *buf*. The size of the buffer is set by *bufsiz*. The result stored in *buf* does not include a terminating null character.

If the buffer is too small to contain the value of the symbolic link, that value is truncated to the size of the buffer (*bufsiz*). If the value returned is the size of the buffer, use *lstat()* to determine the actual size of the symbolic link.

### Returned Value

When *bufsiz* is greater than 0 and *readlink()* completes successfully, it returns the number of bytes placed in the buffer. When *bufsiz* is 0 and *readlink()* completes successfully, it returns the number of bytes contained in the symbolic link and the buffer is not changed.

If the returned value is equal to *bufsiz*, you can determine the contents of the symbolic link with either *lstat()* or *readlink()*, with a 0 value for *bufsiz*.

If *readlink()* is unsuccessful, it returns the value `-1` and sets *errno* to one of the following:

**ENOTDIR** A component of the path prefix is not a directory.

**ENAMETOOLONG**

*pathname* is longer than `PATH_MAX` characters, or some component of *pathname* is longer than `NAME_MAX` characters while `_POSIX_NO_TRUNC` is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds `PATH_MAX`. The `PATH_MAX` and `NAME_MAX` values can be determined using *pathconf()*.

**ENOENT** The named file does not exist.

**EACCES** Search permission is denied for a component of the path prefix.

**ELOOP** A loop exists in symbolic links. This error is issued if more than `POSIX_SYMLINK_MAX` symbolic links are encountered during resolution of the *path* argument.

**EINVAL** The named file is not a symbolic link.

EIO                    **Added for XPG4.2:** An I/O error occurred while reading from the file system.

### Example CBC3BR05

```
/* CBC3BR05 */
#define _POSIX1_SOURCE 2
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

main() {
    char fn[]="readlink.file";
    char sl[]="readlink.symlink";
    char buf[30];
    int fd;

    if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        close(fd);
        if (symlink(fn, sl) != 0)
            perror("symlink() error");
        else {
            if (readlink(sl, buf, sizeof(buf)) < 0)
                perror("readlink() error");
            else printf("readlink() returned '%s' for '%s'\n", buf, sl);

            unlink(sl);
        }
        unlink(fn);
    }
}
```

### Output

readlink() returned 'readlink.file' for 'readlink.symlink'

### Related Information

- “unistd.h” on page 53
- “lstat() — Get Status of File or Symbolic Link” on page 778
- “stat() — Get File Information” on page 1404
- “symlink() — Create a Symbolic Link to a Path Name” on page 1479
- “unlink() — Remove a Directory Entry” on page 1660

## readv() — Read Data on a File or Socket and Store in a Set of Buffers

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/uio.h>
```

```
ssize_t readv(int fs, const struct iovec *iov, int iovcnt);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/uio.h>
```

```
int readv(int fs, struct iovec *iov, int iovcnt);
```

### General Description

The `readv()` call reads data from a file or a socket with descriptor *fs* and stores it in a set of buffers. The data is scattered into the buffers specified by `iov[0]...iov[iovcnt-1]`.

Parameter	Description
<i>fs</i>	The file or socket descriptor.
<i>iov</i>	A pointer to an <b>iovec</b> structure.
<i>iovcnt</i>	The number of buffers pointed to by the <i>iov</i> parameter.

The **iovec** structure is defined in **uio.h** and contains the following fields:

Element	Description
<i>iov_base</i>	The pointer to the buffer.
<i>iov_len</i>	The length of the buffer.

If the descriptor refers to a socket, then it must be a connected socket.

This call returns a number of bytes of data equal to but not exceeding the sum of all the *iov\_len* fields. If less than the number of bytes requested is available, the call returns the number currently available. If data is not available for the socket *fs*, and the socket is in blocking mode, `readv()` call blocks the caller until data arrives. If data is not available and *fs* is in nonblocking mode, `readv()` returns a `-1` and sets the error code to `EWOULDBLOCK`. See “`fcntl()` — Control Open File Descriptors” on page 350 or “`ioctl()` — Control Device” on page 672 for a description of how to set nonblocking mode.

For datagram sockets, this call returns the entire datagram that was sent, provided that the datagram fits into the specified buffer. Excess datagram data is discarded. Stream sockets act like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and application A sends 1000 bytes, each call to this function can return 1 byte, or 10 bytes,

or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been received.

For X/Open sockets, if the total number of bytes to read is 0, `readv()` returns 0. If `readv()` is for a file and no data is available, `readv()` returns 0. If a `readv()` is interrupted by a signal before it reads any data, it returns -1 with `errno` set to `EINTR`. If `readv()` is interrupted by a signal after it has read data, it returns the number of bytes read. If `fs` refers to a socket, `readv()` is the equivalent of `recv()` with no flags set.

If the connection is broken on a stream socket and data is available, then `readv()` reads the data and gives no error on the first read operation. If the connection is broken on a stream socket and no data is available, then `readv()` returns 0 bytes as EOF on the first read operation.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

If successful, the number of bytes read into the buffer is returned. The value -1 indicates an error. The value of the error code indicates the specific error.

Error Code	Description
EAGAIN	The <b>O_NONBLOCK</b> flag is set for the file descriptor and the process would be delayed by the <code>readv()</code> .
EBADF	<i>fs</i> is not a valid file or socket descriptor.
ECONNRESET	A connection was forcibly closed by a peer.
EFAULT	Using <i>iov</i> and <i>iovcnt</i> would result in an attempt to access storage outside the caller's address space.
EINTR	<code>readv()</code> was interrupted by a signal that was caught before any data was available.
EINVAL	<i>iovcnt</i> was not valid, or one of the fields in the <i>iov</i> array was not valid.
ENOBUFS	Insufficient system resources are available to complete the call.
ENOTCONN	A receive is attempted on a connection-oriented socket that is not connected.
ETIMEDOUT	The connection timed out during connection establishment, or due to a transmission timeout on active connection.
EWOULDBLOCK	The socket is in nonblocking mode and data is not available to read.

## Related Information

- “connect() — Connect a Socket” on page 214
- “fcntl() — Control Open File Descriptors” on page 350
- “getsockopt() — Get the Options Associated with a Socket” on page 606
- “ioctl() — Control Device” on page 672
- “read() — Read From a File or Socket” on page 1080
- “recv() — Receive Data on a Socket” on page 1103
- “recvfrom() — Receive Messages on a Socket” on page 1106
- “recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers” on page 1110
- “select() — Monitor Activity on Files/Sockets and Message Queues” on page 1159
- “selectex() — Monitor Activity on Files/Sockets and Message Queues” on page 1166
- “send() — Send Data on a Socket” on page 1178
- “sendmsg() — Send Messages on a Socket” on page 1185
- “sendto() — Send Data on a Socket” on page 1190
- “setsockopt() — Set Options Associated with a Socket” on page 1270
- “socket() — Create a Socket” on page 1371
- “write() — Write Data on a File or Socket” on page 1780
- “writev() — Write Data on a File or Socket Socket from an Array” on page 1785

## realloc() — Change Reserved Storage Block Size

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
void *realloc(void *ptr, size_t size);
```

### General Description

Changes the size of a previously reserved storage block. The *ptr* argument points to the beginning of the block. The *size* argument gives the new size of the block in bytes. The contents of the block are unchanged up to the shorter of the new and old sizes.

If the *ptr* is NULL, `realloc()` reserves a block of storage of *size* bytes. It does not give all bits of each element an initial value of 0.

If *size* is 0 and *ptr* is not NULL, the storage pointed to by *ptr* is freed and NULL is returned.

If you use `realloc()` with a pointer that does not point to a *ptr* created previously by `malloc()`, `calloc()`, or `realloc()`, or if you pass *ptr* to storage already freed, you get undefined behavior—usually an exception.

If you ask for more storage, the contents of the extension are undefined and are not guaranteed to be 0. You can specify the bytes to which storage is initialized, which then ensures the contents of the extension.

The storage to which the returned value points is aligned for storage of any type of object. Under OS/390 C only, if 4K alignment is required, the `__4kmalc()` function should be used. (This function is only available to C applications in stand-alone Systems Programming Facility applications.) The library functions specific to the System Programming C environment are described in the *OS/390 C/C++ Programming Guide*.

### Special Behavior for C++

The C++ keywords `new` and `delete` are not interoperable with `calloc()`, `free()`, `malloc()`, or `realloc()`.

### Returned Value

Returns a pointer to the reallocated storage block. The storage location of the block might be moved by the `realloc()` function. Thus, the returned value is not necessarily the same as the *ptr* argument to `realloc()`.

The returned value is NULL if *size* is 0. If there is not enough storage to expand the block to the given size, the original block is unchanged and a NULL pointer is

returned. If `realloc()` returns a `NULL` because there is not enough storage, it will also return an error value in `errno`. The following are the possible values of `errno`:

`ENOMEM`      Insufficient memory is available

## Example

### CBC3BR06

```

/* CBC3BR06
   This example allocates storage for the prompted size of array and then
   uses realloc() to reallocate the block to hold the new size of the array.
   The contents of the array are printed after each allocation.
*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    long * array;    /* start of the array */
    long * ptr;      /* pointer to array */
    int i;           /* index variable */
    int num1, num2; /* number of entries of the array */

    void print_array( long *ptr_array, int size);

    printf( "Enter the size of the array\n" );
    scanf( "%i", &num1 );

    /* allocate num1 entries using malloc() */
    if ( (array = (long *)malloc( num1 * sizeof( long ))) != NULL ) {
        for ( ptr = array, i = 0; i < num1 ; ++i ) /* assign values */
            *ptr++ = i;
        print_array( array, num1 );
        printf("\n");
    }
    else { /* malloc error */
        printf( "Out of storage\n" );
        abort();
    }

    /* Change the size of the array ... */
    printf( "Enter the size of the new array\n" );
    scanf( "%i", &num2);

    if ( (array = (long *)realloc( array, num2* sizeof( long ))) != NULL )
    {
        for ( ptr = array + num1, i = num1; i <= num2; ++i )
            *ptr++ = i + 2000; /* assign values to new elements */
        print_array( array, num2 );
    }

    else { /* realloc error */
        printf( "Out of storage\n" );
        abort();
    }
}

void print_array( long * ptr_array, int size )
{
    int i;
    long * index = ptr_array;

    printf("The array of size %d is:\n", size);
    for ( i = 0; i < size; ++i )          /* print the array out */

```

```
    printf( "  array[ %i ] = %li\n", i, ptr_array[i] );  
}
```

### Output

If the initial value entered is 2 and the second value entered is 4, then expect the following output:

Enter the size of the array

The array of size 2 is:

array[ 0 ] = 0

array[ 1 ] = 1

Enter the size of the new array

The array of size 4 is:

array[ 0 ] = 0

array[ 1 ] = 1

array[ 2 ] = 2002

array[ 3 ] = 2003

### Related Information

- “System Programming Facilities” in the *OS/390 C/C++ Programming Guide*
- “stdlib.h” on page 45
- “calloc() — Reserve and Initialize Storage” on page 141
- “free() — Free a Block of Storage” on page 458
- “malloc() — Reserve Storage Block” on page 786
- “spc.h” on page 42



## realpath() — Resolve Path Name

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>
char *realpath(const char * file_name, char *resolved_name);
```

### General Description

The `realpath()` function derives, from the path name pointed to by *file\_name*, an absolute path name that names the same file, whose resolution does not involve `."`, `."`, or symbolic links. The generated path name is stored, up to a maximum of `{PATH_MAX}` bytes, in the buffer pointed to by *resolved\_name*.

### Returned Value

Upon successful completion, `realpath()` returns a pointer to the resolved name. Otherwise, `realpath()` returns a null pointer and sets `errno` to indicate the error, and the contents of the buffer pointed to by *resolved\_name* are undefined.

The `realpath()` function will fail if:

EACCES	Read or search permission was denied for a component of <i>file_name</i> .
EINVAL	Either the <i>file_name</i> or <i>resolved_name</i> argument is a null pointer.
EIO	An error occurred while reading from the file system.
ELOOP	Too many symbolic links were encountered in resolving <i>path</i>
ENAMETOOLONG	Pathname is longer than <code>PATH_MAX</code> characters, or some component of pathname is longer than <code>NAME_MAX</code> characters while <code>_POSIX_NO_TRUNC</code> is in effect. For symbolic links, the length of the pathname string substituted for a symbolic link exceeds <code>PATH_MAX</code> . The <code>PATH_MAX</code> and <code>NAME_MAX</code> values are determined using <code>pathconf()</code> .
ENOENT	A component of <i>file_name</i> does not name an existing file or <i>file_name</i> points to an empty string.
ENOTDIR	A component of the path prefix is not a directory.
ERANGE	File system will return <code>ERANGE</code> if the result to be stored in 'resolved_name' is larger than <code>PATH_MAX</code> .

### Related Information

- “`getcwd()` — Get Path Name of the Working Directory” on page 508
- “`sysconf()` — Determine System Configuration Options” on page 1483

## re\_comp() — Compile Regular Expression

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <re_comp.h>
```

```
char *re_comp(const char *string);
```

### General Description

The `re_comp()` function converts a regular expression string into an internal form suitable for pattern matching by `re_exec()`.

The parameter *string* is a pointer to a character string defining a source regular expression to be compiled.

If `re_comp()` is called with a null argument, the current regular expression remains unchanged.

Strings passed to `re_comp()` must be terminated by a null byte, and may include newline characters.

### Notes:

1. The `re_comp()` function is provided for historical reasons. New applications should use the new functions `fnmatch()`, `glob()`, `regcomp()` and `regexec()`, which provide full internationalized regular expression functionality compatible with ISO POSIX.2 standard.
2. The OS/390 UNIX implementation of the `re_comp()` function supports only the POSIX locale. Any other locales will yield unpredictable results.
3. The `re_comp()` and `re_exec()` functions are supported on the thread level. They must be issued from the *same* thread to work properly.

The `re_comp()` function supports simple regular expressions, which are defined below.

### Simple Regular Expressions

A Simple Regular Expression (SRE) specifies a set of character strings. The simplest form of regular expression is a string of characters with no special meaning. A small set of special characters, known as metacharacters, do have special meaning when encountered in patterns.

The following one-character regular expressions (RE) match a single character:

1. An ordinary character *c* (*not* a special character) is a one character regular expression that matches itself.

2. A backslash (\) followed by any special character (i.e.  $\backslash c$  where  $c$  is any special character) is a one character regular expression that matches the special character itself. The special characters are:
  - a. ., \*, [, and \ (period, asterisk, left square bracket, and backslash, respectively) which are always special, except when they appear within square brackets ([]).
  - b. ^ (caret or circumflex), which is special at the beginning of the entire regular expression, or when it immediately follows the left of a pair of square brackets ([]).
  - c. \$ (dollar symbol), which is special at the end of the regular expression.
  - d. The character used to bound (delimit) an entire regular expression, which is special for that regular expression.

**Note:** A backslash (\) followed by an ordinary character is a one character regular expression that matches the ordinary character itself.

3. A period (.) is a one-character RE that matches any character, except new-line.
4. A non-empty string within square brackets (*[string]*) is a one-character RE that matches any one character in that *string*. Thus, *[abc]*, if compared to other strings, would match any which contained a, b, or c.

If the caret symbol (^) is the first character of the string within square brackets (i.e. *^[string]*), the one-character RE matches any characters except newline and the remaining characters within the square brackets. Thus, *^[abc]*, if compared to other strings, would *fail* to match any which contains even one a, b, or c.

Ranges may be specified as *c-c*. The hyphen symbol, within square brackets, means "through". It may be used to indicate a range of consecutive ASCII characters. For example, *[0-9]* is equivalent to *[0123456789]*.

The *-* (hyphen) can be used by itself, but only if it is the first (after an initial ^, if any), or last character in the expression.

The right square bracket (]) can be used as part of the string but only if it is the first character within it (after an initial ^, if any). For example, the expression *[a-d]* matches either a right square bracket or one of the characters a through d.

The following rules may be used to construct REs from one character REs:

1. A one-character RE is a RE that matches whatever the one-character RE matches.
2. A one-character RE followed by an asterisk symbol (\*) is a RE that matches 0 or more occurrences of the one-character RE. For example, *(a\*e)* will match any of the following: e, ae, aaaaae. The longest leftmost match is chosen.
3. A one-character RE followed by *\{m\}*, *\{m,\}*, or *\{m,u\}* is a RE that matches a range of occurrences of the one-character RE. Non-negative integer values enclosed in *\{\}* indicate the number of times to apply the preceding one-character RE. *m* is the minimum number and *u* is the maximum number. *u* must be less than 256. If you specify only *m*, it indicates the exact number of times to apply the regular expression.

*\{m,\}* is equivalent to *\{m,u\}*. They both match *m* or more occurrences of the expression. The \* (asterisk) operation is equivalent to *\{0,\}*.

The maximum number of occurrences is matched.

4. REs can be concatenated. The concatenation of REs is a RE that matches the concatenation of the strings matched by each component of the RE.
5. A RE enclosed between the character sequences `\( and \)` is a RE that matches whatever the unadorned RE matches. The `\( and \)` sequences are ignored.
6. The expression `\n` (where  $1 \leq n \leq 9$ ) matches the same string of characters as was matched by an expression enclosed between `\( and \)` earlier in the same regular expression. The sub-expression it specified is that beginning with the *n*th occurrence of `\(` counting from the left. For example, in the expression, `\(a\)^(e)\1`, the `\1` is equivalent to *a*, giving *area*.

An entire RE may be constrained to match only an initial segment or final segment of a line (or both).

1. A caret (^) at the beginning of an entire RE constrains that RE to match an initial segment of a line.
2. A dollar symbol (\$) at the end of an entire RE constrains that RE to match a final segment of a line. For example, the construct `^entire RE$` constrains the entire RE to match the entire line.

## Returned Value

If the string pointed to by the *string* argument is successfully converted, `re_comp()` returns a null pointer. Otherwise, a pointer to an error message string (null-terminated) is returned.

The following `re_comp()` error messages are defined:

```
EDC7008E No previous regular expression
EDC7009E Regular expression too long
EDC7010E \(\) imbalance
EDC7011E \{\} imbalance
EDC7012E [] imbalance
EDC7013E Too many \(\) pairs.
EDC7014E Incorrect range values in \{\}
EDC7015E Back reference number in \digit incorrect
EDC7016E Incorrect endpoint in range expression
```

**Note:** The error message string is not to be freed by the application. It will be freed when the thread terminates.

## Related Information

- “`re_comp.h`” on page 39
- “`fnmatch()` — Match Filename or Pathname” on page 415
- “`glob()` — Generate Pathnames Matching a Pattern” on page 636
- “`regcomp()` — Compile Regular Expression” on page 1120
- “`regexexec()` — Execute Compiled Regular Expression” on page 1126
- “`re_exec()` — Match Regular Expression” on page 1115

# recv() — Receive Data on a Socket

## Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

## Format

### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

ssize_t recv(int socket, void *buffer,
              size_t length, int flags);
```

### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/socket.h>

int recv(int socket, char *buffer, int length, int flags);
```

## General Description

The `recv()` call receives data on a socket with descriptor *socket* and stores it in a buffer. The `recv()` call applies only to connected sockets.

Parameter	Description
<i>socket</i>	The socket descriptor.
<i>buf</i>	The pointer to the buffer that receives the data.
<i>len</i>	The length in bytes of the buffer pointed to by the <i>buf</i> parameter.
<i>flags</i>	<p>The <i>flags</i> parameter is set by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator ( <code> </code> ) must be used to separate them.</p> <p><b>MSG_OOB</b> Reads any out-of-band data on the socket. Out-of-band data is sent when the <b>MSG_OOB</b> flag is on for a <code>send()</code>, <code>sendto()</code>, or <code>sendmsg()</code>.</p> <p>The <code>fcntl()</code> command should be used with <code>F_SETOWN</code> to specify the recipient, either a pid or a gid, of a SIGURG signal that will be sent when out-of-band data is sent. If no recipient is set, no signal will be sent. For more information, see the <code>fcntl()</code> command. The recipient of the data determines whether to receive out-of-band data inline or not inline by the setting of the <code>SO_OOBINLINE</code> option of <code>setsockopt()</code>. If <code>SO_OOBINLINE</code> is set off and the <b>MSG_OOB</b> flag is set on, the out-of-band data byte will be read out-of-line. It is invalid for the <b>MSG_OOB</b> flag to be set on when <code>SO_OOBINLINE</code> is set on. If there is out-of-band data available, and the <b>MSG_OOB</b> flag is not set (<code>SO_OOBINLINE</code> can be on or off), then the data up to, but not including, the out-of-band data will be read. When the read</p>

cursor has reached the out-of-band data byte, then only the out-of-band data will be read on the next read. The `SIOCATMARK` option of `ioctl()` can be used to determine if the read cursor is currently at the out-of-band data byte. For more information, refer to the `setsockopt()` and `ioctl()` commands.

**MSG\_PEEK**      Peeks at the data present on the socket; the data is returned but not consumed, so that a subsequent receive operation sees the same data.

This call returns the length of the incoming message or data. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard excess bytes. If data is not available for the socket *socket*, and *socket* is in blocking mode, the `recv()` call blocks the caller until data arrives. If data is not available and *socket* is in nonblocking mode, `recv()` returns a `-1` and sets the error code to `EWOULDBLOCK`. See “`fcntl()` — Control Open File Descriptors” on page 350 or “`ioctl()` — Control Device” on page 672 for a description of how to set nonblocking mode.

For datagram sockets, this call returns the entire datagram that was sent, provided that the datagram fits into the specified buffer. Stream sockets act like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and application A sends 1000 bytes, each call to this function can return 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been received.

For sockets that are defined as `AF_INET` and `SOCK_DGRAM` type sockets, bulk mode I/O will be supported only after the socket has been connected and the `setibmssockopt()` or `sock_do_bulkmode()` function is issued to set a socket for bulk mode use.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

If successful, the length of the message or datagram in bytes is returned. The value `-1` indicates an error. The value `0` indicates the connection is closed. The value of the error code indicates the specific error.

#### Error Code      Description

<code>EBADF</code>	<i>socket</i> is not a valid socket descriptor.
<code>ECONNRESET</code>	A connection was forcibly closed by a peer.
<code>EFAULT</code>	Using the <i>buf</i> and <i>len</i> parameters would result in an attempt to access storage outside the caller's address space.
<code>EINTR</code>	The <code>recv()</code> call was interrupted by a signal that was caught before any data was available.
<code>EINVAL</code>	The request is invalid or not supported. The <code>MSG_OOB</code> flag is set and no out-of-band data is available.

EIO	There has been a network or transport failure.
ENOBUFS	Insufficient system resources are available to complete the call.
ENOTCONN	A receive is attempted on a connection-oriented socket that is not connected.
ENOTSOCK	The descriptor is for a file, not for a socket.
EOPNOTSUPP	The specified flags are not supported for this socket type or protocol.
ETIMEDOUT	The connection timed out during connection establishment, or due to a transmission timeout on active connection.
EWouldBlock	<i>socket</i> is in nonblocking mode and data is not available to read.

## Related Information

- “connect() — Connect a Socket” on page 214
- “fcntl() — Control Open File Descriptors” on page 350
- “getsockopt() — Get the Options Associated with a Socket” on page 606
- “ioctl() — Control Device” on page 672
- “read() — Read From a File or Socket” on page 1080
- “readv() — Read Data on a File or Socket and Store in a Set of Buffers” on page 1093
- “recvfrom() — Receive Messages on a Socket” on page 1106
- “recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers” on page 1110
- “select() — Monitor Activity on Files/Sockets and Message Queues” on page 1159
- “selectex() — Monitor Activity on Files/Sockets and Message Queues” on page 1166
- “send() — Send Data on a Socket” on page 1178
- “sendmsg() — Send Messages on a Socket” on page 1185
- “sendto() — Send Data on a Socket” on page 1190
- “setsockopt() — Set Options Associated with a Socket” on page 1270
- “socket() — Create a Socket” on page 1371
- “write() — Write Data on a File or Socket” on page 1780
- “writev() — Write Data on a File or Socket Socket from an Array” on page 1785

## recvfrom() — Receive Messages on a Socket

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int recvfrom(int socket, void *buffer, size_t length, int flags,
             struct sockaddr *address, size_t *address_length);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/socket.h>

int recvfrom(int socket, char *buffer, int length, int flags,
             struct sockaddr *address, int *address_length);
```

### General Description

The `recvfrom()` call receives data on a socket named by descriptor *socket* and stores it in a buffer. The `recvfrom()` call applies to any datagram socket, whether connected or unconnected.

#### Parameter Description

Parameter	Description
<i>socket</i>	The socket descriptor.
<i>buffer</i>	The pointer to the buffer that receives the data.
<i>length</i>	The length in bytes of the buffer pointed to by the <i>buffer</i> parameter.
<i>flags</i>	A parameter that can be set to 0 or MSG_PEEK, or MSG_OOB.

**MSG\_OOB** Reads any out-of-band data on the socket. Out-of-band data is sent when the MSG\_OOB flag is on for a `send()`, `sendto()`, or `sendmsg()`.

The `fcntl()` command should be used with `F_SETOWN` to specify the recipient, either a pid or a gid, of a SIGURG signal that will be sent when out-of-band data is sent. If no recipient is set, no signal will be sent. For more information, see the `fcntl()` command. The recipient of the data determines whether to receive out-of-band data inline or not inline by the setting of the `SO_OOBINLINE` option of `setsockopt()`. If `SO_OOBINLINE` is set off and the MSG\_OOB flag is set on, the out-of-band data byte will be read out-of-line. It is invalid for the MSG\_OOB flag to be set on when `SO_OOBINLINE` is set on. If there is out-of-band data available, and the MSG\_OOB flag is not set (`SO_OOBINLINE` can be on or off), then the data up to, but not including,



the out-of-band data will be read. When the read cursor has reached the out-of-band data byte, then only the out-of-band data will be read on the next read. The SIOCATMARK option of `ioctl()` can be used to determine if the read cursor is currently at the out-of-band data byte. For more information, refer to the `setsockopt()` and `ioctl()` commands.

**MSG\_PEEK**      Peeks at the data present on the socket; the data is returned but not consumed, so that a subsequent receive operation sees the same data.

**MSG\_WAITALL**      Requests that the function block until the full amount of data requested can be returned. The function may return a smaller amount of data if a signal is caught, the connection is terminated, or an error is pending for the socket. The function may also return earlier if out-of-band (OOB) data is inline and there is OOB data to be read. In this case, the data up to the OOB data is returned on the first *recvfrom*. The OOB data is returned on the subsequent read request. This option is valid only for the AF\_UNIX domain.

*address*            A pointer to a socket address structure from which data is received. If *address* is a nonzero value, the source address is returned.

*address\_length*      The size of *name* in bytes.

If *address* is nonzero, the source address of the message is filled. *address\_length* must first be initialized to the size of the buffer associated with *name*, and is then modified on return to indicate the actual size of the address stored there.

If either *address* or *address\_length* is a null pointer, then *address* and *address\_length* are unchanged.

If *name* is nonzero, the source address of the message is filled. *namelength* must first be initialized to the size of the buffer associated with *name*, and is then modified on return to indicate the actual size of the address stored there.

This call returns the length of the incoming message or data. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard excess bytes. If data is not available for the socket *socket*, and *socket* is in blocking mode, the `recvfrom()` call blocks the caller until data arrives. If data is not available and *socket* is in nonblocking mode, `recvfrom()` returns a -1 and sets the error code to EWOULDBLOCK. See “`fcntl()` — Control Open File Descriptors” on page 350 or “`ioctl()` — Control Device” on page 672 for a description of how to set nonblocking mode.

For datagram sockets, this call returns the entire datagram that was sent, provided that the datagram fits into the specified buffer. Stream sockets act like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and application A sends 1000 bytes, each call to this function can return 1 byte, or 10 bytes, or the entire 1000 bytes. There-

fore, applications using stream sockets should place this call in a loop, calling this function until all data has been received.

For sockets that are defined as AF\_INET and SOCK\_DGRAM type sockets, bulk mode I/O will be supported when the setibmssockopt() or sock\_do\_bulkmode() function is issued to set a socket for bulk mode use.

### Special Behavior for C++

To use this function with C++, you must use the \_XOPEN\_SOURCE\_EXTENDED 1 feature test macro.

### Returned Value

If successful, the length of the message or datagram in bytes is returned. The value -1 indicates an error. The value of the error code indicates the specific error.

Error Code	Description
EBADF	<i>socket</i> is not a valid socket descriptor.
ECONNRESET	The connection was forcibly closed by a peer.
EFAULT	Using the <i>buffer</i> and <i>length</i> parameters would result in an attempt to access storage outside the caller's address space.
EINTR	A signal interrupted recvfrom() before any data was available.
EINVAL	The request is invalid or not supported. The MSG_OOB flag is set and no out-of-band data is available.
EIO	There has been a network or transport failure.
ENOBUFS	Insufficient system resources are available to complete the call.
ENOTCONN	A receive is attempted on a connection-oriented socket that is not connected.
ENOTSOCK	The descriptor is for a file, not for a socket.
EOPNOTSUPP	The specified flags are not supported for this socket type.
ETIMEDOUT	The connection timed out during connection establishment, or due to a transmission timeout on active connection.
EWouldBlock	<i>socket</i> is in nonblocking mode and data is not available to read.

### Related Information

- “fcntl() — Control Open File Descriptors” on page 350
- “getsockopt() — Get the Options Associated with a Socket” on page 606
- “ioctl() — Control Device” on page 672
- “read() — Read From a File or Socket” on page 1080
- “readv() — Read Data on a File or Socket and Store in a Set of Buffers” on page 1093
- “recv() — Receive Data on a Socket” on page 1103
- “recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers” on page 1110
- “select() — Monitor Activity on Files/Sockets and Message Queues” on page 1159

- “selectex() — Monitor Activity on Files/Sockets and Message Queues” on page 1166
- “send() — Send Data on a Socket” on page 1178
- “sendmsg() — Send Messages on a Socket” on page 1185
- “sendto() — Send Data on a Socket” on page 1190
- “setsockopt() — Set Options Associated with a Socket” on page 1270
- “socket() — Create a Socket” on page 1371
- “write() — Write Data on a File or Socket” on page 1780
- “writev() — Write Data on a File or Socket Socket from an Array” on page 1785

## recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>
```

```
ssize_t recvmsg(int socket, struct msghdr *message, int flags);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/socket.h>
```

```
int recvmsg(int socket, struct msghdr *message, int flags);
```

### General Description

The `recvmsg()` call receives messages on a socket with descriptor *socket* and stores them in an array of message headers.

#### Parameter Description

<i>socket</i>	The socket descriptor.
<i>msg</i>	An array of message headers into which messages are received.
<i>flags</i>	<p>The <i>flags</i> parameter is set by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator ( <code> </code> ) must be used to separate them.</p> <p><b>MSG_OOB</b> Reads any out-of-band data on the socket. Out-of-band data is sent when the MSG_OOB flag is on for a <code>send()</code>, <code>sendto()</code> or <code>sendmsg()</code>.</p> <p>The <code>fcntl</code> command should be used with <code>F_SETOWN</code> to specify the recipient, either a pid or a gid, of a SIGURG signal that will be sent when out-of-band data is sent. If no recipient is set, no signal will be sent. For more information, see the <code>fcntl()</code> command. The recipient of the data determines whether to receive out-of-band data inline or not inline by the setting of the SO_OOBINLINE option of <code>setsockopt()</code>. If SO_OOBINLINE is set off and the MSG_OOB flag is set on, the out-of-band data byte will be read out-of-line. It is invalid for the MSG_OOB flag to be set on when SO_OOBINLINE is set on. If there is out-of-band data available, and the MSG_OOB flag is not set (SO_OOBINLINE can be on or off), then the data up to, but not including, the out-of-band data will be read. When the read cursor has reached the out-of-band data byte, then</p>

only the out-of-band data will be read on the next read, and the output MSG\_OOB msg\_flag in the message header will be set on. The SIOCATMARK option of ioctl() can be used to determine if the read cursor is currently at the out-of-band data byte. For more information, refer to the setsockopt() and ioctl() commands.

**MSG\_PEEK**      Peeks at the data present on the socket; the data is returned but not consumed, so that a subsequent receive operation will see the same data.

**MSG\_WAITALL**      Requests that the function block until the full amount of data requested can be returned. The function may return a smaller amount of data if a signal is caught, the connection is terminated, or a error is pending for the socket. This option is valid only for the AF\_UNIX domain.

A message header is defined by a **msg\_hdr** structure. A definition of this structure can be found in the **sys/socket.h** include file and contains the following elements:

Element	Description
<i>msg_iov</i>	An array of <i>iovec</i> buffers into which the message is placed.
<i>msg_iovlen</i>	The number of elements in the <i>msg_iov</i> array.
<i>msg_name</i>	An optional pointer to a buffer where the sender's address is stored.
<i>msg_namelen</i>	The size of the address buffer.
<i>caddr_t msg_accrighs</i>	Access rights sent/received (ignored if specified by the user).
<i>int msg_accrighslen</i>	Length of access rights data (ignored if specified by the user).
<i>msg_control</i>	Ancillary data, see below.
<i>msg_controllen</i>	Ancillary data buffer length.
<i>msg_flags</i>	Flags on received message.

Ancillary data consists of a sequence of pairs, each consisting of a **cmsghdr** structure followed by a data array. The data array contains the ancillary data message, and the **cmsghdr** structure contains descriptive information that allows an application to correctly parse the data.

The **sys/socket.h** header file defines the **cmsghdr** structure that includes at least the following members:

Element	Description
<i>cmsg_len</i>	Data byte count, including header.
<i>cmsg_level</i>	Originating protocol.
<i>cmsg_type</i>	Protocol-specific type.

The **sys/socket.h** header file defines the following macro for use as the **cmsg\_type** value when **cmsg\_level** is SOL\_SOCKET:

SCM\_RIGHTS

Indicates that the data array contains the access rights to be sent or received. This option is valid only for the AF\_UNIX domain.

The **sys/socket.h** header file defines the following macros to gain access to the data arrays in the ancillary data associated with a message header:

CMSG\_DATA(*cmsg*)

If the argument is a pointer to a **cmsg\_hdr** structure, this macro returns an unsigned character pointer to the data array associated with the **cmsg\_hdr** structure.

CMSG\_NXTHDR(*mhdr, cmsg*)

If the first argument is a pointer to a **msghdr** structure and the second argument is a pointer to a **cmsg\_hdr** structure in the ancillary data, pointed to by the **msg\_control** field of that **msghdr** structure, this macro returns a pointer to the next **cmsg\_hdr** structure, or a null pointer if this structure is the last **cmsg\_hdr** in the ancillary data.

CMSG\_FIRSTHDR(*mhdr*)

If the argument is a pointer to a **msghdr** structure, this macro returns a pointer to the first **cmsg\_hdr** structure in the ancillary data associated with this **msghdr** structure, or a null pointer if there is no ancillary data associated with the **msghdr** structure.

The `recvmsg()` call applies to sockets, regardless of whether they are in the connected state.

This call returns the length of the data received. If data is not available for the socket *socket*, and *socket* is in blocking mode, the `recvmsg()` call blocks the caller until data arrives. If data is not available and *socket* is in nonblocking mode, `recvmsg()` returns a -1 and sets the error code to EWOULDBLOCK. See “fcntl() — Control Open File Descriptors” on page 350 or “ioctl() — Control Device” on page 672 for a description of how to set nonblocking mode.

For datagram sockets, this call returns the entire datagram that was sent, provided that the datagram fits into the specified buffer. Stream sockets act like streams of information with no boundaries separating data. For example, if applications A and B are connected with a stream socket and application A sends 1000 bytes, each call to this function can return 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been received.

On successful completion, the **msg\_flags** member for the message header is the bitwise-inclusive OR of all of the following flags that indicate conditions detected for the received message:

MSG\_OOB Out-of-band data was received.

MSG\_TRUNC Normal data was truncated.

MSG\_CTRUNC

Control data was truncated.

For sockets that are defined as AF\_INET and SOCK\_DGRAM type sockets, bulk mode I/O will be supported when the `setibmssockopt()` or `sock_do_bulkmode()` function is issued to set a socket for bulk mode use.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

If successful, the length of the message in bytes is returned. The value `-1` indicates an error. The value of the error code indicates the specific error.

#### Error Code      Description

EBADF	<i>socket</i> is not a valid socket descriptor.
ECONNRESET	The connection was forcibly closed by a peer.
EFAULT	Using <i>msg</i> would result in an attempt to access storage outside the caller's address space.
EINTR	The function was interrupted by a signal before any data was available.
EINVAL	The request is invalid or not supported. The sum of the <b>iov_len</b> values overflows a <b>ssize_t</b> .
EIO	There has been a network or transport failure.
ENOBUFS	Insufficient system resources are available to complete the call.
ENOTCONN	A receive is attempted on a connection-oriented socket that is not connected.
EOPNOTSUPP	The specified flags are not supported for this socket type.
ENOTSOCK	The descriptor is for a file, not for a socket.
ETIMEDOUT	The connection timed out during connection establishment, or due to a transmission timeout on active connection.
EWouldBlock	<i>socket</i> is in nonblocking mode and data is not available to read.

### Related Information

- “connect() — Connect a Socket” on page 214
- “fcntl() — Control Open File Descriptors” on page 350
- “getsockopt() — Get the Options Associated with a Socket” on page 606
- “ioctl() — Control Device” on page 672
- “read() — Read From a File or Socket” on page 1080
- “readv() — Read Data on a File or Socket and Store in a Set of Buffers” on page 1093
- “recv() — Receive Data on a Socket” on page 1103
- “recvfrom() — Receive Messages on a Socket” on page 1106
- “select() — Monitor Activity on Files/Sockets and Message Queues” on page 1159
- “selectex() — Monitor Activity on Files/Sockets and Message Queues” on page 1166

- “send() — Send Data on a Socket” on page 1178
- “sendmsg() — Send Messages on a Socket” on page 1185
- “sendto() — Send Data on a Socket” on page 1190
- “setsockopt() — Set Options Associated with a Socket” on page 1270
- “socket() — Create a Socket” on page 1371
- “write() — Write Data on a File or Socket” on page 1780
- “writev() — Write Data on a File or Socket Socket from an Array” on page 1785



## re\_exec() — Match Regular Expression

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <re_comp.h>
```

```
int re_exec(const char *string);
```

### General Description

The `re_exec()` function attempts to match the string pointed to by the *string* argument with the last regular expression passed to `re_comp()`.

The parameter *string* is a pointer to a character string to be compared.

Strings passed to `re_exec()` must be terminated by a null byte, and may include newline characters.

### Notes:

1. The OS/390 UNIX implementation of the `re_exec()` function supports only the POSIX locale. Any other locales will yield unpredictable results.
2. The `re_comp()` and `re_exec()` functions are supported on the thread level. They must be issued from the *same* thread to work properly.
3. The `re_exec()` function is provided for historical reasons. New applications should use the new functions `fnmatch()`, `glob()`, `regcomp()` and `regexexec()`, which provide full internationalized regular expression functionality compatible with ISO POSIX.2 standard.

The `re_exec()` function supports simple regular expressions, which are defined in “`re_comp()` — Compile Regular Expression” on page 1100.

### Returned Value

Upon successful completion, `re_exec()` returns 1 if the input *string* matches the last compiled regular expression. Otherwise, `re_exec()` returns 0 if the input *string* fails to match the last compiled regular expression, and -1 if the compiled regular expression is invalid (indicating an internal error).

### Related Information

- “`re_comp.h`” on page 39
- “`fnmatch()` — Match Filename or Pathname” on page 415
- “`glob()` — Generate Pathnames Matching a Pattern” on page 636
- “`regcomp()` — Compile Regular Expression” on page 1120
- “`regexexec()` — Execute Compiled Regular Expression” on page 1126
- “`re_comp()` — Compile Regular Expression” on page 1100

## regcmp() — Compile Regular Expression

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	C only	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <libgen.h>

char *regcmp(const char *pattern[,...],(char *)0);

char *regex(const char *cmppat,const char
*subject[,subexp,...]);
extern char *__loc1;
```

### General Description

The `regcmp()` function concatenates regular expression (RE) patterns specified by a list of one or more *pattern* arguments. The end of this list must be delimited by a null pointer. The `regcmp()` function then converts the concatenated RE pattern into an internal form suitable for use by the pattern matching `regex()` function. If conversion is successful, the `regcmp()` function returns a pointer to the converted pattern. Otherwise, it returns a null pointer. The `regcmp()` function uses `malloc()` to obtain storage for the converted pattern. It is the application's responsibility to free unneeded space so allocated.

The `regex()` function executes a converted pattern *cmppat* against a *subject* string. If *cmppat* matches all or part of the *subject* string, the `regex()` function returns a pointer to the next unmatched character in the *subject* string and sets the external variable `__loc1` to point the first matched character in the *subject* string. If no match is found between *cmppat* and the *subject* string, the `regex()` function returns a null pointer.

The `regcmp()` and `regex()` functions are supported in any locale. However, results are unpredictable if they are not run in the same locale.

Following are valid RE symbols and their meaning to the `regcmp()` and `regex()` functions:

<u>Expression</u>	<u>Meaning</u>
NUL	Terminate RE pattern and text string
c	Any non-special character, c, is a one-character RE which matches itself.
\s	A backslash (\) followed by a special character, s, is a one-character RE which matches the special character itself.

The following characters are special:

period, ., asterisk, \*, plus, +, dollar, \$, left square bracket, [, left brace, {, right brace, }, left parenthesis, (, right parenthesis, ), and backslash, \, are always special except when they appear within square brackets ([ ]).

caret (^) is special at the beginning of an entire RE (which is another name for a pattern).

**Note:** An non-special character preceded by \ is a one-character RE which matches the non-special character.

yz	Concatenation of REs y and z matches concatenation of strings matched by y and z.
.	The period (.) special character RE matches any single character <u>except</u> the <newline> character.
^	The caret (^) at the beginning of an entire RE is an RE which matches the beginning of a string. Thus, it <b>anchors</b> or limits matches by the entire RE to the beginning of strings.
\$	The dollar (\$) at the end of an entire RE is an RE which only the end of a string (delimited by the <NUL> character). Thus, it <b>anchors</b> or limits matches by the entire RE to the end of strings.  <b>Note:</b> \n (the C language designation for a <newline> character) must be used in an entire RE to match any embedded or trailing <newline> character in a text string.
(...)	Parentheses are used to delimit a subexpression which matches whatever the REs comprising the subexpression would have matched without the delimiting parentheses.
(...) \$n	\$n, where n is a digit between 0 and 9, inclusive, may be used to tag a subexpression. The tag tells the regex() function to return the substring matched by the subexpression at address specified by (n+1)th argument after <i>subject</i> .
*	A one-character RE <u>or</u> subexpression followed by an asterisk (*) is a RE that matches zero or more occurrences of the one-character RE or subexpression. If there is any choice, the longest leftmost string that permits a match is chosen.
+	A one-character RE <u>or</u> subexpression followed by a plus (+) is a RE that matches one or more occurrences of the one-character RE or subexpression. Whenever a choice exists, the RE matches as many occurrences as possible.
{m,n}	A one-character RE <u>or</u> subexpression followed by integer values, m and n, enclosed in braces is a RE which matches repeated occurrences of whatever the preceding one-character RE or subexpression matched. The value of m, which must be in the range 0 to 255, inclusive, is the minimum number of occurrences required for a match. The value of n which, if specified, must also must be in the range 0 to 255, inclusive, is the maximum. The value of n, if specified, must be greater than or equal to the value m. The following brace expressions are valid:
{m}	Matches exactly m occurrences of the preceding one-character RE or subexpression.
{m,}	Matches m or more occurrences of the preceding one-character RE or subexpression. There is no limit on the number of occurrences which will be

matched. The plus (+) and asterisk (\*) operations are equivalent to {1,} and {0,}, respectively.

{m,n} Matches between m and n occurrences, inclusive.

Whenever a choice exists, the RE matches as many occurrence as possible.

[...] A non-empty list of characters enclosed by square brackets is a one-character RE that matches any one character in the list.

[^...] A non-empty list of characters preceded by a caret (^) enclosed by square brackets is a one-character RE that matches any character except <newline> and the characters in the list. The ^ has special meaning only if it is the first character after the left bracket ([).

[c1-c2] The hyphen (-) between two characters c1 and c2 within square brackets designates the list of characters whose collating values fall between the collating values of c1 and c2 in the current locale. The collating value of c2 must be greater than or equal to c1. Also, c2 may not be used as the ending point of one range and the starting point of another range. In other words, c1-c2-c3 is invalid.

The - loses special meaning if it occurs first or last in the bracket expression or if it is used for c1 or c2.

The right bracket, ], does not terminate a bracket expression when it is the first character within it (after an initial ^, if any). For example, the expression [[0-9] matches a right bracket or a digit in the range 0-9, inclusive.

#### Notes:

1. Multiple duplication symbols applied to the same RE will be interpreted in the following order of precedence:

- a. \*
- b. +
- c. {}

2. RE Order of precedence is as follows, from high to low:

- a. escaped character \character
- b. bracket expression [...]
- c. subexpression (...)
- d. duplication \* + {}
- e. concatenation yz
- f. anchors ^ \$

**Note:** The regcmp() function is provided for historical reasons. New applications should use the new functions fnmatch(), glob(), regcomp() and regexexec(), which provide full internationalized regular expression functionality compatible with ISO POSIX.2 standard.

## Returned Value

If the pattern formed by concatenating the list of *pattern* arguments is successfully converted, regcmp() returns a pointer to the converted pattern. Otherwise, it returns a null pointer. If regcmp() is unable to allocate storage for the converted pattern, it sets errno to ENOMEM.

If the regex() function successfully matches the converted pattern *cmppat* to all or part of the *subject* string, it returns a pointer to the next unmatched character in *subject*. Otherwise, it returns a null pointer.

## Related Information

- “libgen.h” on page 31
- “fnmatch() — Match Filename or Pathname” on page 415
- “glob() — Generate Pathnames Matching a Pattern” on page 636
- “regcomp() — Compile Regular Expression” on page 1120
- “regexexec() — Execute Compiled Regular Expression” on page 1126
- “re\_comp() — Compile Regular Expression” on page 1100
- “re\_exec() — Match Regular Expression” on page 1115

## regcomp() — Compile Regular Expression

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 OS/390 UNIX	both	

### Format

```
#include <regex.h>
```

```
int regcomp(regex_t *preg, const char *pattern, int cflags);
```

### General Description

Compiles the regular expression specified by *preg* into an executable string of op-codes.

*preg* is a pointer to a compiled regular expression.

*pattern* is a pointer to a character string defining a source regular expression (described below).

*cflags* is a bit flag defining configurable attributes of compilation process:

REG\_EXTENDED

Support extended regular expressions.

REG\_NEWLINE

Eliminate any special significance to the new-line character.

REG\_ICASE Ignore case in match.

REG\_NOSUB Report only success or fail in regexec(), that is, verify the syntax of a regular expression. If this flag is set, the regcomp() function sets *re\_nsub* to the number of parenthesized subexpressions found in *pattern*. Otherwise, a subexpression results in an error.

The regcomp() function under OS/390 C/C++ will use the definition of characters according to the current LC\_SYNTAX category. The characters, [, ], {, }, |, ^, and \$, have varying code points in different encoded character sets.

To provide an ASCII input/output format for applications using this function, define feature test macro \_\_LIBASCII as described on page 22.

### Regular Expressions

The functions regcomp(), regerror(), regexec(), and regfree() use regular expressions in a similar way to the UNIX awk, ed, grep, and egrep commands.

The simplest form of regular expression is a string of characters with no special meaning. The following characters do have special meaning; they are used to form extended regular expressions:

Symbol	Description
.	The period symbol matches any one character except the terminal new-line character.
<code>[character–character]</code>	The hyphen symbol, within square brackets, means “through”. It fills in the intervening characters according to the current collating sequence. For example, <code>[a–z]</code> can be equivalent to <code>[abc...xyz]</code> or, with a different collating sequence, it can be equivalent to <code>[aAbBcC...xXyYzZ]</code> .
<code>[string]</code>	A string within square brackets specifies any of the characters in <i>string</i> . Thus <code>[abc]</code> , if compared to other strings, would match any that contained a, b, or c.  No assumptions are made at compile time about the actual characters contained in the range.
<code>[m]</code> <code>[m,]</code> <code>[m,u]</code>	Integer values enclosed in <code>[]</code> indicate the number of times to apply the preceding regular expression. <i>m</i> is the minimum number, and <i>u</i> is the maximum number. <i>u</i> must be less than 256. If you specify only <i>m</i> , it indicates the exact number of times to apply the regular expression.  <code>[m,]</code> is equivalent to <code>[m,u]</code> . They both match <i>m</i> or more occurrences of the expression. The <code>+</code> (plus) and <code>*</code> (asterisk) operations are equivalent to <code>[1,]</code> and <code>[0,]</code> respectively.
*	The asterisk symbol indicates 0 or more of any characters. For example, <code>[a*e]</code> is equivalent to any of the following: <code>99ae9</code> , <code>aaaaae</code> , <code>a999e99</code> .
\$	The dollar symbol matches the end of the string. (Use <code>\n</code> to match a new-line character.)
<code>character+</code>	The plus symbol specifies one or more occurrences of a character. Thus, <code>smith+ern</code> is equivalent to, for example, <code>smithhhern</code> .
<code>[^string]</code>	The caret symbol, when inside square brackets, negates the characters within the square brackets. Thus <code>[^abc]</code> , if compared to other strings, would <i>fail</i> to match any that contains even one a, b, or c.
<code>(expression)\$n</code>	Stores the value matched by the enclosed regular expression in the $(n+1)^{\text{th}}$ <i>ret</i> parameter. Ten enclosed regular expressions are allowed. Assignments are made unconditionally.
<code>(expression)</code>	Groups a subexpression allowing an operator, such as <code>*</code> , <code>+</code> , or <code>[]</code> , to work on the subexpression enclosed in parentheses. For example, <code>(a*(cb+)*)\$0</code> .

**Notes:**

- Do *not* use multibyte characters.
- You can use the `]` (right square bracket) alone within a pair of square brackets, but only if it immediately follows either the opening left square bracket or if it immediately follows `[^`. For example: `[]–]` matches the `]` and `–` characters.
- All the preceding symbols are *special*. You precede them with `\` to use the symbol itself. For example, `a\.e` is equivalent to `a.e`.

- You can use the `-` (hyphen) by itself, but only if it is the first or last character in the expression. For example, the expression `[]--0]` matches either the `]` or else the characters `-` through `0`. Otherwise, use `\-`.

## Returned Value

If `regcomp()` is successful, the function returns 0; otherwise, the function returns nonzero, and the content of *preg* is undefined.

## Example CBC3BR07

```
/* CBC3BR07
   This example compiles an extended regular expression.
*/
#include <regex.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

main() {
    regex_t    preg;
    char       *string = "a simple string";
    char       *pattern = ".*(simple).*";
    int        rc;

    if ((rc = regcomp(&preg, string, REG_EXTENDED)) != 0) {
        printf("regcomp() failed, returning nonzero (%d)", rc);
        exit(1);
    }
}
```

## Related Information

- “`regex.h`” on page 40
- “`regerror()` — Return Error Message” on page 1123
- “`regexexec()` — Execute Compiled Regular Expression” on page 1126
- “`regfree()` — Free Memory for Regular Expression” on page 1129



## regerror() — Return Error Message

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#include <regex.h>
```

```
size_t regerror(int errcode, const regex_t *preg, char *errbuf, size_t
errbuf_size);
```

### General Description

Finds the description for *errcode*. (For a description of regular expressions, see “Regular Expressions” on page 1120.)

### Returned Value

Returns the integer value that is the size of the buffer needed to hold the generated description string for the error condition corresponding to *errcode*. The function returns the following messages.

errcode	Description String
REG_NOMATCH	RE pattern not found
REG_BADPAT	Invalid regular expression
REG_ECOLLATE	Invalid collating element
REG_ECTYPE	Invalid character class
REG_EESCAPE	Last character is \
REG_ESUBREG	Invalid number in \digit
REG_EBRACK	[] imbalance
REG_EPAREN	\( \) or () imbalance
REG_EBRACE	\{ \} or { } imbalance
REG_BADBR	Invalid \{ \} range exp
REG_ERANGE	Invalid range exp endpoint
REG_ESPACE	Out of memory
REG_BADRPT	?*+ not preceded by valid RE
REG_ECHAR	Invalid multibyte character
REG_EBOL	¬ anchor and not BOL
REG_EEOL	\$ anchor and not EOL

The LC\_SYNTAX characters in the messages will be converted to the code points from the current LC\_SYNTAX category.

## Example

### CBC3BR08

```

/* CBC3BR08
   This example compiles an invalid regular expression, and print error
   message regerror().
*/
#include <regex.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

main() {
    regex_t    preg;
    char       *pattern = "a[missing.bracket";
    int        rc;
    char       buffer[100];

    if ((rc = regcomp(&preg, pattern, REG_EXTENDED)) != 0) {
        regerror(rc, &preg, buffer, 100);
        printf("regcomp() failed with '%s'\n", buffer);
        exit(1);
    }
}

```

## Related Information

- Chapters about internationalization in the *OS/390 C/C++ Programming Guide*
- “regex.h” on page 40
- “regcomp() — Compile Regular Expression” on page 1120
- “regexexec() — Execute Compiled Regular Expression” on page 1126
- “regfree() — Free Memory for Regular Expression” on page 1129

## regex() — Execute Compiled Regular Expression

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	C only	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <libgen.h>

char *regex(const char *cmppat, const char
*subject[, subexp, ...]);
extern char *__loc1;
```

### General Description

The `regex()` function executes a converted pattern *cmppat* produced by the `regcomp()` function against a *subject* string. If *cmppat* matches all or part of the *subject* string, the `regex()` function returns a pointer to the next unmatched character in the *subject* string and sets the external variable `__loc1` to point the first matched character in the *subject* string. If no match is found between *cmppat* and the *subject* string, the `regex()` function returns a null pointer.

The `regex()` and `regcomp()` functions are supported in any locale. However, results are unpredictable if they are not run in the same locale.

Refer to “`regcomp()` — Compile Regular Expression” on page 1116 for a description of regular expression syntax and semantics supported by the `regex()` and `regcomp()` functions.

**Note:** The `regex()` function is provided for historical reasons. New applications should use the new functions `fnmatch()`, `glob()`, `regcomp()` and `regexexec()`, which provide full internationalized regular expression functionality compatible with ISO POSIX.2 standard.

### Returned Value

If the `regex()` function successfully matches the converted pattern *cmppat* to all or part of the *subject* string, it returns a pointer to the next unmatched character in *subject*. Otherwise, it returns a null pointer.

### Related Information

- “`libgen.h`” on page 31
- “`fnmatch()` — Match Filename or Pathname” on page 415
- “`glob()` — Generate Pathnames Matching a Pattern” on page 636
- “`regcomp()` — Compile Regular Expression” on page 1120
- “`regexexec()` — Execute Compiled Regular Expression” on page 1126
- “`re_comp()` — Compile Regular Expression” on page 1100
- “`re_exec()` — Match Regular Expression” on page 1115

## regexec() — Execute Compiled Regular Expression

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#include <regex.h>
```

```
int regexec(const regex_t *preg, const char *string,
            size_t nmatch, regmatch_t *pmatch, int eflags);
```

### General Description

Compares the null-terminated string *STRING* against the compiled regular expression, *preg*. (For a description of regular expressions, see “Regular Expressions” on page 1120.)

*preg* is a pointer to a compiled regular expression to compare against *STRING*.

*string* is a pointer to a string to be matched.

*nmatch* is the number of subexpressions to match.

*pmatch* is an array of offsets into *STRING* which matched the corresponding subexpressions in *preg*.

*eflags* is a bit flag defining customizable behavior of *regexec()*.

**REG\_NOTBOL** Indicates that the first character of *STRING* is not the beginning of the line.

**REG\_NOTEOL** Indicates that the first character of *STRING* is not the end of the line.

If *nmatch* parameter is 0 or **REG\_NOSUB** was set on the call to *regcomp()*, *regexec()* ignores the *pmatch* argument. Otherwise, the *pmatch* argument points to an array of at least *nmatch* elements. The *regexec()* function fills in the elements of the array with offsets of the substrings of *STRING* that correspond to the parenthesized subexpressions of the original *pattern* specified to *regcomp()*. The 0th element of the array corresponds to the entire pattern. If there are more than *nmatch* subexpressions, only the first *nmatch*–1 are recorded.

When matching a basic or extended regular expression, any given parenthesized subexpression of *pattern* might participate in the match of several different substrings of *STRING*. The following rules determine which substrings are reported in *pmatch*.

1. If a subexpression participated in a match several times, the offset of the last matching substring is reported in *pmatch*.
2. If a subexpression did not match in the source *STRING*, the offset shown in *pmatch* is set to –1.

3. If a subexpression contains subexpressions, the data in *pmatch* refers to the last such subexpression.
4. If a subexpression matches a zero-length string, the offsets in *pmatch* refer to the byte immediately following the matching string.

If `EREG_NOSUB` was set when `regcomp()` was called, the contents of *pmatch* are unspecified.

If `REG_NEWLINE` was set when `regcomp()` was called, new-line characters are allowed in `STRING`.

#### Notes:

- With OS/390 C/C++, the string passed to the `regexec()` function is assumed to be in the initial shift state, unless `REG_NOTBOL` is specified. If `REG_NOTBOL` is specified, the shift state used is the shift state after the last call to the `regexec()` function.
- The information returned by the `regexec()` function in the `regmatch_t` structure has the shift-state at the start and end of the string added. This will assist an application to perform replacements or processing of the partial string. To perform replacements, the application must add the required shift-out and shift-in characters where necessary. No library functions are available to assist the application.
- If `MB_CUR_MAX` is specified as 4, but the charmap file does not specify the DBCS characters, and a collating-element (for example, `[:a:]`) is specified in the pattern, the DBCS characters will not match against the collating-element even if they have an equivalent weight to the collating-element.

#### Returned Value

If a match is found, `regexec()` returns the value 0; otherwise, it returns nonzero indicating either no match or an error.

#### Example CBC3BR09

```
/* CBC3BR09
   This example compiles an extended regular expression, and match against
   a string.
*/
#include <regex.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

main() {
    regex_t    preg;
    char       *string = "a simple string";
    char       *pattern = ".*(simple).*";
    int        rc;
    size_t     nmatch = 2;
    regmatch_t pmatch[2];

    if ((rc = regcomp(&preg, pattern, REG_EXTENDED)) != 0) {
        printf("regcomp() failed, returning nonzero (%d)\n", rc);
        exit(1);
    }
}
```

```
    if ((rc = regex(&preg, string, nmatch, pmatch, 0)) != 0) {  
        printf("failed to ERE match '%s' with '%s', returning %d.\n",  
            string, pattern, rc);  
    }  
  
    regfree(&preg);  
}
```

**Related Information**

- Chapters about internationalization in the *OS/390 C/C++ Programming Guide*
- “regex.h” on page 40
- “regcomp() — Compile Regular Expression” on page 1120
- “regerror() — Return Error Message” on page 1123
- “regfree() — Free Memory for Regular Expression” on page 1129

## regfree() — Free Memory for Regular Expression

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#include <regex.h>
```

```
void regfree(regex_t *preg);
```

### General Description

Frees any memory that was allocated by `regcomp()` to implement *preg*. The expression defined by *preg* is no longer a compiled regular or extended expression. (For a description of regular expressions, see “Regular Expressions” on page 1120.)

### Example CBC3BR10

```
/* CBC3BR10
   This example compiles an extended regular expression and a free regular
   expression.
*/
#include <regex.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

main() {
    regex_t    preg;
    char       *pattern = ".*(simple).*";
    int        rc;

    if ((rc = regcomp(&preg, pattern, REG_EXTENDED)) != 0) {
        printf("regcomp() failed, returning nonzero (%d)\n", rc);
        exit(1);
    }

    regfree(&preg);
}
```

### Related Information

- Chapters about internationalization in the *OS/390 C/C++ Programming Guide*
- “`regex.h`” on page 40
- “`regcomp()` — Compile Regular Expression” on page 1120
- “`regerror()` — Return Error Message” on page 1123
- “`regexexec()` — Execute Compiled Regular Expression” on page 1126

## release() — Delete a Load Module

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	C only	

### Format

```
#include <stdlib.h>
```

```
int release(void(*fetch_ptr)());
```

### General Description

Removes from memory the load modules retrieved by `fetch()` or fetch control blocks created by `fetchep()`. The *fetch\_ptr* parameter is obtained from a call to `fetch()` or `fetchep()`. Once released, the `fetch()` and any associated `fetchep()` pointers are no longer valid.

To avoid infringing on the user's name-space, this non-standard function has two names. One name, the external entry point name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use `LANGLVL(EXTENDED)`.

**Note:** The external entry point name for `release()` is `__rlse()`, **NOT** `__release()`.

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters `__rlse()`), or compile with `LANGLVL(EXTENDED)`. When you use `LANGLVL(EXTENDED)` any relevant information in the header is also exposed.

All fetched modules and fetch control blocks created by `fetchep()` are released automatically on program termination.

Using `release()` on a module obtained by using `fetch()` will also cause the `release()` of any child fetch control blocks created by `fetchep()` for this module. However, using `release()` on a child fetch control block will have no effect on the parent modules or sibling fetch control blocks obtained by using `fetch()`. Trying to use a fetch control block after it has been released will result in undefined behavior. (A Fetch Control Block (FECB) is an internal executable control block. The `fetch` pointer points to it.

When nonreentrant modules have been fetched multiple times, you should release them in the reverse order; otherwise, the load modules may not be deleted immediately.

### Returned Value

Returns the value 0 if successful, and a nonzero result otherwise.



**Example**

/\* The following C example uses the fetch() function to load a module, and later uses release() to delete the module from memory.

```
*/
#include <stdlib.h>

void (*fetch_ptr)();

int main(void) {
    fetch_ptr = fetch("sample");
    :
    release(fetch_ptr); /* all modules are released */
}
```

**Related Information**

- “stdlib.h” on page 45
- “fetch() — Get a Load Module” on page 369
- “fetchep() — Share Writable Static” on page 382

## remainder — Remainder Function

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>
```

```
double remainder(double x, double y);
```

### General Description

The remainder function returns the floating point remainder  $r = x - ny$  when  $y$  is nonzero. The value  $n$  is the integral value nearest the exact value  $x/y$ . When  $|n - x/y| = 1/2$  the value  $n$  is chosen to be even.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

If it succeeds, remainder returns the remainder of the division of  $x$  by  $y$  as described. If  $y$  is zero, remainder returns zero and sets `errno` to `EDOM`.

### Related Information

- “math.h” on page 35
- “abs() — Calculate Integer Absolute Value” on page 73

## remove() — Delete File

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
int remove(const char *filename);
```

### General Description

Deletes the file specified by *filename*, unless the file is open. The `remove()` function removes memory files and DASD data sets. (Non-DASD data sets, such as tapes, are not supported.) It also removes individual members of PDSs and PDSEs, and even removes memory files that simulate PDSs.

The interpretation of the file name passed to `remove()` depends on whether POSIX(ON) is specified. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information about using POSIX support. For full details about *filename* considerations, see one of the “Opening Files” sections in the *OS/390 C/C++ Programming Guide*.

Memory files must exist and they must be closed. However, if you have an OS/390 UNIX C application running POSIX(ON), memory files don't need to be closed when removing an HFS memory file. The OS/390 UNIX services rules of interoperability apply. See the appropriate “Opening Files” sections in the *OS/390 C/C++ Programming Guide*, for specifying file names for MVS data sets and HFS files.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Special Behavior for XPG4

If *filename* does not name a directory, `remove(filename)` is equivalent to `unlink(filename)`. If *filename* names a directory, `remove(filename)` is equivalent to `rmdir(filename)`.

### Returned Value

Returns the value 0 if it successfully deletes the file. A returned value of nonzero indicates an error.

### Example CBC3BR12

```
/* CBC3BR12
   When you invoke this example with a file name, the program attempts to
   remove that file. It issues a message if an error occurs.
*/
#include <stdio.h>
```

## remove

```
int main(int argc, char ** argv)
{
    if ( argc != 2 )
        printf( "Usage: %s fn\n", argv[0] );
    else
        if ( remove( argv[1] ) != 0 )
            printf( "Could not remove file\n" );
}
```

### Related Information

- “stdio.h” on page 43
- “fopen() — Open a File” on page 417
- “rename() — Rename File” on page 1136

## remque() — Remove an Element from a Doubly-linked List

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <search.h>
```

```
void remque(void *element);
```

### General Description

The `remque()` function removes the element pointed to by *element* from a doubly-linked list. The function operates on pointers to structures which have a pointer to their successor in the list as their first element, and a pointer to their predecessor as the second. The application is free to define the remaining contents of the structure, and manages all storage itself.

### Returned Value

The `remque()` function returns no value.

### Related Information

- “search.h” on page 40
- “insque() — Insert an Element into a Doubly-linked List” on page 671

## rename() — Rename File

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
int rename(const char *oldname, const char *newname);
```

### General Description

Changes the name of the file, from the name pointed to by *oldname* to the name pointed to by *newname*.

The *oldname* pointer must point to the name of an existing file. The *newname* pointer must not specify the name of an existing file. You cannot rename an open file. In case of an error, the name of the file is not changed.

The `rename()` function renames memory files and DASD data sets. (Non-DASD data sets, such as tapes, are not supported.) It also renames individual members of PDSs (and PDSEs); it even renames files that simulate PDSs.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Special Behavior for POSIX C

Memory files must be closed unless you are working under OS/390 UNIX services.

The interpretation of the file name passed to `rename()` depends on whether the program is running POSIX(ON) or POSIX(OFF). (See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information.)

You cannot rename an HFS file to an MVS data-set name or rename an MVS data set to an HFS file name.

Both *oldname* and *newname* must be of the same type, that is, both directories or both files.

If *newname* already exists, it is removed before *oldname* is renamed to *newname*. Thus, if *newname* specifies the name of an existing directory, it must be an empty directory.

If the *oldname* argument points to a symbolic link, the symbolic link is renamed. If the *newname* argument points to a symbolic link, the link is removed and *oldname* is renamed to *newname*. `rename()` does not affect any file or directory named by the contents of the symbolic link.

For `rename()` to succeed, the process needs write permission on the directory containing *oldname* and the directory containing *newname*. If *oldname* and *newname* are directories, `rename()` also needs write permission on the directories themselves.

If *oldname* and *newname* both refer to the same file, `rename()` returns successfully and performs no other action.

When `rename()` is successful, it updates the change and modification times for the parent directories of *oldname* and *newname*.

## Returned Value

Returns the value 0 if it is successful.

On an error, `rename()` returns a nonzero value and sets `errno` to one of the following.

EACCES	An error occurred for one of these reasons: <ul style="list-style-type: none"> <li>• The process did not have search permission on some component of the old or new path name.</li> <li>• The process did not have write permission on the parent directory of the file or directory to be renamed.</li> <li>• <i>oldname</i> or <i>newname</i> were directories.</li> <li>• The process did not have write permission on <i>oldname</i> or <i>newname</i>.</li> </ul>
EBUSY	<i>oldname</i> and <i>newname</i> specify directories, but one of them cannot be renamed because it is in use as a root or a mount point.
EINVAL	This error occurs for one of these reasons: <ul style="list-style-type: none"> <li>• <i>oldname</i> is part of the path name prefix of <i>newname</i>.</li> <li>• <i>oldname</i> or <i>newname</i> refers to either <code>.</code> (dot) or <code>..</code> (dot-dot).</li> </ul>
EIO	<b>Added for XPG4.2:</b> A physical I/O error has occurred.
EISDIR	<i>newname</i> is a directory, but <i>oldname</i> is not a directory.
ELOOP	A loop exists in symbolic links. This error is issued if the number of symbolic links encountered during resolution of <i>oldname</i> or <i>newname</i> is greater than <code>POSIX_SYMLINK_MAX</code> .
ENAMETOOLONG	<i>pathname</i> is longer than <code>PATH_MAX</code> characters, or some component of <i>pathname</i> is longer than <code>NAME_MAX</code> characters while <code>_POSIX_NO_TRUNC</code> is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds <code>PATH_MAX</code> . The <code>PATH_MAX</code> and <code>NAME_MAX</code> values can be determined using <code>pathconf()</code> .
ENOENT	No file or directory named <i>oldname</i> was found, or either <i>oldname</i> or <i>newname</i> was not specified.
ENOSPC	The directory intended to contain <i>newname</i> cannot be extended.
ENOTDIR	A component of the <i>pathname</i> prefix for <i>oldname</i> or <i>newname</i> is not a directory, or <i>oldname</i> is a directory and <i>newname</i> is a file that is not a directory.

ENOTEMPTY *newname* specifies a directory, but the directory is not empty.

EPERM or EACCES

**Added for XPG4.2:** The S\_ISVTX flag is set on the directory containing the file referred to by *oldname* and the caller is not the file owner, nor is the caller the directory owner, nor does the caller have appropriate privileges; or *newname* refers to an existing file, the S\_ISVTX flag is set on the directory containing this file and the caller is not the file owner, nor is the caller the directory owner, nor does the caller have appropriate privileges.

EROFS Renaming would require writing on a read-only file system.

EXDEV *oldname* and *newname* identify files or directories on different file systems. OS/390 UNIX services do not support links between different files systems.

### Example CBC3BR13

```
/* CBC3BR13
   This example takes two file names as input and uses rename() to change
   the file name from the first name to the second name.
*/
#include <stdio.h>

int main(int argc, char ** argv )
{
    if ( argc != 3 )
        printf( "Usage: %s old_fn new_fn\n", argv[0] );
    else if ( rename( argv[1], argv[2] ) != 0 )
        printf( "Could not rename file\n" );
}
```

### Related Information

- “stdio.h” on page 43
- “fopen() — Open a File” on page 417
- “remove() — Delete File” on page 1133



## rewind() — Set File Position to Beginning of File

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
void rewind(FILE *stream);
```

### General Description

Repositions the file position indicator of the stream pointed to by *stream*. A call to `rewind()` is the same as the statement below, except that `rewind()` also clears the error indicator for the *stream*.

```
(void) fseek(stream, 0L, SEEK_SET);
```

### Returned Value

There is no returned value. If an error occurs, `errno` is set. After the error, the file position does not change. The next operation may be either a read or a write operation.

### Special Behavior for XPG4.2

The `rewind()` function returns -1 and sets `errno` to `ESPIPE` if the underlying file type for the stream is a PIPE or a socket.

### Example

#### CBC3BR14

```
/* CBC3BR14
   This example first opens a file, myfile, for input and output.
   It writes integers to the file, uses rewind() to reposition the file
   pointer to the beginning of the file, and then reads the data back in.
*/
#include <stdio.h>

int main(void)
{
    FILE *stream;
    int data1, data2, data3, data4;
    data1 = 1; data2 = -37;

    /* Place data in the file */
    stream = fopen("myfile.dat", "w+");
    fprintf(stream, "%d %d\n", data1, data2);

    /* Now read the data file */
    rewind(stream);
    fscanf(stream, "%d", &data3);
    fscanf(stream, "%d", &data4);
    printf("The values read back in are: %d and %d\n",
```

## rewind

```
        data3, data4);  
}
```

### Output

The values read back in are: 1 and -37

### Related Information

- “stdio.h” on page 43
- “fgetpos() — Get File Position” on page 390
- “fseek() — Change File Position” on page 474
- “fsetpos() — Set File Position” on page 477
- “ftell() — Get Current File Position” on page 485

## rewinddir() — Reposition a Directory Stream to the Beginning

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <dirent.h>
```

```
void rewinddir(DIR *dir);
```

### General Description

Repositions an open directory stream to the beginning. *dir* points to a DIR object associated with an open directory.

The next call to `readdir()` reads the first entry in the directory. If the contents of the directory have changed since the directory was opened, a call to `rewinddir()` updates the directory stream so that a subsequent `readdir()` can read the new contents.

### Returned Value

There is no returned value.

### Example CBC3BR15

```
/* CBC3BR15
   This example produces the contents of a directory by opening it,
   rewinding it, and closing it.
*/
#define _POSIX_SOURCE
#include <dirent.h>
#include <errno.h>
#include <sys/types.h>
#include <stdio.h>

main() {
    DIR *dir;
    struct dirent *entry;

    if ((dir = opendir("/")) == NULL)
        perror("opendir() error");
    else {
        puts("contents of root:");
        while ((entry = readdir(dir)) != NULL)
            printf("%s ", entry->d_name);
        rewinddir(dir);
        puts("");
        while ((entry = readdir(dir)) != NULL)
            printf("%s ", entry->d_name);
        closedir(dir);
        puts("");
    }
}
```

## Output

contents of root:

```
. .. bin dev etc lib tmp u usr  
. .. bin dev etc lib tmp u usr
```

## Related Information

- “dirent.h” on page 24
- “stdio.h” on page 43
- “sys/types.h” on page 49
- “closedir() — Close a Directory” on page 194
- “opendir() — Open a Directory” on page 877
- “readdir() — Read an Entry from a Directory” on page 1086
- “seekdir() — Set Position of Directory Stream” on page 1158
- “telldir() — Current Location of Directory Stream” on page 1557

## rexec() — Execute Commands One at a Time on a Remote Host

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) MVS 5.2.2

### Format

```
#include <rexec.h>
```

```
int rexec (char **Host, int Port, char *User, char *Password,
          char *Command, int *ErrFileDescParam)
```

### General Description

The rexec (remote execution) subroutine allows the calling process to execute commands on a remote host. If the rexec connection succeeds, a socket in the Internet domain of type SOCK\_STREAM is returned to the calling process and is given to the remote command as standard input and standard output.

Host contains the name of a remote host that is listed in the /etc/hosts file or /etc/resolv.config file. If the name of the host is not found in either file, the rexec fails.

Port specifies the well-known DARPA Internet port to use for the connection. A pointer to the structure that contains the necessary port can be obtained by issuing the following library call: getservbyname("exec","tcp").

User and Password points to a user ID and password valid at the host. Command points to the name of the command to be executed at the remote host.

ErrFileDescParam specifies one of the following values:

- Not 0 (zero) = an auxiliary channel to a control process is set up, and a descriptor for it is placed in the ErrFileDescParam parameter. The control process provides diagnostic output from the remote command on this channel and also accepts bytes as signal numbers to be forwarded to the process group of the command. This diagnostic information does not include remote authorization failure, since this connection is set up after authorization has been verified.
- 0 (zero) = the standard error of the remote command is the same as standard output, and no provision is made for sending arbitrary signals to the remote process. In this case, however, it may be possible to send out-of-band data to the remote command.

This function is supported only in a POSIX program. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### **Returned Value**

If `rexec()` is successful, the system returns a socket to the remote command.

If the `rexec()` subroutine is unsuccessful, the system returns a `-1` indicating that the specified host name does not exist.

### **Related Information**

None

## rindex() — Search for Character

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <strings.h>
```

```
char *rindex(const char *string, int c);
```

### General Description

The `rindex()` function locates the last occurrence of `c` (converted to an unsigned char) in the string pointed to by `string`.

The string argument to the function must contain a null character (`\0`) marking the end of the string.

The `rindex()` function is identical to “`strrchr()` — Find Last Occurrence of Character in String” on page 1449.

### Returned Value

The `rindex()` function returns a pointer to the first occurrence of `c` (converted to an unsigned character) in the string pointed to by `string`. The function returns a null pointer if `c` was not found.

There are no `errno` values defined for `rindex()`.

### Related Information

- “`strings.h`” on page 46
- “`memchr()` — Search Buffer” on page 809
- “`index()` — Search for Character” on page 660
- “`strchr()` — Search for Character” on page 1416
- “`strrchr()` — Find Last Occurrence of Character in String” on page 1449
- “`strspn()` — Search String” on page 1451
- “`strstr()` — Locate Substring” on page 1453

## rint() — Round to Nearest Integral Value

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>
```

```
double rint(double x);
```

### General Description

The rint() function returns the integral value (represented as a double float) nearest *x* in the direction of 0.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

rint() always succeeds.

### Related Information

- “math.h” on page 35
- “abs() — Calculate Integer Absolute Value” on page 73
- “isnan() — Test for NaN” on page 712



rmdir() — Remove a Directory

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

Format

```
#define POSIX_SOURCE
#include <unistd.h>
```

```
int rmdir(const char *pathname);
```

General Description

Removes a directory, *pathname*, provided that the directory is empty. *pathname* must not end in . (dot) or .. (dot-dot).

If *pathname* refers to a symbolic link, rmdir() does not affect any file or directory named by the contents of the symbolic link. rmdir() does not remove a directory that still contains files or subdirectories.

| To provide an ASCII input/output format for applications using this function, define  
| feature test macro \_\_LIBASCII as described on page 22.

Special Behavior for XPG4.2

If *pathname* refers to a symbolic link, the rmdir() function fails and sets errno to ENOTDIR.

If no process currently has the directory open, rmdir() deletes the directory itself. The space occupied by the directory is freed for new use. If one or more processes have the directory open when it is removed, the directory itself is not removed until the last process closes the directory. New files cannot be created under a directory after the last link is removed, even if the directory is still open.

rmdir() removes the directory even if it is the working directory of a process.

If rmdir() is successful, the change and modification times for the parent directory are updated.

Returned Value

If successful, rmdir() returns the value 0. If unsuccessful, it returns the value -1 and sets errno to one of the following:

- EACCES      The process did not have search permission for some component of *pathname*, or it did not have write permission for the directory containing the directory to be removed.
- EBUSY      *pathname* cannot be removed, because it is currently being used by the system or a process.
- EINVAL      The last component of *pathname* contains a . (dot) or a .. (dot-dot).

ELOOP	A loop exists in symbolic links. More than POSIX_SYMLINK_MAX (an integer defined in the limits.h header file) symbolic links are detected in the resolution of <i>pathname</i> .
ENAMETOOLONG	<i>pathname</i> is longer than PATH_MAX characters or some component of <i>pathname</i> is longer than NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH_MAX. The PATH_MAX and NAME_MAX values can be determined using pathconf().
ENOENT	<i>pathname</i> does not exist, or it is an empty string.
ENOTDIR	Some component of the <i>pathname</i> prefix is not a directory.
ENOTEMPTY	The directory still contains files or subdirectories.
EROFS	The directory to be removed is on a read-only file system.

### Special Behavior for XPG4.2

The following new values of errno are added for XPG4.2:

EIO	A physical I/O error has occurred.
EPERM or EACCES	The S_ISVTX flag is set on the parent directory of the directory to be removed and the caller is not the owner of the directory to be removed, nor is the caller the owner of the parent directory, nor does the caller have the appropriate privileges.

### Example CBC3BR16

```

/* CBC3BR16
   This example removes a directory.
*/
#define _OPEN_SYS
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

main() {
    char new_dir[]="new_dir";
    char new_file[]="new_dir/new_file";
    int fd;

    if (mkdir(new_dir, S_IRWXU|S_IRGRP|S_IXGRP) != 0)
        perror("mkdir() error");
    else if ((fd = creat(new_file, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        close(fd);
        unlink(new_file);
    }

    if (rmdir(new_dir) != 0)
        perror("rmdir() error");
    else

```

```
| puts("removed!");  
| }
```

**Related Information**

- “unistd.h” on page 53
- “mkdir() — Make a Directory” on page 817
- “unlink() — Remove a Directory Entry” on page 1660

## rpmatch() — Test for a Yes/No Response Match

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#include <stdlib.h>
```

```
int rpmatch(const char *response);
```

### External Entry Point

```
@@RPMTCH, __rpmtch
```

### General Description

Tests whether a string pointed to by *response* matches either the affirmative or the negative response set by LC\_MESSAGES category in the current locale.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

### Returned Value

If the string pointed to by *response* matches the affirmative expression in the current locale, the rpmatch() function returns:

- 1                If the response string matches the affirmative expression.
- 0                If the response string matches the negative expression.
- 1               If the response string does not match either the affirmative or the negative expression.

### Example CBC3BR17

```
/* CBC3BR17
   This example asks for a reply and checks the response. */
#include "locale.h"
#include "stdio.h"
#include "stdlib.h"

main() {
    char *response;
    char buffer[100];
    int rc;

    printf("Enter reply");
    response = fgets(buffer, 100, stdin);
```

```
rc = rpmatch(response);  
if (rc > 0) printf("Response was affirmative\n");  
else if (rc == 0) printf("Response was negative\n");  
else printf("Response was neither negative or affirmative\n");  
}
```

### Related Information

- “stdlib.h” on page 45

## sbrk() — Change Space Allocation

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
```

```
void *sbrk(int incr);
```

### General Description

The `sbrk()` function is used to change the space allocated for the calling process. The change is made by adding `incr` bytes to the process' break value and allocating the appropriate amount of space. The amount of allocated space increases when `incr` is positive and decreases when `incr` is negative. If `incr` is zero the current value of the program break is returned by `sbrk()`. The newly-allocated space is set to 0. However, if the application first decrements and then increments the break value, the contents of the reallocated space are not zeroed.

The storage space from which the `brk()` and `sbrk()` functions allocate storage is separate from the storage space that is used by the other memory allocation functions (`malloc()`, `calloc()`, etc.). Because this storage space must be a contiguous segment of storage, it is allocated from the initial heap segment only and thus is limited to the initial heap size specified for the calling program or the largest contiguous segment of storage available in the initial heap at the time of the first `brk()` or `sbrk()` call. Since this is a separate segment of storage, the `brk()` and `sbrk()` functions can be used by an application that is using the other memory allocation functions. However, it is possible that the user's region may not be large enough to support extensive usage of both types of memory allocation.

Prior usage of the `sbrk()` function has been limited to specialized cases where no other memory allocation function performed the same function. Because the `sbrk()` function may be unable to sufficiently increase the space allocation of the process when the calling application is using other memory functions, the use of other memory allocation functions, such as `mmap()`, is now preferred because it can be used portably with all other memory allocation functions and with any function that uses other allocation functions. Applications that require the use of `brk()` and/or `sbrk()` should refrain from using the other memory allocation functions and should be run with an initial heap size that will satisfy the maximum storage requirements of the program.

The `sbrk()` function is not supported from a multi-threaded environment, it will return in error if it is invoked in this environment.

**Returned Value**

If successful, `sbrk()` returns 0. Otherwise, it returns -1 and sets `errno` to indicate the error. The following are the possible values of `errno`:

- |        |   |
|--------|---|
| ENOMEM | The requested change would allocate more space than allowed for the calling process.                      |
| EINVAL | The caller is running in a multi-threaded environment, this is not a valid environment for this function. |

**Related Information**

- “`brk()` — Change Space Allocation” on page 133

## scalb() — Load Exponent

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <math.h>
```

```
double scalb(double x, double n);
```

### General Description

The `scalb()` function computes  $x \bullet 16^n$ . 16 being the S/390 machine floating point radix.

If  $n$  is not an integer, it is silently truncated.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

If it succeeds, `scalb()` returns the function of its arguments as described above.

`scalb()` will fail under the following conditions:

- If the result would underflow, `scalb()` will return 0 and set `errno` to `ERANGE`.
- If the result would overflow, `scalb()` will return  $\pm\text{HUGE\_VAL}$  according to the sign of  $x$  and set `errno` to `ERANGE`.

### Related Information

- “`math.h`” on page 35
- “`ldexp()` — Multiply by a Power of Two” on page 738



**scanf() — Read and Format Data**

The information for this function is included in “fscanf() – scanf() – sscanf() — Read and Format Data” on page 464.

## seed48() — Pseudo-random Number Initializer

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>
```

```
unsigned short int *seed48(unsigned short int seed16v[3]);
```

### General Description

The drand48(), erand48(), jrand48(), lrand48(), mrand48() and nrand48() functions generate uniformly distributed pseudorandom numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The lcong48(), seed48(), and srand48() functions are initialization functions, one of which should be invoked before either the drand48(), lrand48() or mrand48() function is called.

The drand48(), lrand48() and mrand48() functions generate a sequence of 48-bit integer values,  $X(i)$ , according to the linear congruential formula:

$$X(n+1) = (aX(n) + c) \bmod (2^{**48}) \quad n \geq 0$$

The initial values of  $X$ ,  $a$ , and  $c$  are:

```
X(0) = 1
a    = 5deece66d (base 16)
c    = b          (base 16)
```

C/370 provides storage to save the most recent 48-bit integer value of the sequence,  $X(i)$ . This storage is shared by the drand48(), lrand48() and mrand48() functions. The seed48() function is used to reinitialize the most recent 48-bit value in this storage. The seed48() function replaces the low order (rightmost) 16 bits of this storage with *seed16v*[0], the middle order 16 bits with *seed16v*[1], and the high order 16 bits with *seed16v*[2].

The values  $a$  and  $c$ , may be changed by calling the lcong48() function. The seed48() function restores the initial values of  $a$  and  $c$ .

### Special Behavior for OS/390 UNIX Services

You can make the seed48() function and other functions in the drand48 family thread specific by setting the environment variable `_RAND48` to the value `THREAD` before calling any function in the drand48 family.

If you do not request thread specific behavior for the drand48 family, C/370 serializes access to the storage for  $X(n)$ ,  $a$  and  $c$  by functions in the drand48 family when they are called by a multithreaded application.

If thread specific behavior is requested, calls to the `drand48()`, `lrand48()` and `mrnd48()` functions from thread `t` generate a sequence of 48-bit integer values,  $X(t,i)$ , according to the linear congruential formula:

$$X(t,n+1) = (a(t)X(t,n) + c(t)) \bmod (2^{**}48) \quad n \geq 0$$

C/370 provides thread specific storage to save the most recent 48-bit integer value of the sequence,  $X(t,i)$ . When the `seed48()` function is called from thread `t`, it reinitializes the most recent 48-bit value in this storage. The `seed48()` function replaces the low order (rightmost) 16 bits of this storage with `seed16v[0]`, the middle order 16 bits with `seed16v[1]`, and the high order 16 bits with `seed16v[2]`.

The values of  $a(t)$  and  $c(t)$  may be changed by calling the `lcong48()` function from thread `t`. When the `seed48()` function is called from this thread, it restores the initial values of  $a(t)$  and  $c(t)$  for the thread which are:

```
a(t)  = 5deece66d (base 16)
c(t)  = b          (base 16)
```

### Returned Value

When the `seed48()` function is called, it saves the most recent 48-bit integer value in the sequence,  $X(i)$ , in an array of unsigned short ints provided by C/370 before reinitializing storage for the most recent value in the sequence,  $X(i)$ . The `seed48()` function returns a pointer to the array containing the saved value.

### Special Behavior for OS/390 UNIX Services

If thread specific behavior is requested for the `drand48` family and the `seed48()` function is called on thread `t`, it saves the most recent 48-bit integer value in the sequence,  $X(t,i)$ , for the thread in a thread specific array of unsigned short ints before reinitializing storage for the most recent value in the sequence,  $X(t,i)$ . The `seed48()` function returns a pointer to this thread specific array containing the saved value.

### Related Information

- “`stdlib.h`” on page 45
- “`drand48()` — Pseudo-random Number Generator” on page 286
- “`erand48()` — Pseudo-random Number Generator” on page 314
- “`jrand48()` — Pseudo-random Number Generator” on page 724
- “`lcong48()` — Pseudo-random Number Initializer” on page 736
- “`lrnd48()` — Pseudo-random Number Generator” on page 773
- “`mrnd48()` — Pseudo-random Number Generator” on page 843
- “`nrnd48()` — Pseudo-random Number Generator” on page 868
- “`srnd48()` — Pseudo-random Number Initializer” on page 1401

## seekdir() — Set Position of Directory Stream

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE
#include <dirent.h>
```

```
void seekdir(DIR *dirp, long int loc);
```

### General Description

The `seekdir()` function sets the position of the next `readdir()` operation on the directory stream specified by *dirp* to the position specified by *loc*. The value of *loc* should have been returned from an earlier call to `telldir()`. The new position reverts to the one associated with the directory stream when `telldir()` was performed. If the value of *loc* was not obtained from an earlier call to `telldir()` or if a call to `rewinddir()` occurred between the call to `telldir()` and the call to `seekdir()`, the result of subsequent calls to `readdir()` are unspecified.

**Note:** If files were added or removed from the directory after `telldir()` was called and before `seekdir()` is done, the results are also unspecified.

### Returned Value

The `seekdir()` function returns no value. If the *loc* argument is negative, the directory stream is unchanged.

### Related Information

- “`dirent.h`” on page 24
- “`stdio.h`” on page 43
- “`sys/types.h`” on page 49
- “`closedir()` — Close a Directory” on page 194
- “`opendir()` — Open a Directory” on page 877
- “`readdir()` — Read an Entry from a Directory” on page 1086
- “`rewinddir()` — Reposition a Directory Stream to the Beginning” on page 1141
- “`telldir()` — Current Location of Directory Stream” on page 1557

## select() — Monitor Activity on Files/Sockets and Message Queues

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#define _OPEN_MSGQ_EXT
#include <sys/types.h>
#include <sys/time.h>
#include <sys/msg.h>

int select(int nmsgsfds, fd_set *readlist,
           fd_set *writelist, fd_set *exceptlist,
           struct timeval *timeout);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#define _OPEN_MSGQ_EXT
#include <sys/types.h>
#include <sys/time.h>
#include <sys/msg.h>

int select(int nmsgsfds, fd_set *readlist,
           fd_set *writelist, fd_set *exceptlist,
           struct timeval *timeout);
```

`_OPEN_MSGQ_EXT` must be defined if message queues are to be monitored (X/Open and OE sockets only).

### General Description

The `select()` call monitors activity on a set of sockets and/or a set of message queue identifiers until a timeout occurs, to see if any of the sockets and message queues have read, write, or exception processing conditions pending. This call also works with regular file descriptors, pipes, and terminals.

#### Parameter

#### Description

*nmsgsfds*

The number of message queues and the number of file or socket descriptors to check.

This parameter is divided into two parts. The first half (the high-order 16 bits) gives the number of elements of an array that contains message queue identifiers. This number must not exceed the value 32,767.

The second half (the low-order 16 bits) gives the number of bits within a bit set that correspond to the file or socket descriptors to check. This value should equal the greatest descriptor number to check + 1.

## select

If either half of the *nmsgsfds* parameter is equal to a value of 0, the corresponding bit sets or arrays are assumed not to be present.

If `_OPEN_MSGQ_EXT` is not defined, only file or socket descriptors may be monitored. In this case *nmsgsfds* must be less than or equal to `FD_SETSIZE` (defined to be 2048 in `sys/time.h`), and greater than or equal to zero. Also, `FD_SETSIZE` may not be defined by your program.

The bit set used to specify file or socket descriptors is fixed in size with 1 bit for every possible file or socket. Use the *nmsgsfds* parameter to force `select()` to check only a subset of the allocated bit set.

If your application allocates sockets 3, 4, 5, 6, and 7 and you want to check all of your allocations, the second half of *nmsgsfds* should be set to 8, the highest descriptor you specified + 1. If your application checks sockets 3 and 4, the second half of *nmsgsfds* should be set to 5.

### *readlist, writelist, exceptlist*

Pointers to `fd_set` types, arrays of message queue identifiers, or `sellist` structures to check for reading, writing, and exceptional conditions, respectively. The type of parameter to pass depends on whether you want to monitor file/socket descriptors, message queue identifiers, or both. To monitor file/socket descriptors only, set the high-order halfword of *nmsgsfds* to 0, the low-order halfword to (highest descriptor number + 1), and use `fd_set` pointers. To monitor message queues only, set the low-order halfword of *nmsgsfds* to 0, the high-order halfword to the number of elements in each array you want `select()` to consider, and pass pointers to arrays of message queue identifiers. To monitor both, set *nmsgsfds* as described above, and pass pointers to `sellist` structures.

The **sellist** structure allows you to specify both file/socket descriptors and message queues. Your program must define the **sellist** structure in the following form:

```
struct sellist {  
    fd_set fdset;           /* file/socket descriptor bit set */  
    int  msgids[max_size]; /* array of message queue identifiers */  
};
```

If you use a `sellist` structure, the highest descriptor you can monitor is 2047.

The description of the type *fd\_set* is given below. Each integer of the `msgids` array specifies a message queue identifier whose status is to be checked. Elements with a value of -1 are acceptable and will be ignored. The value contained in the first half of *nmsgsfds* determines exactly how many elements of the array are to be checked.

### *timeout*

The pointer to the time to wait for the `select()` call to complete.

If *timeout* is not a NULL pointer, it specifies a maximum interval to wait for the selection to complete. The maximum timeout value is 31 days. If *timeout* is a

NULL pointer, the `select()` call blocks until a socket or message becomes ready. To poll the sockets and return immediately, *timeout* should be a non-NULL pointer to a zero-valued **timeval** structure.

To allow you to test more than one socket at a time, the sockets to test are placed into a bit set of type *fd\_set*. A bit set is a string of bits such that if *x* is an element of the set, the bit representing *x* is set to 1. If *x* is not an element of the set, the bit representing *x* is set to 0. For example, if socket 33 is an element of a bit set, then bit 33 is set to 1. If socket 33 is not an element of a bit set, then bit 33 is set to 0.

Because the bit sets contain a bit for every socket that a process can allocate, the size of the bit sets is constant. If your program needs to allocate a large number of sockets, you may need to increase the size of the bit sets. Increasing the size of the bit sets should be done when you compile the program. To increase the size of the bit sets, define `FD_SETSIZE` before including **sys/time.h**. `FD_SETSIZE` is the largest value of any socket that your program expects to use `select()` on. It is defined to be 2048 in **sys/time.h**.

**Note:** `FD_SETSIZE` may only be defined by the application program if the extended version of `select()` is used (by defining `_OPEN_MSGQ_EXT`). Do NOT define `FD_SETSIZE` in your program if a `select` structure will be used.

**Note:** The OS/390 UNIX POSIX.1 implementation allows you to control the maximum number of open descriptors allowed per process. This maximum possible value is 65535. If your application program requires a large number of either socket or file descriptors, you should protect your code from possible run-time errors by:

- Adding a check before your `select()` or `selectex()` calls to see if the bit set size contained in *nmsgsfds* is larger than `FD_SETSIZE`.
- Dynamically allocate bit strings large enough to hold the largest descriptor value in your application program, rather than rely on the static bit strings created at compile time. When allocating your own bit strings, use `malloc()` to define an area large enough to represent each bit, rounded up to the next 4-byte multiple. For example, if your largest descriptor value is 31, you need 4 bytes; if your largest descriptor is 32, you need 8 bytes.
- If you dynamically allocate your own bit strings, the `FD_ZERO()` macro will *not* work. The application must zero that storage, by using the `memset` function—that is, `memset(ptr, 0, mallocsize)`. The other macros can be used with the dynamically allocated bit strings, as long as the descriptor you are manipulating is within the bit string. If the descriptor number is larger than the bit string, unpredictable results can occur.

The application program must make sure that the parameters *readlist*, *writelist*, and *exceptlist* point to bit strings that are as large as the bit string size in parameter *nmsgsfds*. OS/390 UNIX services will try to access bits 0 through *n*−1 (where *n* = the value of the second halfword of *nmsgsfds*), for each of the bit strings. If the bit strings are too short, you will receive unpredictable results when you run your application program.

The following macros are provided to manipulate bit sets.

Macro	Description
<code>FD_ZERO(&amp;fdset)</code>	Sets all bits in the bit set <i>fdset</i> to zero. After this operation, the bit set does not contain sockets as elements. This macro should be called to initialize the bit set before calling <code>FD_SET()</code> to set a socket as a member.  <b>Note:</b> If you used <code>malloc()</code> to dynamically allocate a new area, the <code>FD_ZERO()</code> macro can cause unpredictable results and should <i>not</i> be used. You should zero the area using the <code>memset()</code> function.
<code>FD_SET(sock, &amp;fdset)</code>	Sets the bit for the socket <i>sock</i> to a 1, making <i>sock</i> a member of the bit set <i>fdset</i> .
<code>FD_CLR(sock, &amp;fdset)</code>	Clears the bit for the socket <i>sock</i> in bit set <i>fdset</i> . This operation sets the appropriate bit to a zero.
<code>FD_ISSET(sock, &amp;fdset)</code>	Returns <code>&gt; 0</code> if <i>sock</i> is a member of the bit set <i>fdset</i> . Returns zero if <i>sock</i> is not a member of <i>fdset</i> . (This operation returns the bit representing <i>sock</i> .)

The following macros are provided to manipulate the *nmsgsfds* parameter and the return value from `select()`:

Macro	Description
<code>_SET_FDS_MSGS(nmsgsfds, nmsgs, nfds)</code>	Sets the high-order halfword of <i>nmsgsfds</i> to <i>nmsgs</i> , and sets the low-order halfword of <i>nmsgsfds</i> to <i>nfds</i> .
<code>_NFDS(n)</code>	If the return value <i>n</i> from <code>select()</code> is non-negative, returns the number of descriptors that meet the read, write, and exception criteria. A descriptor may be counted multiple times if it meets more than one given criterion.
<code>_NMSGs(n)</code>	If the return value <i>n</i> from <code>select()</code> is non-negative, returns the number of message queues that meet the read, write, and exception criteria. A message queue may be counted multiple times if it meets more than one given criterion.

A socket is ready for reading when incoming data is buffered for it or when a connection request is pending. To test whether any sockets are ready for reading, use either `FD_ZERO()` or `memset()`, if the function was dynamically allocated, to initialize the *fdset* bit set in *readlist* and invoke `FD_SET()` for each socket to test.

A socket is ready for writing if there is buffer space for outgoing data. A non-blocking stream socket in the process of connecting (`connect()` returned `EINPROGRESS`) is selected for write when the `connect()` completes. A call to `write()`, `send()`, or `sendto()` does not block provided that the amount of data is less than the amount of buffer space. If a socket is selected for write, the amount of available buffer space is guaranteed to be at least as large as the size returned from using `SO_SNDBUF` with `getsockopt()`. To test whether any sockets are ready for writing, initialize the *fdset* bit set in *writelst* with either `FD_ZERO()` or `memset()`, if dynamically allocated, and use `FD_SET()` for each socket to test.



A message queue is ready for reading when any time it has a message on it. It is considered ready for writing when any time it is not full. A message queue is full when it has either reached its number of messages limit or its number of bytes limit. An exception condition exists when a message queue is deleted while a `select()` caller is waiting on the queue.

The programmer can pass NULL for any of the *readlist*, *writelist*, and *exceptlist* parameters. However, when they are not NULL, they must all point to the same type of structures. For example, suppose the *readlist* points to a *sellist*. If the *writelist* is not NULL, it must point to a *sellist* also. Now, let us say the *writelist* is not NULL. If the programmer wants to check a set of file descriptors for read status only, the appropriate bits in the bit set in the *sellist* structure pointed to by the *writelist* must be set to 0. If the programmer wants to check a set of message queues for write status only, the appropriate elements in the array in the *sellist* structure pointed to by the *readlist* must be set to -1. Regular files are always ready for reading and writing.

Because the sets of sockets passed to `select()` are bit sets, the `select()` call must test each bit in each bit set before polling the socket for its status. The `select()` call tests only sockets in the range 0 to  $n-1$  (where  $n$  = the value of the second halfword of *nmsgsfds*).

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

The value -1 indicates the error code should be checked for an error. The value zero indicates an expired time limit.

When the return value is greater than 0, then it is similar to *nmsgsfds* in that the high-order 16 bits give the number of message queues, and the low-order 16 bits give the number of descriptors. These values indicate the sum total that meet each of the read, write, and exception criteria. Note that a descriptor or a message queue may be counted multiple times if it meets more than one given criterion. Should the return value for message queues exceed the value 32,767, only 32,767 will be reported. This is to ensure that the return value does not appear to be negative. Should the return value for file/socket descriptors be greater than 65,535, only 65,535 will be reported.

If the return value is greater than zero, the files/sockets that are ready in each bit set are set to 1. Files/Sockets in each bit set that are not ready are set to zero. Use the macro `FD_ISSET()` with each file/socket to test its status. For those message queues that do not meet the conditions their identifiers in the *msgsid* arrays will be replaced with a value of -1.

Error Code	Description
EBADF	One of the bit sets specified an invalid socket or a message queue identifier is invalid. <code>FD_ZERO()</code> was probably not called to clear the bit set before the sockets were set.
EFAULT	One of the parameters contained an invalid address.

## select

EINTR	The select() function was interrupted before any of the selected events occurred and before the timeout interval expired.
EINVAL	One of the fields in the <b>timeval</b> structure is invalid, or there was an invalid <i>nmsgsfds</i> value.
EIO	One of the sockets or devices being selected has become inoperative due to a network problem. This can occur for a socket if TCP/IP is shutdown. To find out which descriptor is bad, you can code a loop to individually fstat() or select() on each descriptor, without waiting, until you get a failure.

### Example

In the following example, select() is used to poll sockets for reading (socket sr), writing (socket sw), and exception (socket se) conditions, and to check message queue ids mr, mw, and me.

```
#define _XOPEN_SOURCE_EXTENDED 1
#define _OPEN_MSGQ_EXT

#include <sys/types.h>
#include <sys/time.h>
#include <sys/msg.h>

struct sellist {
    fd_set fdset;
    int msgids[2];
};

/*
 * sock_msg_stats(sr, sw, se, mr, mw, me) - Print the status of
 *      sockets sr, sw, and se, and of message queue ids mr, mw,
 *      and me.
 */
int sock_msg_stats(sr, sw, se, mr, mw, me)
int sr, sw, se, mr, mw, me;
{
    struct sellist *reading, *writing, *excepting;
    struct sellist read, write, except;
    struct timeval timeout;
    int rc, max_sock, sock_size, nmsgsfds;
    int msgids[1];          /* we only check 1 message queue */

    /* What's the maximum socket number? */
    max_sock = MAX( sr, sw );
    max_sock = MAX( max_sock, se );

    /* initialize the static bit sets */
    FD_ZERO( &read.fdset );   reading = &read;
    FD_ZERO( &write.fdset );  writing = &write;
    FD_ZERO( &except.fdset ); excepting = &except;

    /* add sr, sw, and se to the appropriate bit set */
    FD_SET( sr, &reading->fdset );
    FD_SET( sw, &writing->fdset );
    FD_SET( se, &excepting->fdset );

    /* initialize the message id arrays */
```

```

reading->msgids[0] = mr;
writing->msgids[0] = mw;
excepting->msgids[0] = me;

/* set the nmsgsfds parameter */
_SET_FDS_MSGS( nmsgsfds, 1, max_sock+1 );

/* make select poll by sending a 0 timeval */
memset( &timeout, 0, sizeof(timeout) );
/* poll */
rc = select( nmsgsfds, reading, writing, excepting, &timeout);

if ( rc < 0 ) {
    /* an error occurred during the SELECT() */
    perror( "select" );
}
else if ( rc == 0 ) {
    /* no sockets or messages were ready in our little poll */
    printf( "nobody is home.\n" );
} else
    if ( _NFDS(rc) > 0 ) {
        /* at least one of the sockets is ready */
        printf("sr is %s\n",
            FD_ISSET(sr,&reading->fdset) ? "READY" : "NOT READY");
        printf("sw is %s\n",
            FD_ISSET(sw,&writing->fdset) ? "READY" : "NOT READY");
        printf("se is %s\n",
            FD_ISSET(se,&excepting->fdset) ? "READY": "NOT READY");
    } else
        if ( _NMSGs(rc) > 0 ) {
            /* at least one message queue is ready */
            printf("mr is %s\n",
                reading->msgids[0] == -1 ? "NOT READY" : "READY");
            printf("mw is %s\n",
                writing->msgids[0] == -1 ? "NOT READY" : "READY");
            printf("me is %s\n",
                excepting->msgids[0] == -1 ? "NOT READY" : "READY");
        }
}

```

## Related Information

- “msgctl() — Message Control Operations” on page 845
- “msgget() — Get Message Queue” on page 847
- “msgrcv() — Message Receive Operation” on page 850
- “msgsnd() — Message Send Operations” on page 852
- “poll() — Monitor Activity on File Descriptors and Message Queues” on page 910
- “selectex() — Monitor Activity on Files/Sockets and Message Queues” on page 1166

## selectex() — Monitor Activity on Files/Sockets and Message Queues

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 5.1

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#define _ALL_SOURCE
#define _OPEN_MSGQ_EXT
#include <sys/types.h>
#include <sys/time.h>
#include <sys/msg.h>

int selectex(int nmsgsfds, fd_set *readlist,
             fd_set *writelist,
             fd_set *exceptlist,
             struct timeval *timeout, int *ecbptr);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#define _ALL_SOURCE
#define _OPEN_MSGQ_EXT
#include <sys/types.h>
#include <sys/time.h>
#include <sys/msg.h>

int selectex(int nmsgsfds, fd_set *readlist,
             fd_set *writelist,
             fd_set *exceptlist,
             struct timeval *timeout, int *ecbptr);
```

`_OPEN_MSGQ_EXT` must be defined if message queues are to be monitored (X/Open and OE sockets only).

### General Description

The `selectex()` call provides an extension to the `select()` call by allowing you to use an ECB that defines an event not described by *readlist*, *writelist*, or *exceptlist*.

The `selectex()` call monitors activity on a set of files/sockets and message queues until a timeout occurs, or until the ECB is posted, to see if any of the files/sockets and message queues have read, write, or exception processing conditions pending.

See `select()` for more information.

Parameter	Description
<i>nmsgsfds</i>	The number of message queues and the number of file or socket descriptors to check. (Refer to <code>select()</code> for a full description of this and other parameters below.)
<i>readlist</i>	A pointer to an <code>fd_set</code> type, array of message queue identifiers, or <i>sellist</i> structure specifying descriptors and message queues to check for reading.

<i>writelst</i>	A pointer to an <code>fd_set</code> type, array of message queue identifiers, or <i>sellist</i> structure specifying descriptors and message queues to check for writing.
<i>exceptlst</i>	A pointer to an <code>fd_set</code> type, array of message queue identifiers, or <i>sellist</i> structure specifying descriptors and message queues to be checked for exceptional pending conditions.
<i>timeout</i>	The pointer to the time to wait for the <code>selectex()</code> call to complete.
<i>ecbptr</i>	This variable can contain one of the following values: <ol style="list-style-type: none"> <li>1. A pointer to a user event control block. To specify this usage of <i>ecbptr</i>, the high-order bit must be set to '0'B.</li> <li>2. A pointer to a list of ECBs. To specify this usage of <i>ecbptr</i>, the high order bit must be set to '1'B.  The list can contain the pointers for up to 1013 ECBs. The high-order bit of the last pointer in the list must be set to '1'B.</li> <li>3. A NULL pointer. This indicates no ECBs are specified.</li> </ol>

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

The value `-1` indicates the error code should be checked for an error. The value `0` indicates an expired time limit or that the ECB is posted.

When the return value is greater than `0`, then it is similar to *nmsgsfds* in that the high-order 16 bits give the number of message queues, and the low-order 16 bits give the number of descriptors. These values indicate the sum total that meet each of the read, write, and exception criteria. Note that a descriptor or a message queue may be counted multiple times if it meets more than one requested criterion. Should the return value for message queues exceed the value 32,767, only 32,767 will be reported. This is to ensure that the return value does not appear to be negative. Should the return value for file/socket descriptors be greater than 65,535, only 65,535 will be reported.

If the return value is greater than zero, the files/sockets that are ready in each bit set are set to 1. Files/Sockets in each bit set that are not ready are set to zero. Use the macro `FD_ISSET()` with each socket to test its status. For those message queues that do not meet the conditions their identifiers in the *msgsid* array will be replaced with a value of `-1`.

Error Code	Description
EBADF	One of the descriptor sets specified an incorrect descriptor or a message queue identifier is invalid.
EFAULT	One of the parameters contained an invalid address.
EINTR	The <code>selectex()</code> function was interrupted before any of the selected events occurred and before the timeout interval expired.
EINVAL	One of the fields in the <i>timeval</i> structure is incorrect.

**EIO** One of the sockets or devices being selected has become inoperative due to a network problem. This can occur for a socket if TCP/IP is shutdown. To find out which descriptor is bad, you can code a loop to individually `fstat()` or `select()` on each descriptor, without waiting, until you get a failure.

**Related Information**

- “`accept()` — Accept a New Connection on a Socket” on page 75
- “`connect()` — Connect a Socket” on page 214
- “`msgctl()` — Message Control Operations” on page 845
- “`msgget()` — Get Message Queue” on page 847
- “`msgrcv()` — Message Receive Operation” on page 850
- “`msgsnd()` — Message Send Operations” on page 852
- “`poll()` — Monitor Activity on File Descriptors and Message Queues” on page 910
- “`recv()` — Receive Data on a Socket” on page 1103
- “`selectex()` — Monitor Activity on Files/Sockets and Message Queues” on page 1166
- “`send()` — Send Data on a Socket” on page 1178

# semctl() — Semaphore Control Operations

## Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 5.2.2

## Format

```
#define _XOPEN_SOURCE
#include <sys/sem.h>
```

```
int semctl(int semid, int semnum, int cmd, ...);
```

## General Description

The semctl() function performs control operations in semaphore set *semid* as specified by the argument *cmd*.

Depending on the value of argument *cmd*, argument *semnum* may be ignored or identify one specific semaphore number.

The fourth argument is optional and depends upon the operation requested. If required, it is of type *union semun*, which the application program must explicitly declare:

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
} arg;
```

Each semaphore in the semaphore set is represented by the following anonymous data structure:

unsigned short int	semval	Semaphore value
pid_t	sempid	Process ID of last operation
unsigned short int	semncnt	Number of processes waiting for semval to become greater than current value
unsigned short int	semzcnt	Number of processes waiting for semval to become zero

When semctl() is used to identify one specific semaphore number for commands GETVAL, SETVAL, GETPID, GETNCNT, and GETZCNT, then references are made to this anonymous data structure for the semaphore *semnum*.

The following semaphore control operations as specified by argument *cmd* may be specified. The level of permission required for each operation is shown with each command. These symbolic constants are defined by the <sys/sem.h> header:

GETVAL	Returns the value of semval, if the current process has read permission.
SETVAL	Sets the value of semval to arg.val, where arg is the value of the fourth argument to semctl(). When this command is successfully executed, the semadj value corresponding to the specified

	semaphore in all processes is cleared. This command requires alter permission. For an <code>__IPC_BINSEM</code> semaphore set the only values that may be set are zero and one.
GETPID	Returns the most recent process to update the semaphore ( <code>sempid</code> ), if the current process has read permission.
GETNCNT	Returns the number of threads waiting on the semaphore to become greater than the current value, if the current process has read permission.
GETZCNT	Returns the number of threads waiting on the semaphore to become zero, if the current process has read permission. For an <code>__IPC_BINSEM</code> semaphore set this operation will always return a zero; threads are not allowed to wait for the semaphore to become zero in this type of semaphore set.
GETALL	Stores <code>semval</code> s for each semaphore in the semaphore set and place into the array pointed to by <code>arg.array</code> , where <code>arg</code> is the fourth argument to <code>semctl()</code> . <code>GETALL</code> requires read permission. It is the caller's responsibility to insure that the storage allocated is large enough to hold the number of semaphore elements. The number of semaphore values stored is <code>sem_nsems</code> , which may be obtained using the <b>IPC_STAT</b> command.
SETALL	Sets <code>semval</code> values for each semaphore in the semaphore set according to the array pointed to by <code>arg.array</code> , where <code>arg</code> is the fourth argument to <code>semctl()</code> . <code>SETALL</code> requires alter permission. Each <code>semval</code> value must be zero or positive. When this command is successfully executed, the <code>semadj</code> values corresponding to each specified semaphore in all processes are cleared. It is the caller's responsibility to insure that the storage allocated is large enough to hold the number of semaphore elements. The number of semaphore values set is <code>sem_nsems</code> , which may be obtained using the <b>IPC_STAT</b> command. If <code>__IPC_BINSEM</code> was specified on the <code>semget</code> , this option should not be used while there is the possibility of other threads performing semaphore operations on this semaphore, as there may be no serialization while updating the semaphore values; therefore a <code>SETALL</code> will not be allowed after a <code>semop</code> has been done to the <code>__IPC_BINSEM</code> semaphore set. Also, for the <code>__IPC_BINSEM</code> semaphore set, the only values that may be set are zero and one.
IPC_STAT	This command obtains status information for the semaphore identifier specified by <i>semid</i> . This requires read permission. This information is stored in the address specified by the fourth argument defined by data structure <code>semid_ds</code> .
IPC_SET	Set the value of the <code>sem_perm.uid</code> , <code>sem_perm.gid</code> , and <code>sem_perm.mode</code> in <code>semid_ds</code> data structure for the semaphore identifier specified by <i>semid</i> . These values are set to the values found in <code>semid_ds</code> structure pointed to by the fourth argument.  Any value for <code>sem_perm.uid</code> and <code>semperm.gid</code> may be set.  Only mode bits defined under <code>semget()</code> function argument <i>semflg</i> may be set in <code>sem_perm.mode</code> .



This command can only be executed by a process that has an effective user ID equal to either that of a process with appropriate privileges or to the value of `sem_perm.cuid` or `sem_perm.uid` in the `semid_ds` structure associated with *semid*.

**IPC\_RMID** Remove the semaphore identifier specified by argument *semid* from the system and free the storage for the set of semaphores in the `semid_ds` structure.

This command can only be executed by a process that has an effective user ID equal to either that of a process with appropriate privileges or to the value of `sem_perm.cuid` or `sem_perm.uid` in the `semid_ds` structure associated with *semid*. For an `__IPC_BINSEM` semaphore set, it is recommended that all use of `semop` should be completed before removing the semaphore ID.

## Returned Value

If successful, the value returned by `semctl()` depends on the value of the argument *cmd* as follows:

<b>GETVAL</b>	value of <code>semval</code> is returned
<b>GETPID</b>	value of <code>sempid</code> is returned
<b>GETNCNT</b>	value of <code>semncnt</code> is returned
<b>GETZCNT</b>	value of <code>semzcnt</code> is returned
All others	value of zero is returned

If unsuccessful, `semctl()` returns `-1`, and returns the error value in `errno`. The following are the possible values of `errno`:

<b>EACCES</b>	Operation permission (read or write) is denied to the calling process.
<b>EINVAL</b>	The value of argument <i>semid</i> is not a valid semaphore identifier, or the value of <i>semnum</i> is less than zero or greater than or equal to the number of semaphores in the set, or the argument <i>cmd</i> is not a valid command, or the bits specified for <code>sem_perm.mode</code> are undefined. Note that the valid range of <i>semnum</i> is 0 to (number of semaphores in the set minus 1).
<b>EPERM</b>	The argument <i>cmd</i> has a value of <b>IPC_RMID</b> or <b>IPC_SET</b> and the effective user ID of the caller is not that of a process with appropriate privileges and is not the value of <code>sem_perm.cuid</code> or <code>sem_perm.uid</code> in the <code>semid_ds</code> data structure associated with <i>semid</i> .
<b>ERANGE</b>	The argument <i>cmd</i> has a value of <b>SETVAL</b> or <b>SETALL</b> and the <code>semval</code> value to be set exceeds the system limit as defined in <code>&lt;sys/sem.h&gt;</code> .

## Related Information

- “`sys/sem.h`” on page 48
- “`sys/ipc.h`” on page 47
- “`semget()` — Get a Set of Semaphores” on page 1172
- “`semop()` — Semaphore Operations” on page 1175

## semget() — Get a Set of Semaphores

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE
#include <sys/sem.h>
```

```
int semget(key_t key, int nsems, int semflg);
```

### General Description

The `semget()` function returns the semaphore identifier associated with *key*.

A semaphore identifier is created with a `sem_ds` data structure, see `<sys/sem.h>`, associated with *nsems* semaphores when any of the following is true:

- Argument *key* has a value of **IPC\_PRIVATE**
- Argument *key* is not associated with a semaphore ID and (`semflg & IPC_CREAT`) is non zero.

Valid values for the field *semflg* include any combination of the following defined in `<sys/ipc.h>` and `<sys/modes.h>`:

IPC_CREAT	Creates a semaphore if the <i>key</i> specified does not already have an associated ID. <b>IPC_CREATE</b> is ignored when <b>IPC_PRIVATE</b> is specified.
IPC_EXCL	Causes the <code>semget()</code> function to fail if the <i>key</i> specified has an associated ID. <b>IPC_EXCL</b> is ignored when <b>IPC_CREAT</b> is not specified or <b>IPC_PRIVATE</b> is specified.
__IPC_BINSEM	Binary semaphore - semaphore must behave in a binary manner: number of semaphore operations must be 1 and the <code>semop</code> must be 1 with a <code>semval</code> of 0 or the <code>semop</code> must be -1 with a <code>semval</code> of 0 or 1. <b>SEM_UNDO</b> is not allowed on a <code>semop()</code> with this option. The use of this flag will cause improved performance if the PLO instruction is available on the hardware.  See <i>OS/390 C/C++ Programming Guide</i> for further information on semaphore performance.
S_IRUSR	Permits read access when the effective user ID of the caller matches either <code>sem_perm.cuid</code> or <code>sem_perm.uid</code> .
S_IWUSR	Permits write access when the effective user ID of the caller matches either <code>sem_perm.cuid</code> or <code>sem_perm.uid</code> .
S_IRGRP	Permits read access when the effective group ID of the caller matches either <code>sem_perm.cgid</code> or <code>sem_perm.gid</code> .
S_IWGRP	Permits write access when the effective group ID of the caller matches either <code>sem_perm.cgid</code> or <code>sem_perm.gid</code> .
S_IROTH	Permits others read access

S\_IWOTH      Permits others write access

When a semaphore set associated with argument *key* already exists, setting **IPC\_EXCL** and **IPC\_CREAT** in argument *semflg* will force `semget()` to fail.

When a `sem_ds` data structure is created the following anonymous data structure is created for each semaphore in the set:

unsigned short int	<code>semval</code>	Semaphore value
<code>pid_t</code>	<code>sempid</code>	Process ID of last operation
unsigned short int	<code>semncnt</code>	Number of processes waiting for <code>semval</code> to become greater than current value
unsigned short int	<code>semzcnt</code>	Number of processes waiting for <code>semval</code> to become zero

The following fields are initialized when a `sem_ds` data structure is created:

- The fields `sem_perm.cuid` and `sem_perm.uid` are set equal to the effective user ID of the calling process.
- The fields `sem_perm.cgid.` and `sem_perm.gid` are set equal to effective group ID of the calling process.
- The low order 9 bits of `sem_perm.mode` are set to the value in the low order 9 bits of *semflg*.
- The field `sem_nsems` is set to the value of *nsems*.
- The field `sem_otime` is set to 0.
- The field `sem_ctime` is set to the current time.
- The anonymous data structure containing `semval` for each semaphore is not initialized. `semctl()` commands **SETVAL** and **SETALL** should be used to initialize each semaphore's `semval` value.

## Returned Value

If successful, `semget()` returns a non-negative semaphore identifier.

If unsuccessful, `semget()` returns -1, and returns the error value in `errno`. The following are the possible values of `errno`:

EACCES	A semaphore identifier exists for the argument <i>key</i> , but access permission as specified by the low order 9 bits of <i>semflg</i> could not be granted.
EEXIST	A semaphore identifier exists for the argument <i>key</i> and both <b>IPC_CREAT</b> and <b>IPC_EXCL</b> are specified in <i>semflg</i> .
EINVAL	The value of <i>nsems</i> is either less than zero or greater than the system limit. A semaphore identifier associated with <i>key</i> does not exist and the <i>nsems</i> is zero. A semaphore identifier associated with <i>key</i> already exists and the <i>nsems</i> value specified on <code>semget()</code> when the semaphore identifier was created is less than the <i>nsems</i> value on the current <code>semget()</code> . The <i>semflg</i> argument specified flags not currently supported.
ENOENT	A semaphore identifier does not exist for the argument <i>key</i> and <b>IPC_CREAT</b> is not specified.
ENOSPC	A system limit of number of semaphore identifiers has been reached.

When *semflg* equals 0, the following applies:

- If a semaphore identifier has already been created with *key* earlier, and the calling process of this `semget()` has read and/or write permissions to it, then `semget()` returns the associated semaphore identifier.
- If a semaphore identifier has already been created with *key* earlier, and the calling process of this `semget()` does not have read and/or write permissions to it, then `semget()` returns -1 and sets `errno` to `EACCES`.
- If a semaphore identifier has not been created with *key* earlier, then `semget()` returns -1 and sets `errno` to `ENOENT`.

### Related Information

- “`sys/sem.h`” on page 48
- “`sys/ipc.h`” on page 47
- “`sys/stat.h`” on page 48
- “`sys/types.h`” on page 49
- “`semctl()` — Semaphore Control Operations” on page 1169
- “`semop()` — Semaphore Operations” on page 1175
- “`ftok()` — Generate an Interprocess Communication (IPC) key” on page 488

# semop() — Semaphore Operations

## Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 5.2.2

## Format

```
#define _XOPEN_SOURCE
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, size_t nsops);
```

## General Description

The semop() function performs semaphore operations atomically on a set of semaphores associated with argument *semid*. The argument *sops* is a pointer to an array of sembuf data structures. The argument *nsops* is the number of sembuf structures in the array.

The structure sembuf is defined as follows:

short	sem_num	Semaphore number in the range 0 to (nsems - 1)
short	sem_op	Semaphore operation
short	sem_flg	Operation flags

Each semaphore in the semaphore set, identified by *sem\_num*, is represented by the following anonymous data structure. This data structure for all semaphores is updated atomically when semop() returns successfully:

unsigned short int	semval	Semaphore value
pid_t	sempid	Process ID of last operation
unsigned short int	semncnt	Number of processes waiting for <i>semval</i> to become greater than current value
unsigned short int	semzcnt	Number of processes waiting for <i>semval</i> to become zero

Each semaphore operation specified by *sem\_op* is performed on the corresponding semaphore specified by *semid* and *sem\_num*.

The variable *sem\_op* specifies one of three semaphore operations:

1. If *sem\_op* is a negative integer and the calling process has alter permission, one of the following will occur:
  - If *semval*, see <sys/sem.h>, is greater than or equal to the absolute value of *sem\_op*, the absolute value of *sem\_op* is subtracted from *semval*.
  - If *semval* is less than the absolute value of *sem\_op* and (*sem\_flg* & **IPC\_NOWAIT**) is nonzero, semop() will return immediately.
  - If *semval* is less than the absolute value of *sem\_op* and (*sem\_flg* & **IPC\_NOWAIT**) is zero, semop() will increment the *semncnt* associated with the specified semaphore and suspend execution of the calling process until one of the following conditions occurs:

- The value of `semval` becomes greater than or equal to the absolute value of `sem_op`. When this occurs, the value of `semncnt` associated with the specified semaphore is decremented, the absolute value of `sem_op` is subtracted from `semval`.
  - The *semid* for which the calling process is awaiting action is removed from the system. When this occurs, `errno` is set equal to `EIDRM` and `-1` is returned.
  - The calling process receives a signal that is to be caught. When this occurs, the value of `semncnt` associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in `sigaction()`.
2. If `sem_op` is a positive integer and the calling process has alter permission, the value of `sem_op` is added to `semval`.
  3. If `sem_op` is zero and the calling process has read permission, one of the following will occur:
    - If `semval` is zero, `semop()` will return immediately.
    - If `semval` is nonzero and `(sem_flg&IPC_NOWAIT)` is nonzero, `semop()` will return immediately.
    - If `semval` is nonzero and `(sem_flg&IPC_NOWAIT)` is 0, `semop()` will increment the `semzcnt` associated with the specified semaphore and suspend execution of the calling thread until one of the following occurs:
      - The value of `semval` becomes 0, at which time the value of `semzcnt` associated with the specified semaphore is decremented.
      - The *semid* for which the calling process is awaiting action is removed from the system. When this occurs, `errno` is set equal to `EIDRM` and `-1` is returned.
      - The calling process receives a signal that is to be caught. When this occurs, the value of `semzcnt` associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in `sigaction()`.
      - Upon successful completion, the value of `sempid` for each semaphore specified in the array pointed to by *sops* is set equal to the process ID of the calling process.

`sem_flg` contains the **IPC\_NOWAIT** and **SEM\_UNDO** flags described as follows:

#### IPC\_NOWAIT

Will cause `semop()` to return `EAGAIN` rather than place the thread into wait state.

#### SEM\_UNDO

Will result in `semadj` adjustment values being maintained for each semaphore on a per process basis. If `sem_op` value is not equal to zero and **SEM\_UNDO** is specified, then `sem_op` value is subtracted from the current process' `semadj` value for that semaphore. When the current process is terminated, see `exit()`, the `semadj` value(s) will be added to the `semval` for each semaphore. The `semctl()` command **SETALL** may be used to clear all `semadj` values in all processes. If `__IPC_BINSEM` was specified on `semget` for this semaphore, the `Sem_UNDO` flag will cause an error to be returned.

A semaphore set created with the `__IPC_BINSEM` flag must behave in the following manner: number of semaphore operations must be 1 and the semop must be +1 with a semval of 0 or the semop must be -1 with a semval of 0 or 1. `SEM_UNDO` is not allowed on a semop() with this option. The use of this flag will cause improved performance if the PLO instruction is available on the hardware.

### Returned Value

If successful, semop() returns zero. Also the *semid* parameter value for each semaphore that is operated upon is set to the process ID of the calling process.

If unsuccessful, semop() returns -1, and returns the error value in errno. The following are the possible values of errno:

E2BIG	The value <i>nsops</i> is greater than the system limit.
EACCES	Operation permission is denied to the calling process. Read access is required when <i>sem_op</i> is zero. Write access is required when <i>sem_op</i> is not zero.
EAGAIN	The operation would result in suspension of the calling process but <b>IPC_NOWAIT</b> in <i>sem_flg</i> was specified.
EFBIG	<i>sem_num</i> is less than zero or greater or equal to the number of semaphores in the set specified on in semget() argument <i>nsems</i> .
EIDRM	<i>semid</i> was removed from the system while the invoker was waiting.
EINTR	semop() was interrupted by a signal.
ENOSPC	The limit on the number of individual processes requesting a <b>SEM_UNDO</b> would be exceeded.
ERANGE	An operation would cause <i>semval</i> or <i>semadj</i> to overflow the system limit as defined in <sys/sem.h>.
EINVAL	The value of argument <i>semid</i> is not a valid semaphore identifier. For an <code>__IPC_BINSEM</code> semaphore set, the <i>sem_op</i> is other than +1 for a <i>sem_val</i> of 0 or a -1 for a <i>sem_val</i> of 0 or 1. Also, for an <code>__IPC_BINSEM</code> semahore set, the number of semaphore operations is greater than one.

### Related Information

- “sys/sem.h” on page 48
- “sys/ipc.h” on page 47
- “sys/types.h” on page 49
- “semget() — Get a Set of Semaphores” on page 1172
- “semctl() — Semaphore Control Operations” on page 1169
- “exec Functions” on page 322
- “rexec() — Execute Commands One at a Time on a Remote Host” on page 1143
- “exit() — End Program” on page 330
- “\_exit() — End a Process and Bypass the Cleanup” on page 332
- “fork() — Create a New Process” on page 422

## send() — Send Data on a Socket

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>
```

```
ssize_t send(int socket, const void *buffer, size_t length, int
flags);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/socket.h>
```

```
int send(int socket, char *buffer, int length, int flags);
```

### General Description

The `send()` call sends data on the socket with descriptor *socket*. The `send()` call applies to all connected sockets.

#### Parameter Description

<i>socket</i>	The socket descriptor.
<i>msg</i>	The pointer to the buffer containing the message to transmit.
<i>length</i>	The length of the message pointed to by the <i>msg</i> parameter.
<i>flags</i>	<p>The <i>flags</i> parameter is set by. If more than one flag is specified, the logical OR operator ( ) must be used to separate them.</p> <p><b>MSG_OOB</b> Sends out-of-band data on sockets that support this notion. Only SOCK_STREAM sockets support out-of-band data. The out-of-band data is a single byte.</p> <p>Before out-of-band data can be sent between two programs, there must be some coordination of effort. If the data is intended to not be read inline, the recipient of the out-of-band data must specify the recipient of the SIGURG signal that is generated when the out-of-band data is sent. If no recipient is set, no signal is sent. The recipient is set up by using F_SETOWN operand of the fcntl command, specifying either a pid or gid. For more information on this operand, refer to the fcntl command.</p> <p>The recipient of the data determines whether to receive out-of-band data inline or not inline by the setting of the SO_OOBINLINE option of setsockopt(). For more information on receiving out-of-band data, refer to the setsockopt(), recv(), recvfrom() and recvmsg() commands.</p>



**MSG\_DONTROUTE**

The `SO_DONTROUTE` option is turned on for the duration of the operation. This is usually used only by diagnostic or routing programs.

If there is not enough available buffer space to hold the socket data to be transmitted, and the socket is in blocking mode, `send()` blocks the caller until additional buffer space becomes available. If the socket is in nonblocking mode, `send()` returns a `-1` and sets the error code to `EWOULDBLOCK`. See “`fcntl()` — Control Open File Descriptors” on page 350 or “`ioctl()` — Control Device” on page 672 for a description of how to set nonblocking mode.

For datagram sockets, this call sends the entire datagram, provided that the datagram fits into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application wishes to send 1000 bytes, each call to this function can send 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been sent.

**Special Behavior for C++**

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

**Returned Value**

The value `-1` indicates locally detected errors. The value of the error code indicates the specific error. No indication of failure to deliver is implicit in a `send()` routine.

A value of 0 or greater indicates the number of bytes sent, however, this does not assure that data delivery was complete. A connection can be dropped by a peer socket and a `SIGPIPE` signal generated at a later time if data delivery is not complete.

Error Code	Description
<code>EBADF</code>	<i>socket</i> is not a valid socket descriptor.
<code>ECONNRESET</code>	A connection was forcibly closed by a peer.
<code>EDESTADDRREQ</code>	The socket is not connection-oriented and no peer address is set.
<code>EFAULT</code>	Using the <i>msg</i> and <i>length</i> parameters would result in an attempt to access storage outside the caller's address space.
<code>EINTR</code>	A signal interrupted <code>send()</code> before any data was transmitted.
<code>EIO</code>	There has been a network or transport failure.
<code>EMSGSIZE</code>	The message was too big to be sent as a single datagram.
<code>ENOBUFS</code>	Buffer space is not available to send the message.
<code>ENOTCONN</code>	The socket is not connected.
<code>ENOTSOCK</code>	The descriptor is for a file, not for a socket.
<code>EOPNOTSUPP</code>	The <i>socket</i> argument is associated with a socket that does not support one or more of the values set in <i>flags</i> .

EPIPE	For a connected stream socket the connection to the peer socket has been lost. A SIGPIPE signal is sent to the calling process.
EWOULDBLOCK	<i>socket</i> is in nonblocking mode and no data buffers are available.

**Related Information**

- “connect() — Connect a Socket” on page 214
- “fcntl() — Control Open File Descriptors” on page 350
- “getsockopt() — Get the Options Associated with a Socket” on page 606
- “ioctl() — Control Device” on page 672
- “read() — Read From a File or Socket” on page 1080
- “readv() — Read Data on a File or Socket and Store in a Set of Buffers” on page 1093
- “recv() — Receive Data on a Socket” on page 1103
- “recvfrom() — Receive Messages on a Socket” on page 1106
- “recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers” on page 1110
- “select() — Monitor Activity on Files/Sockets and Message Queues” on page 1159
- “selectex() — Monitor Activity on Files/Sockets and Message Queues” on page 1166
- “sendmsg() — Send Messages on a Socket” on page 1185
- “sendto() — Send Data on a Socket” on page 1190
- “socket() — Create a Socket” on page 1371
- “write() — Write Data on a File or Socket” on page 1780
- “writev() — Write Data on a File or Socket Socket from an Array” on page 1785

# send\_file() — Send File Data Over a Socket

## Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 UNIX

## Format

```
#define _OPEN_SYS_SOCKET_EXT2
#include <sys/socket.h>

int send_file(int *socket_ptr,
              struct sf_parms *sf_struct,
              int options);
```

## General Description

The send\_file() function sends data from the file associated with the open file handle over the connection associated with the socket.

The function takes the following arguments:

- socket\_ptr*     A socket file descriptor.
- sf\_parms*       A structure that contains variables needed by sendfile - header information, file information, trailer information and results of operation.
- options*        Specifies one of the following options:
  - SF\_CLOSE        Close the connection after the data has been successfully sent or queued for transmission.
  - SF\_REUSE        Prepare the socket for reuse after the data has been successfully sent or queued for transmission and the existing connection closed.

## Send\_File Structure - sf\_parms

The *sf\_parms* is a structure that contains the file descriptor, a header data buffer, and a trailer data buffer.

*sf\_parms* is defined in <sys/sockets.h> and contains the following variables:

- header\_data*     Pointer to a buffer that contains header data which is to be sent before the file data. It may be a null pointer if *header\_length* is zero.
- header\_length*   Specifies the number of bytes in the *header\_data*. It must be set to zero to indicate that header data is not to be sent.
- file\_descriptor*   File descriptor for a file that has been opened for read. This is the descriptor for the file that contains the data to be transmitted.
- file\_size*        The size, in bytes, of the file associated with *file\_descriptor*. This field is filled in by the system.
- file\_offset*       Specifies the byte offset into the file from which to start sending data.

<i>file_bytes</i>	Specifies the number of bytes from the file to be transmitted. Setting <i>file_bytes</i> to -1 will transmit the entire file from the <i>offset</i> . In this case the system will replace the -1 with (actual file size - <i>file_offset</i> ). Setting <i>file_bytes</i> to 0 will result in no file data being transmitted and <i>file_descriptor</i> is ignored. If <i>file_descriptor</i> is not a regular file it may be necessary to supply a specific value for <i>file_bytes</i> unless a normal "End-of-File" indication is expected from <i>file_descriptor</i> during this operation or you simply want the operation to run forever transferring bytes as they arrive.
<i>trailer_data</i>	Pointer to a buffer that contains trailer data which is to be sent after the file data.
<i>trailer_length</i>	Specifies the number of bytes in the <i>trailer_data</i> .
<i>bytes_sent</i>	Number of bytes that were sent in this call to <code>send_file()</code> . If it takes multiple calls to <code>send_file()</code> to send all the data (due to signal handling) then this field contains the value for the last call to <code>send_file()</code> , it is not a running total. This field is set by the system.

### Usage Notes

The `send_file()` function attempts to write *header\_length* bytes from the buffer pointed to by *header\_data*, followed by *file\_bytes* from the file associated with *file\_descriptor*, followed by *trailer\_length* bytes from the buffer pointed to by *trailer\_data*, over the connection associated with the socket pointed to by *socket\_ptr*.

As data is sent, the system will update variables in the *sf\_parms* structure so that if the `send_file()` function is interrupted by a signal, the application simply needs to reissue `send_file()`.

If the application sets *file\_offset* > the actual file size, or *file\_bytes* > (the actual file size - *file\_offset*), the return value will be -1 with an *EINVAL* error.

SF\_CLOSE and SF\_REUSE will only be effective after all the data has been sent successfully.

If *options* = SF\_REUSE, and socket reuse is not supported, the system will close the socket and set the socket pointed to by *socket\_ptr* to -1. See "Application Usage" for details.

### Application Usage

`send_file()` is designed to work with `accept_and_recv()` to provide an efficient file transfer capability for a connection oriented server with short connection times and high connection rates.

On the first call to `accept_and_recv()`, it is recommended that the application set the socket pointed to by *accept\_socket* to -1. This will cause the system to assign the accepting socket. On the call to `send_file()`, if the application requests socket reuse (*options* = SF\_REUSE) and the system does not support it, the system will close the socket pointed to by *socket\_ptr* and will set the socket pointed to by *socket\_ptr* to -1. The application then passes this value onto the next call to `accept_and_recv()` (by setting *accept\_socket* = *\*socket\_ptr*).

To take full advantage of the performance improvements offered by the `accept_and_recv()` and `send_file()` functions, a process/thread model different from the one where a parent accepts in a loop and spins off child process threads is needed. The parent/process thread is eliminated. Multiple worker processes/threads are created, and each worker process/thread then executes the `accept_and_recv()` and `send_file()` functions in a loop. The performance benefits of `accept_and_recv()` and `send_file()` include fewer buffer copies, recycled sockets, and optimal scheduling.

## Returned Value

The return values from `send_file()` are:

Returns 0 if the request is successful.

Returns -1 if the request is unsuccessful. Check *errno* for more information .

Returns 1 if the request was interrupted by a signal, or because a nonblocking descriptor would have blocked, while sending data. Since the *sf\_parms* structure is updated by the system to account for the data that has been sent you can continue the operation from where it was interrupted by recalling `send_file()` without changing the *sf\_parms* structure.

## Restrictions

If `O_NONBLOCK` is set on the socket file descriptor, the function may return a -1 with *errno* set to `EWOULDBLOCK` or `EAGAIN`, or it may complete before all the data is sent. If `O_NONBLOCK` is not set, `send_file()` blocks until the requested data can be sent.

## Errors

If unsuccessful, `send_file()` returns -1 and sets *errno* to one of the following:

<code>EACCESS</code>	The calling process does not have the appropriate privileges.
<code>EBADF</code>	One of two errors occurred: <ol style="list-style-type: none"> <li>1. The socket pointed to by <i>socket_ptr</i> is not a valid descriptor.</li> <li>2. <i>file_descriptor</i> is not a valid descriptor.</li> </ol>
<code>ECONNABORTED</code>	A connection has been aborted.
<code>ECONNRESET</code>	A connection has been forcibly closed by a peer.
<code>EFAULT</code>	The data buffer pointed to by <i>socket_ptr</i> , <i>file_size</i> , <i>header_data</i> , or <i>trailer_data</i> was not valid.
<code>EINTR</code>	The <code>send_file()</code> function was interrupted by a signal that was caught before any data was sent.
<code>EINVAL</code>	The value specified by <i>options</i> is not valid.
<code>EIO</code>	An I/O error occurred.
<code>ENETDOWN</code>	The local interface to reach the destination is unknown.
<code>EMSGSIZE</code>	The message is too large to be sent all at once, as the socket requires.

ENETUNREACH

No route to the destination is present.

ENOBUFS

No buffer space is available.

ENOMEM

There was insufficient memory available to complete the operation.

ENOSR

There were insufficient STREAMS resources available for the operation to complete.

ENOTCONN

The socket is not connected.

ENOSYS

This function is not supported in the current environment.

ENOTSOCK

The file descriptor pointed to by the *socket\_ptr* argument does not refer to a socket.

EPIPE

The socket is shutdown for writing, or the socket is in connection mode and is no longer connected.

EWOULDBLOCK

A descriptor is marked nonblocking and the operation could not be performed without blocking.

### Related Information

- “socket() — Create a Socket” on page 1371
- “accept\_and\_recv() — Accept Connection and Receive First Message” on page 78
- “read() — Read From a File or Socket” on page 1080
- “send() — Send Data on a Socket” on page 1178

# sendmsg() — Send Messages on a Socket

## Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

## Format

### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

ssize_t sendmsg(int socket, struct msghdr *message, int flags);
```

### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/socket.h>

int sendmsg(int socket, struct msghdr *message, int flags);
```

## General Description

The sendmsg() call sends messages on a socket with descriptor *socket* passed in an array of message headers.

Parameter	Description
-----------	-------------

<i>socket</i>	The socket descriptor.		
<i>msg</i>	An array of message headers from which messages are sent.		
<i>flags</i>	<p>The <i>flags</i> parameter is set by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator ( ) must be used to separate them.</p> <table><tr><td>MSG_OOB</td><td><p>Sends out-of-band data on the socket. Only SOCK_STREAM sockets support out-of-band data. The out-of-band data is a single byte.</p><p>Before out-of-band data can be sent between two programs, there must be some coordination of effort. If the data is intended to not be read inline, the recipient of the out-of-band data must specify the recipient of the SIGURG signal that is generated when the out-of-band data is sent. If no recipient is set, no signal is sent. The recipient is set up by using F_SETOWN operand of the fcntl command, specifying either a pid or gid. For more information on this operand, refer to the fcntl command.</p><p>The recipient of the data determines whether to receive out-of-band data inline or not inline by the setting of the SO_OOBINLINE option of setsockopt(). For more information on receiving out-of-band data, refer to the</p></td></tr></table>	MSG_OOB	<p>Sends out-of-band data on the socket. Only SOCK_STREAM sockets support out-of-band data. The out-of-band data is a single byte.</p> <p>Before out-of-band data can be sent between two programs, there must be some coordination of effort. If the data is intended to not be read inline, the recipient of the out-of-band data must specify the recipient of the SIGURG signal that is generated when the out-of-band data is sent. If no recipient is set, no signal is sent. The recipient is set up by using F_SETOWN operand of the fcntl command, specifying either a pid or gid. For more information on this operand, refer to the fcntl command.</p> <p>The recipient of the data determines whether to receive out-of-band data inline or not inline by the setting of the SO_OOBINLINE option of setsockopt(). For more information on receiving out-of-band data, refer to the</p>
MSG_OOB	<p>Sends out-of-band data on the socket. Only SOCK_STREAM sockets support out-of-band data. The out-of-band data is a single byte.</p> <p>Before out-of-band data can be sent between two programs, there must be some coordination of effort. If the data is intended to not be read inline, the recipient of the out-of-band data must specify the recipient of the SIGURG signal that is generated when the out-of-band data is sent. If no recipient is set, no signal is sent. The recipient is set up by using F_SETOWN operand of the fcntl command, specifying either a pid or gid. For more information on this operand, refer to the fcntl command.</p> <p>The recipient of the data determines whether to receive out-of-band data inline or not inline by the setting of the SO_OOBINLINE option of setsockopt(). For more information on receiving out-of-band data, refer to the</p>		

setsockopt(), recv(), recvfrom() and recvmsg() commands.

**MSG\_DONTROUTE** The SO\_DONTROUTE option is turned on for the duration of the operation. This is usually used only by diagnostic or routing programs.

A message header is defined by the **msghdr** structure, which can be found in the **sys/socket.h** include file and contains the following elements:

Element	Description
<i>msg_iov</i>	An array of <i>iovec</i> buffers containing the message.
<i>msg_iovlen</i>	The number of elements in the <i>msg_iov</i> array.
<i>msg_name</i>	The optional pointer to the buffer containing the recipient's address.
<i>msg_namelen</i>	The size of the address buffer.
<i>caddr_t msg_accrighs</i>	Access rights sent/received (ignored if specified by the user). This field is ignored by OS/390 UNIX services.
<i>int msg_accrighslen</i>	Length of access rights data (ignored if specified by the user). This field is ignored by OS/390 UNIX services.
<i>msg_control</i>	Ancillary data, see below.
<i>msg_controllen</i>	Ancillary data buffer length.
<i>msg_flags</i>	Flags on received message.

Ancillary data consists of a sequence of pairs, each consisting of a **cmsghdr** structure followed by a data array. The data array contains the ancillary data message, and the **cmsghdr** structure contains descriptive information that allows an application to correctly parse the data.

The **sys/socket.h** header file defines the **cmsghdr** structure that includes at least the following members:

Element	Description
<i>cmsg_len</i>	Data byte count, including header.
<i>cmsg_level</i>	Originating protocol.
<i>cmsg_type</i>	Protocol-specific type.

The **sys/socket.h** header file defines the following macro for use as the **cmsg\_type** value when **cmsg\_level** is SOL\_SOCKET:

**SCM\_RIGHTS** Indicates that the data array contains the access rights to be sent or received. This option is valid only for the AF\_UNIX domain.

The **sys/socket.h** header file defines the following macros to gain access to the data arrays in the ancillary data associated with a message header:

**CMSG\_DATA(*cmsg*)** If the argument is a pointer to a **cmsghdr** structure, this macro returns an unsigned character pointer to the data array associated with the **cmsghdr** structure.



**MSG\_NXTHDR(*mhdr, cmsg*)**

If the first argument is a pointer to a **msghdr** structure and the second argument is a pointer to a **cmsghdr** structure in the ancillary data, pointed to by the **msg\_control** field of that **msghdr** structure, this macro returns a pointer to the next **cmsghdr** structure, or a null pointer if this structure is the last **cmsghdr** in the ancillary data.

**MSG\_FIRSTHDR(*mhdr*)**

If the argument is a pointer to a **msghdr** structure, this macro returns a pointer to the first **cmsghdr** structure in the ancillary data associated with this **msghdr** structure, or a null pointer if there is no ancillary data associated with the **msghdr** structure.

The `sendmsg()` call applies to sockets regardless of whether they are in the connected state.

This call returns the length of the data sent. If there is not enough available buffer space to hold the socket data to be transmitted, and the socket is in blocking mode, `sendmsg()` blocks the caller until additional buffer space becomes available. If the socket is in nonblocking mode, `sendmsg()` returns a `-1` and sets the error code to `EWouldBlock`. See “`fcntl()` — Control Open File Descriptors” on page 350 or “`ioctl()` — Control Device” on page 672 for a description of how to set nonblocking mode.

For datagram sockets, this call sends the entire datagram, provided that the datagram fits into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application wishes to send 1000 bytes, each call to this function can send 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been sent.

**Special Behavior for C++**

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

**Returned Value**

If successful, the length of the message in bytes is returned. The value `-1` indicates an error. The value of the error code indicates the specific error.

A value of 0 or greater indicates the number of bytes sent, however, this does not assure that data delivery was complete. A connection can be dropped by a peer socket and a `SIGPIPE` signal generated at a later time if data delivery is not complete.

Error Code	Description
<code>EBADF</code>	<i>socket</i> is not a valid socket descriptor.
<code>EAFNOSUPPORT</code>	The address family is not supported (it is not <code>AF_UNIX</code> or <code>AF_INET</code> ).
<code>ECONNREFUSED</code>	The attempt to connect was rejected.
<code>ECONNRESET</code>	A connection was forcibly closed by a peer.

EFAULT	Using <i>msg</i> would result in an attempt to access storage outside the caller's address space.
EINTR	A signal interrupted sendmsg() before any data was transmitted.
EINVAL	<i>msg_namelen</i> is not the size of a valid address for the specified address family.
EIO	There has been a network or transport failure.
EMSGSIZE	The message was too big to be sent as a single datagram. The default is <i>large-envelope-size</i> . (Envelopes are used to hold datagrams and fragments during TCP/IP processing. Large envelopes hold UDP datagrams greater than 2KB while they are processed for output, and when they are waiting for an application program to receive them on input.)
ENOBUFS	Buffer space is not available to send the message.
ENOTCONN	The socket is not connected.
ENOTSOCK	The descriptor is for a file, not for a socket.
EOPNOTSUPP	The <i>socket</i> argument is associated with a socket that does not support one or more of the values set in <i>flags</i> .
EPIPE	For a connected stream socket the connection to the peer socket has been lost. A SIGPIPE signal is sent to the calling process.
EWouldBlock	<i>socket</i> is in nonblocking mode and no data buffers are available.

The following are for AF\_UNIX only:

#### Error Code Description

EACCES	Search permission is denied for a component of the path prefix, or write access to the named socket is denied.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating the pathname in the socket address.
ENAMETOOLONG	A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX characters.
ENOENT	A component of the pathname does not name an existing file or the pathname is an empty string.
ENOTDIR	A component of the path prefix of the pathname in the socket address is not a directory.

#### Related Information

- “connect() — Connect a Socket” on page 214
- “fcntl() — Control Open File Descriptors” on page 350
- “getsockopt() — Get the Options Associated with a Socket” on page 606
- “ioctl() — Control Device” on page 672
- “read() — Read From a File or Socket” on page 1080

- “readv() — Read Data on a File or Socket and Store in a Set of Buffers” on page 1093
- “recv() — Receive Data on a Socket” on page 1103
- “recvfrom() — Receive Messages on a Socket” on page 1106
- “recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers” on page 1110
- “select() — Monitor Activity on Files/Sockets and Message Queues” on page 1159
- “selectex() — Monitor Activity on Files/Sockets and Message Queues” on page 1166
- “sendmsg() — Send Messages on a Socket” on page 1185
- “sendto() — Send Data on a Socket” on page 1190
- “setsockopt() — Set Options Associated with a Socket” on page 1270
- “socket() — Create a Socket” on page 1371
- “write() — Write Data on a File or Socket” on page 1780
- “writev() — Write Data on a File or Socket Socket from an Array” on page 1785

# sendto() — Send Data on a Socket

## Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

## Format

### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

ssize_t sendto(int socket, const void *buffer, size_t length, int flags,
               const struct sockaddr *address,
               size_t address_length);
```

### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/socket.h>

int sendto(int socket, char *buffer, int length, int flags,
           struct sockaddr *address, int address_len);
```

## General Description

The sendto() call sends data on the socket with descriptor *socket*. The sendto() call applies to either connected or unconnected sockets.

Parameter	Description
-----------	-------------

<i>socket</i>	The socket descriptor.
<i>message</i>	The pointer to the buffer containing the message to transmit.
<i>length</i>	The length of the message in the buffer pointed to by the <i>msg</i> parameter.
<i>flags</i>	<p>A parameter that can be set to 0 or MSG_DONTROUTE. Setting this parameter is supported only for sockets in the AF_INET domain. Setting these flags is not supported in the AF_UNIX domain.</p> <p><b>MSG_OOB</b> Sends out-of-band data on the socket. Only SOCK_STREAM sockets support out-of-band data. The out-of-band data is a single byte.</p> <p>Before out-of-band data can be sent between two programs, there must be some coordination of effort. If the data is intended to not be read inline, the recipient of the out-of-band data must specify the recipient of the SIGURG signal that is generated when the out-of-band data is sent. If no recipient is set, no signal is sent. The recipient is set up by using F_SETOWN operand of the fcntl command, specifying either a pid or gid. For more information on this operand, refer to the fcntl command.</p> <p>The recipient of the data determines whether to receive out-of-band data inline or not inline by the setting of the SO_OOBINLINE option of setsockopt(). For more infor-</p>

mation on receiving out-of-band data, refer to the `setsockopt()`, `recv()`, `recvfrom()` and `recvmsg()` commands.

#### MSG\_DONTROUTE

The `SO_DONTROUTE` option is turned on for the duration of the operation. This is usually used only by diagnostic or routing programs.

*to* The address of the target.

*tolength* The size of the address pointed to by *to*.

If there is not enough available buffer space to hold the socket data to be transmitted, and the socket is in blocking mode, `sendto()` blocks the caller until additional buffer space becomes available. If the socket is in nonblocking mode, `sendto()` returns a `-1` and sets the error code to `EWOULDBLOCK`. See “`fcntl()` — Control Open File Descriptors” on page 350 or “`ioctl()` — Control Device” on page 672 for a description of how to set nonblocking mode.

For datagram sockets, this call sends the entire datagram, provided that the datagram fits into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application wishes to send 1000 bytes, each call to this function can send 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, applications using stream sockets should place this call in a loop, calling this function until all data has been sent.

#### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

#### Returned Value

If successful, the number of characters sent is returned. The value `-1` indicates an error. The value of the error code indicates the specific error. No indication of failure to deliver is implied in the return value of this call when used with datagram sockets.

A value of 0 or greater indicates the number of bytes sent, however, this does not assure that data delivery was complete. A connection can be dropped by a peer socket and a `SIGPIPE` signal generated at a later time if data delivery is not complete.

#### Error Code Description

##### EAFNOSUPPORT

The address family is not supported (it is not `AF_UNIX` or `AF_INET`).

**EBADF** *socket* is not a valid socket descriptor.

##### ECONNREFUSED

The attempt to connect was rejected.

##### ECONNRESET

A connection was forcibly closed by a peer.

**EFAULT** Using the *msg* and *length* parameters would result in an attempt to access storage outside the caller's address space.

EINTR	A signal interrupted sendto() before any data was transmitted.
EINVAL	<i>tolength</i> is not the size of a valid address for the specified address family.
EIO	There has been a network or transport failure.
EMSGSIZE	The message was too big to be sent as a single datagram. The default is <i>large-envelope-size</i> . (Envelopes are used to hold datagrams and fragments during TCP/IP processing. Large envelopes hold UDP datagrams greater than 2KB while they are processed for output, and when they are waiting for an application program to receive them on input.)
ENOBUFS	Buffer space is not available to send the message.
ENOTCONN	The socket is not connected.
ENOTSOCK	The descriptor is for a file, not for a socket.
EOPNOTSUPP	The <i>socket</i> argument is associated with a socket that does not support one or more of the values set in <i>flags</i> .
EPIPE	For a connected stream socket the connection to the peer socket has been lost. A SIGPIPE signal is sent to the calling process.
EPROTOTYPE	The protocol is the wrong type for this socket. A SIGPIPE signal is sent to the calling process.
EWouldBlock	<i>socket</i> is in nonblocking mode and no data buffers are available.

The following are for AF\_UNIX only:

**Error Code Description**

EACCES	Search permission is denied for a component of the path prefix, or write access to the named socket is denied.
EIO	An I/O error occurred while reading from or writing to the file system.
ELOOP	Too many symbolic links were encountered in translating the pathname in the socket address.
ENAMETOOLONG	A component of a pathname exceeded NAME_MAX characters, or an entire pathname exceeded PATH_MAX characters.
ENOENT	A component of the pathname does not name an existing file or the pathname is an empty string.
ENOTDIR	A component of the path prefix of the pathname in the socket address is not a directory.

**Related Information**

- “read() — Read From a File or Socket” on page 1080
- “readv() — Read Data on a File or Socket and Store in a Set of Buffers” on page 1093
- “recv() — Receive Data on a Socket” on page 1103
- “recvfrom() — Receive Messages on a Socket” on page 1106
- “recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers” on page 1110
- “select() — Monitor Activity on Files/Sockets and Message Queues” on page 1159
- “send() — Send Data on a Socket” on page 1178
- “sendmsg() — Send Messages on a Socket” on page 1185
- “socket() — Create a Socket” on page 1371
- “write() — Write Data on a File or Socket” on page 1780
- “writev() — Write Data on a File or Socket Socket from an Array” on page 1785

## \_\_server\_classify() — Set Classify Area Field

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R3

### Format

```
#include <sys/server.h>
int __server_classify(server_classify_t handle,
                      server_classify_field_t field,
                      const char *value);
```

### General Description

The `__server_classify()` function sets fields in a classify data area. The 'classify data area' is created and initialized by invoking the `__server_classify_create()` function. This 'classify data area' is subsequently used with the `__server_pwu()` function to interface with Work Load Manager (WLM).

The 'handle' argument is a 'classify data area' created on a previous invocation of the `__server_classify_create()` function.

The 'handle' argument must be one of the following values:

#### `_SERVER_CLASSIFY_ACCTINFO`

Set the accounting information. When specified, value contains a null terminated character string of up to 143 characters containing the account information for the work unit to be created.

#### `_SERVER_CLASSIFY_COLLECTION`

Set the customer defined name for a group of associated packages. When specified, value contains a null terminated character string of up to 18 characters containing the collection name associated with the work unit to be created.

#### `_SERVER_CLASSIFY_CONNECTION`

Set the name associated with the environment creating the work unit. When specified, value contains a null terminated character string of up to 8 characters containing the connection name associated with the environment creating the work unit.

#### `_SERVER_CLASSIFY_CONNTKN`

Set the connection token that was returned on a call to `__ConnectWorkMgr()` or `__ConnectServerMgr()`. When specified, value contains a integer value representing the connection token returned on a call to `__ConnectWorkMgr()` or `__ConnectServerMgr()`.

#### `_SERVER_CLASSIFY_CORRELATION`

Set the name associated with the user/program creating the work unit. When specified, value contains a null terminated character string of up to 12 characters that contains the name associated with the user/program creating the work unit.

#### `_SERVER_CLASSIFY_LUNAME`

Set the local LU name associated with the requestor. When specified, value contains a null terminated character string of up to 8 characters containing the local LU name associated with the requestor.



**\_SERVER\_CLASSIFY\_NETID**

Set the network ID associated with the requestor. When specified, value contains a null terminated character string of up to 8 characters containing the network ID associated with the requestor.

**\_SERVER\_CLASSIFY\_PACKAGE**

Set the package name for a set of associated SQL statements. When specified, value contains a null terminated character string of up to 8 characters containing the package name associated with the work unit to be created.

**\_SERVER\_CLASSIFY\_PERFORM**

Set the performance group number (PGN) associated with the work unit. When specified, value contains a null terminated character string of up to 8 characters containing the PGN associated with the work unit to be created.

**\_SERVER\_CLASSIFY\_PLAN**

Set the access plan name for a set of associated SQL statements. When specified, value contains a null terminated character string of up to 8 characters containing the access plan name associated with the work unit to be created.

**\_SERVER\_CLASSIFY\_PRCNAME**

Set the DB2 Stored SQL Procedure name associated with the work unit. When specified, value contains a null terminated character string of up to 18 characters containing the DB2 Stored SQL Procedure name associated with the work unit to be created.

**\_SERVER\_CLASSIFY\_PRIORITY**

Set the priority associated with the work unit to be created. When specified, value contains a integer value representing the priority of the work unit to be created.

**\_SERVER\_CLASSIFY\_RPTCLSNM@**

Set the pointer to an 8 character buffer to receive the output report class name for the work unit to be created. When specified, value contains the pointer to an 8 character buffer to receive the output report class name for the work unit to be created.

**\_SERVER\_CLASSIFY\_SERVCLS@**

Set the pointer to an integer field to receive the output service class for the work unit to be created. When specified, value contains the pointer to a integer field to receive the output service class for the work unit to be created.

**\_SERVER\_CLASSIFY\_SERVCLSNM@**

Set the pointer to an 8 character buffer to receive the output service class name for the work unit to be created. When specified, value contains the pointer to an 8 character buffer to receive the output service class name for the work unit to be created.

**\_SERVER\_CLASSIFY\_SOURCELU**

Set the source LU name associated with the requestor. When specified, value contains a null terminated character string of up to 17 characters containing the source LU name associated with the requestor.

**\_SERVER\_CLASSIFY\_SUBSYSTEM\_PARM**

Set the transaction subsystem parameter. When specified, *value* contains a null terminated character string of up to 255 characters containing the subsystem parameter being used for the `__server_pwu()` call.

**\_SERVER\_CLASSIFY\_TRANSACTION\_CLASS**

Set the transaction class. When specified, *value* contains a null terminated character string of up to 8 characters containing the name of the transaction class for the `__server_pwu()` call.

**\_SERVER\_CLASSIFY\_TRANSACTION\_NAME**

Set the transaction name. When specified, *value* contains a null terminated character string of up to 8 characters containing the name of the transaction for the `__server_pwu()` call.

**\_SERVER\_CLASSIFY\_USERID**

Set the userid. When specified, *value* contains a null terminated character string of up to 8 characters containing the name of the user for the `__server_pwu()` call.

The 'value' argument is the value that the specified 'classify data area' field is to be set to. (For valid values, refer to *OS/390 MVS Programming: Workload Management Services*, GC28-1773.)

The classify area is specific to the calling thread. The `__server_classify()` function call must be done on the same thread of execution as the `__server_classify_create()`. Use of the classify area by another thread can lead to unpredictable results.

**Returned Value**

If successful, `__server_classify()` returns a value of zero.

If unsuccessful, `__server_classify()` returns -1, and returns the error value in `errno`. The following are the possible values of `errno`:

- |        |   |
|--------|---|
| EINVAL | The classify field symbolic is not valid.                         |
| E2BIG  | The character string specified for a classify field is too large. |

**Related Information**

- “sys/server.h” on page 48
- “`__server_classify_create()` — Create a Classify Area” on page 1197
- “`__server_classify_destroy()` — Delete a Classify Area” on page 1198
- “`__server_classify_reset()` — Reset a Classify Area to an Initial State” on page 1199
- “`__server_init()` — Initialize Server” on page 1200
- “`__server_pwu()` — Process Server Work Unit” on page 1203

## \_\_server\_classify\_create() — Create a Classify Area

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R3

### Format

```
#include <sys/server.h>
server_classify_t __server_classify_create(server_classify_t handle,
                                           server_classify_field_t field,
                                           const char *value);
```

### General Description

The `__server_classify_create()` function creates a classify data area to be used on a subsequent `__server_pwu()` or `CreateWorkUnit()` call. The resulting classify data area can be filled in by calls to the `__server_classify()` function. The information in the classify area is used to establish the Transaction class, Transaction Name, userid, and subsystem parameters for the `__server_pwu()` call or to establish the classification rules for the `CreateWorkUnit()` call.

The resulting classify area is specific to the calling thread. Use of the classify area by another thread can lead to unpredictable results.

### Returned Value

If successful, `__server_classify_create()` returns a *classify\_t* which is a handle to the classify area.

If unsuccessful, `__server_classify_create()` returns -1, and returns the error value in `errno`. The following are the possible values of `errno`:

ENOMEM    Not enough storage is available.

### Related Information

The classify data area created by this function can be used without serialization only by the creating thread. In addition, storage for this structure is automatically freed at thread termination.

- “`sys/server.h`” on page 48
- “`__server_classify()` — Set Classify Area Field” on page 1194
- “`__server_classify_destroy()` — Delete a Classify Area” on page 1198
- “`__server_classify_reset()` — Reset a Classify Area to an Initial State” on page 1199
- “`__server_init()` — Initialize Server” on page 1200
- “`__server_pwu()` — Process Server Work Unit” on page 1203

## `__server_classify_destroy()` — Delete a Classify Area

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R3

### Format

```
#include <sys/server.h>
void __server_classify_destroy(server_classify_t area);
```

### General Description

The `__server_classify_destroy()` function deletes a classify data area previously created by a `__server_classify()` call. The *area* parameter specifies the classify area to be deleted. Storage for the classify area is freed. This function must be executed by the same thread that created the classify area.

### Returned Value

`__server_classify_destroy()` does not have a return value.

### Related Information

- “`sys/server.h`” on page 48
- “`__server_classify()` — Set Classify Area Field” on page 1194
- “`__server_classify_create()` — Create a Classify Area” on page 1197
- “`__server_classify_reset()` — Reset a Classify Area to an Initial State” on page 1199
- “`__server_init()` — Initialize Server” on page 1200
- “`__server_pwu()` — Process Server Work Unit” on page 1203

## \_\_server\_classify\_reset() — Reset a Classify Area to an Initial State

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R3

### Format

```
#include <sys/server.h>
void __server_classify_reset(server_classify_t area);
```

### General Description

The `__server_classify_reset()` function resets a classify data area to its initial state. This is equivalent to destroying the classify area and creating another, and is intended to be a higher performance path for applications which must repeatedly change parameters in a classify area. The *area* parameter specifies the handle of the classify area to be reset, and was previously obtained by a `__server_classify()` call. This function must be executed by the same thread that created the classify area.

### Returned Value

`__server_classify_reset()` does not have a return value.

### Related Information

- “sys/server.h” on page 48
- “\_\_server\_classify() — Set Classify Area Field” on page 1194
- “\_\_server\_classify\_create() — Create a Classify Area” on page 1197
- “\_\_server\_classify\_destroy() — Delete a Classify Area” on page 1198
- “\_\_server\_init() — Initialize Server” on page 1200
- “\_\_server\_pwu() — Process Server Work Unit” on page 1203

## \_\_server\_init() — Initialize Server

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R3

### Format

```
#include <sys/server.h>
```

```
int __server_init(int *managertype,
                  const char *subsysname,
                  const char *applenv,
                  int parallelism);
```

### General Description

The `__server_init()` function provides the ability for a server address space to connect to Work Load Manager (WLM) for the purpose of queueing and servicing work requests.

The parameters supported are:

#### *\*managertype*

Points to one or more of the following values that indicate the type of WLM manager the caller is requesting to become. The following are the supported values:

##### SRV\_WORKMGR

Indicates that WLM work management services be made available to the calling address space. This value can be combined with the SRV\_QUEUEMGR and SRV\_SERVERMGR values.

##### SRV\_QUEUEMGR

Indicates that WLM queue management services be made available to the calling address space. This value can be combined with the SRV\_WORKMGR and SRV\_SERVERMGR values.

##### SRV\_SERVERMGR

Indicates that WLM server management services be made available to the calling address space. This value can be combined with the SRV\_WORKMGR and SRV\_QUEUEMGR values.

*\*subsysname* Points to a null terminated character string containing the generic subsystem type (CICS, IMS, WEB, etc.). When SRV\_WORKMGR is specified for the *managertype* parameter this is the primary category under which WLM classification rules are grouped. The character string can be up to 4 bytes in length.

#### *\*subsysname*

Points to a null terminated character string containing the subsystem name used for classifying work requests when SRV\_WORKMGR is specified for the *managertype* parameter. When SRV\_SERVERMGR is specified for the *managertype* parameter the subsystem name

should match the subsystem name specified on the corresponding call to \_\_server\_init() for a work manager (SRV\_WORKMGR *managertype*). The character string can be up to 8 bytes in length. When SRV\_QUEUEMGR is specified for the *managertype* parameter the combination of the *subsys*type and *subsys*name parameter values must be unique to a single MVS system.

- \*applenv* Points to a null terminated character string that contains the name of the application environment under which work requests are served. The character string can be up to 32 bytes in length. This parameter is only valid when SRV\_SERVERMGR is specified for the *managertype* parameter. It should be NULL for all other *managertype* values.
- paralleu* Specifies the maximum number of tasks within the address space which will be created to process concurrent work requests. This parameter is only valid when SRV\_SERVERMGR is specified for the *managertype* parameter. It is ignored for all other *managertype* values.

A successful call to \_\_server\_init() results in the calling address space being connected to WLM for the WLM management services requested. Additionally, for a successful server manager WLM connection call (SRV\_SERVERMGR *managertype*), the calling process is made a child of, and is placed in the session and process group of the corresponding work manager. The corresponding work manager is the process that called server\_init() for the *managertype* combination SRV\_WORKMGR+SRV\_QUEUEMGR with the same *subsys*type and *subsys*name values specified as the server manager process. This parent child relationship allows the server manager and the work manager to use signals to communicate with each other.

## Returned Value

The \_\_server\_init() function returns 0 if successful. Otherwise, it returns -1 and sets errno to indicate the error. The following are the possible values of errno:

- EINVAL The *managertype* parameter contains a value that is not correct.
- EMVSWLMERROR  
A WLM service failed. Use \_\_errno2() to obtain the WLM service reason code for the failure.
- EPERM The calling thread's address space is not permitted to the BPX.WLMSEVER Facility class. The caller's address space must be permitted to the BPX.WLMSEVER Facility class, if the BPX.WLMSEVER is defined. If BPX.WLMSEVER is not defined, the calling process is not defined as a superuser (UID=0).
- EMVSSAF2ERR  
An error occurred in the security product.

## Related Information

- “sys/server.h” on page 48
- “\_\_server\_classify() — Set Classify Area Field” on page 1194
- “\_\_server\_classify\_create() — Create a Classify Area” on page 1197
- “\_\_server\_classify\_destroy() — Delete a Classify Area” on page 1198
- “\_\_server\_classify\_reset() — Reset a Classify Area to an Initial State” on page 1199

- “\_\_server\_pwu() — Process Server Work Unit” on page 1203



## \_\_server\_pwu() — Process Server Work Unit

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R3

### Format

```
#include <sys/server.h>
```

```
int __server_pwu(int fcncode,
    const char      *transclass,
    const char      *applenv,
    server_classify_t classify,
    int             *apldataalen,
    void            **apldata,
    struct __srv_fd_list **fdlstruc);
```

### General Description

The `__server_pwu()` function provides a general purpose interface for managing and processing work via Work Load Manager (WLM) services. The capabilities this service provides include the ability to put work requests onto the WLM work queues, obtain work from the WLM work queues, transfer work to other WLM work servers, end units of work, delete WLM enclaves and refresh WLM work servers.

The parameters supported are:

**fcncode**      Contains one or more of the following values that indicate the function that is requested:

#### SRV\_PUT\_NEWWRK

Indicates that a new work request be put onto the work queue for an application environment server identified by the *applenv* parameter as part of a newly created WLM enclave. This value cannot be combined with any other *fcncode* value.

#### SRV\_PUT\_SUBWRK

Indicates that a new work request be put onto the work queue for an application environment server identified by the *applenv* parameter as part of the WLM enclave associated with the calling thread. This value can be combined with the *SRV\_END\_WRK* *fcncode* value.

#### SRV\_TRANSFER\_WRK

Indicates that the last work request obtained by the calling thread be transferred to the work queue of the target application environment server. As part of the transfer, the calling thread is disassociated from its WLM enclave. This value cannot be combined with any other *fcncode* value.

#### SRV\_GET\_WRK

Indicates that a new work request be obtained from the WLM work queue for the calling application environment server. The *SRV\_GET\_WRK* *fcncode* also results in the

calling thread being associated with the WLM enclave of the work request. If the calling thread is already associated with a WLM enclave due to a prior call to `__server_pwu()` for `SRV_GET_WRK`, it is disassociated from the prior WLM enclave, as well as associated with the obtained work request. When the calling thread goes thru task termination or when its process is terminated the work request is ended and the associated WLM enclave is deleted if it is owned by the terminating task or process. The `SRV_GET_WRK` caller owns the enclave, if the work was queued using the `SRV_PUT_NEWWRK` or `SRV_TRANSFER_WRK` functions. If the caller is a thread created via `pthread_create` (`pthread`), the thread task owns the enclave. If the caller is not a `pthread`, the process owns the enclave. This value can be combined with the `SRV_END_WRK` and `SRV_DEL_ENC` *fcncode* values.

#### `SRV_REFRESH_WRK`

Indicates that the servers associated with the application environments managed by the calling work and queue manager are to be refreshed. This will cause all servers to complete existing work requests and then terminate. New servers will then be started to process new work. This value cannot be combined with any other *fcncode* value.

#### `SRV_END_WRK`

Indicates that the calling thread is to be disassociated from its WLM enclave. This value can be combined with the `SRV_DEL_ENC`, `SRV_PUT_SUBWRK` and `SRV_GET_WRK` *fcncode* values.

#### `SRV_DEL_ENC`

Indicates that the WLM enclave associated with the calling thread is to be deleted. This value can be combined with the `SRV_GET_WRK` and `SRV_END_WRK` *fcncode* values.

#### `SRV_DISCONNECT`

Indicates that the calling server's connection to WLM is to be severed. Once a server is disconnected from WLM, it can no longer use this service to process more requests for the application environment it had been connected to (via a call to the `server_init()` function). If a `SRV_DISCONNECT` is performed by a work and queue manager, all related server managers implicitly lose their connection to WLM. This also results in the related server managers losing their ability to process more requests via this service.

#### `SRV_DISCONNECT_COND`

Indicates that the calling server's connection to WLM is to be severed only if the caller has no more WLM enclaves that it is still managing. A work and queue manager is still managing an enclave if it has yet to be

served by a server manager. Once a server is disconnected from WLM, it can no longer use this service to process more requests for the application environment it had been connected to (via a call to the `server_init()` function). If a `SRV_DISCONNECT_COND` is performed by a work and queue manager, all related server managers implicitly lose their connection to WLM. This also results in the related server managers losing their ability to process more requests via this service.

- `*transclass` Points to a null terminated character string that represents the name of the transaction class to be associated with the work request. This parameter is only valid when the `SRV_PUT_NEWWRK fncode` parameter value is specified. It should be NULL for the other `fncode` parameter values. The character string can be up to 8 bytes in length.
- `*applenv` Points to a null terminated character string that contains the name of the application environment under which work requests are served. This parameter is valid for the set of `SRV_PUT fncode` values, the `SRV_TRANSFER_WRK fncode` value and the `SRV_REFRESH_WRK fncode` value. It should be NULL for the other `fncode` parameter values. The character string can be up to 32 bytes in length.
- `*classify` Points to a character string that contains the classification information for the work request macro.
- `*apldataen` When one of the `SRV_PUT` or `SRV_TRANSFER fncode` parameter values is specified this is a supplied parameter that points to an integer containing the length of the application data specified by the `**apldata` parameter. When the `SRV_GET_WRK fncode` value is specified, this is an output parameter where the `__server_pwu()` function is to return the length of the application data associated with the obtained work request. `*apldataen` is only valid when one of the `SRV_PUT`, `SRV_GET_WRK` or `SRV_TRANSFER fncode` parameter values is specified, it is ignored otherwise. The maximum length supported for the application data is 10 megabytes.
- `**apldata` When one of the `SRV_PUT` or `SRV_TRANSFER fncode` parameter values is specified this is a supplied parameter that points to the application data string. This application data allows the caller to uniquely identify the specific work the caller is requesting. When the `SRV_GET_WRK fncode` value is specified, this is an output parameter where the `__server_pwu()` function is to return a pointer to the application data associated with the obtained work request. The returned data area will be an identical copy of the data area that was supplied on the corresponding `__server_pwu()` call to put the work request on a WLM work queue. `**apldata` is only valid when one of the `SRV_PUT`, `SRV_GET_WRK` or `SRV_TRANSER fncode` parameter values is specified, it is ignored otherwise.
- `**fdlstruct` When one of the `SRV_PUT` or `SRV_TRANSFER fncode` parameter values is specified the `**fdlstruc` parameter is an input parameter that contains a pointer to a `__srv_fd_list` structure. The `__srv_fd_list` structure contains the following members:

<code>int</code>	<code>fdcount</code>	count of file descriptors
<code>int</code>	<code>flags</code>	flag <code>SRV_FDCLOSE</code>
<code>int</code>	<code>fd(SRV_FDS)</code>	file descriptor list

The supplied `__srv_fd_list` structure contains the count of file descriptors to be propagated, followed by the list of file descriptors that are to be propagated to the process that calls `server_pwu()` to obtain the work request created by the call to this service. If the `SRV_FDCLOSE` flag is turned on in the `flags` field of the `__srv_fd_list` structure, all file descriptors in the list are closed in the calling process. If a null pointer is specified, no file descriptors are propagated. When the *\*\*fdlstruc* parameter is used to propagate file descriptors, the caller must ensure that all of the file descriptors in the list are valid open file descriptors in the caller's process, and are not being closed during the processing of this service. If this is not the case, then this function cannot guarantee the proper propagation of the specified file descriptors. When the `SRV_GET_WRK fncode` parameter value is specified the *\*\*fdlstruc* parameter is an output parameter where the `__server_pwu()` function returns a pointer to the `__srv_fd_list` structure associated with the obtained work request. The returned `__srv_fd_list` structure will contain the count of file descriptors in the returned structure, followed by the list of remapped file descriptor values in the calling process of the file descriptors that were supplied in the `__srv_fd_list` structure on the corresponding `__server_pwu()` call to put the work request on a WLM work queue. The `flags` field in the returned `__srv_fd_list` structure will be null. The *\*\*fdlstruc* parameter is only valid when one of the `SRV_PUT`, `SRV_TRANSFER` or `SRV_GET_WRK fncode` parameter values are specified. It is ignored otherwise. The maximum number of file descriptors supported in the file descriptor list structure is 64.

A successful call to `__server_pwu()` for the `SRV_PUT_NEWWRK fncode` not only creates a work request that is placed onto a WLM work queue, but it also creates a new WLM enclave for that work to run in when the work request is obtained. By comparison, the `SRV_PUT_SUBWRK` and `SRV_TRANSFER_WRK fncodes`, queue work requests that are part of the existing WLM enclave of the calling thread.

A successful call to `__server_pwu()` for the `SRV_GET_WRK fncode` not only results in the caller obtaining a work request from a WLM work queue associated with the caller's application environment, but also results in the calling thread being associated with the WLM enclave associated with the obtained work request.

Usage of the `server_pwu` function requires the calling address space to have successfully issued a call to the `__server_init()` function.

For the `SRV_PUT_NEWWRK` function to run successfully, the caller must have successfully issued a call to the `__server_init()` service for one of the following *managertype* parameter combinations:

- `SRV_WORKMGR + SRV_QUEUEMGR`
- `SRV_WORKMGR + SRV_QUEUEMGR + SRV_SERVERMGR`

For the `SRV_PUT_SUBWRK` and `SRV_TRANSFER_WRK` functions to run successfully, the caller must have successfully issued a call to the `__server_init()` service for one of the following *managertype* parameter combinations:

- SRV\_WORKMGR + SRV\_QUEUEMGR SRV\_SERVERMGR
- SRV\_SERVERMGR

For the SRV\_GET\_WRK, SRV\_END\_WRK and SRV\_DEL\_ENC functions to run successfully, the caller must have successfully issued a call to the \_\_server\_init() service for one of the following *managertype* parameter combinations:

- SRV\_WORKMGR + SRV\_QUEUEMGR SRV\_SERVERMGR
- SRV\_SERVERMGR

For the SRV\_REFRESH\_WRK function to run successfully, the caller must have successfully issued a call to the \_\_server\_init() service for any of the following *managertype* parameter combinations:

- SRV\_WORK\_MGR + SRV\_QUEUE\_MGR
- SRV\_WORK\_MGR + SRV\_QUEUE\_MGR + SRV\_SERVER\_MGR

### Returned Value

The \_\_server\_pwu() function returns 0 if successful. Otherwise, it returns -1 and sets errno to indicate the error. The following are the possible values of errno:

EAGAIN	The requested service could not be performed at the current time. Use __errno2() to obtain the reason code for the failure.
EFAULT	An argument of this service contained an address that was not accessible to the caller.
EINVAL	The <i>managertype</i> parameter contains a value that is not correct.
EMVSERR	A MVS environmental or internal error has occurred. Use __errno2() to obtain the exact reason for the failure.
EMVSWLMERROR	A WLM service failed. Use __errno2() to obtain the WLM service reason code for the failure.

### Related Information

- “sys/server.h” on page 48
- “\_\_server\_classify() — Set Classify Area Field” on page 1194
- “\_\_server\_classify\_create() — Create a Classify Area” on page 1197
- “\_\_server\_classify\_destroy() — Delete a Classify Area” on page 1198
- “\_\_server\_classify\_reset() — Reset a Classify Area to an Initial State” on page 1199
- “\_\_server\_init() — Initialize Server” on page 1200

## setbuf() — Control Buffering

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
void setbuf(FILE *stream, char *buffer);
```

### General Description

Controls buffering for the specified *stream*. The *stream* pointer must refer to an open file, and setbuf() must be the first operation on the stream.

If the *buffer* argument is NULL, the *stream* is unbuffered. If not, the buffering mode will be full buffer and the *buffer* must point to a character array of length at least BUFSIZ, which is the buffer size defined in the stdio.h header file. I/O functions use the *buffer*, which you specify here, for input/output buffering instead of the default system-allocated buffer for the given *stream*. If the buffer does not meet the requirements of the OS/390 C/C++ product, the buffer is not used.

The setvbuf() function is more flexible than setbuf(), because you can specify the type of buffering and size of buffer.

**Attention:** If you use setvbuf() or setbuf() to define your own buffer for a stream, you must ensure that the buffer is available the whole time that the stream associated with the buffer is in use.

For example, if the buffer is an automatic array (block scope) and is associated with the stream *s*, leaving the block causes the storage to be deallocated. I/O operations of stream *s* are prevented from using deallocated storage. Any operation on *s* would fail because the operation would attempt to access the nonexistent storage.

To ensure that the buffer is available throughout the life of a program, make the buffer a variable allocated at file scope. This can be achieved by using an identifier of type *array* declared at file scope, or by allocating storage (with malloc() or calloc()) and assigning the storage address to a pointer declared at file scope.

VSAM file types do not support unbuffered I/O, causing requests for unbuffered I/O to fail.

### Returned Value

There is no returned value. For details about errno values, and about buffers you may have set, see discussions about buffering in the *OS/390 C/C++ Programming Guide*.

## Example

### CBC3BS01

```

/* CBC3BS01
   This example opens the file myfile.dat for writing. It then calls
   the setbuf() function to establish a buffer of length BUFSIZ.
   When string is written to the stream, the buffer buf is used and
   contains the string before it is flushed to the file.
*/
#include <stdio.h>

int main(void)
{
    char buf[BUFSIZ];
    char string[] = "hello world";
    FILE *stream;

    stream = fopen("myfile.dat", "wb,recfm=f");

    setbuf(stream,buf);      /* set up buffer */

    fwrite(string, sizeof(string), 1, stream);

    printf("%s\n",buf);      /* string is found in buf now */

    fclose(stream);          /* buffer is flushed out to myfile.dat */
}

```

## Related Information

- “stdio.h” on page 43
- “fclose() — Close File” on page 348
- “fflush() — Write Buffer to File” on page 385
- “fopen() — Open a File” on page 417
- “setvbuf() — Control Buffering” on page 1283

## setcontext() — Restore User Context

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON)

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ucontext.h>

int setcontext(const ucontext_t *ucp);
```

### General Description

The `setcontext()` function restores the user context pointed to by `ucp`. A successful call to `setcontext()` does not return; program execution resumes at the point specified by the `ucp` argument passed to `setcontext()`. The `ucp` argument should be created either by a prior call to `getcontext()`, or by being passed as an argument to a signal handler. If the `ucp` argument was created with `getcontext()`, program execution continues as if the corresponding call of `getcontext()` had just returned. If the `ucp` argument was modified with `makecontext()`, program execution continues with the function passed to `makecontext()`. When that function returns, the process continues as if after a call to `setcontext()` with the context pointed to by the `uc_link` member of the `ucontext_t` structure if it is not equal to 0. If the `ucp` argument was passed to a signal handler, program execution continues with the program instruction following the instruction interrupted by the signal. If the `uc_link` member of the `ucontext_t` structure pointed to by the `ucp` argument is equal to 0, then this context is the main context, and the process will exit when this context returns. The effects of passing a `ucp` argument obtained from any other source are undefined.

`setcontext()` is similar in some respects to `siglongjmp()` (and `longjmp()` and `_longjmp()`). The `getcontext()`–`setcontext()` pair, the `sigsetjmp()`–`siglongjmp()` pair, the `setjmp()`–`longjmp()` pair, and the `_setjmp()`–`_longjmp()` pair cannot be intermixed. A context saved by `getcontext()` should be restored only by `setcontext()`.

### Notes:

1. Some compatibility exists with `siglongjmp()`, so it is possible to use `siglongjmp()` from a signal handler to restore a context created with `getcontext()`, but it is not recommended.
2. If the `ucontext` that is input to `setcontext()` has not been modified by `makecontext()`, you must ensure that the function that calls `getcontext()` does not return before you call the corresponding `setcontext()` function. Calling `setcontext()` after the function calling `getcontext()` returns causes unpredictable program behavior.

This function is supported only in a POSIX program.

The `<ucontext.h>` header file defines the `ucontext_t` type as a structure that includes the following members:



mcontext_t	uc_mcontext	A machine-specific representation of the saved context.
ucontext_t	*uc_link	Pointer to the context that will be resumed when this context returns.
sigset_t	uc_sigmask	The set of signals that are blocked when this context is active.
stack_t	uc_stack	The stack used by this context.

### Special Behavior for C++

If `getcontext()` and `setcontext()` are used to transfer control in a OS/390 C++ program, the behavior in terms of the destruction of automatic objects is undefined. This applies to both OS/390 C++ and C/C++ OS/390 ILC modules. The use of `getcontext()` and `setcontext()` in conjunction with `try()`, `catch()`, and `throw()` is also undefined.

Do not issue `getcontext()` in a C++ constructor or destructor, since the saved context would not be usable in a subsequent `setcontext()` or `swapcontext()` after the constructor or destructor returns.

### Returned Value

If successful, `setcontext()` does not return.

If unsuccessful, `setcontext()` returns `-1`. There are no defined `errno` values.

### Example

This example saves the context in `main` with the *getcontext()* statement. It then returns to that statement from the function *func* using the *setcontext()* statement. Since *getcontext()* always returns 0 if successful, the program uses the variable `x` to determine if *getcontext()* returns as a result of *setcontext()* or not.

```
/* This example shows the usage of getcontext() and setcontext(). */

#define _XOPEN_SOURCE_EXTENDED 1
#include <stdio.h>
#include <ucontext.h>

void func(void);

int x = 0;
ucontext_t context, *cp = &context;

int main(void) {
    getcontext(cp);
    if (!x) {
        printf("getcontext has been called\n");
        func();
    }
    else {
        printf("setcontext has been called\n");
    }
}

void func(void) {
    x++;
    setcontext(cp);
}
```

```
}
```

**Output**

```
getcontext has been called  
setcontext has been called
```

**Related Information**

- “ucontext.h” on page 52
- “getcontext() — Get User Context” on page 505
- “longjmp() — Restore Stack Environment” on page 768
- “\_longjmp() — Non-Local Goto” on page 771
- “makecontext() — Modify User Context” on page 783
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “siglongjmp() — Restore the Stack Environment and Signal Mask” on page 1327
- “swapcontext() — Save and Restore User Context” on page 1472

## setegid() — Set the Effective Group ID

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a	both	MVS 4.3

### Format

```
#define _POSIX1_SOURCE 2
#include <unistd.h>
```

```
int setegid(gid_t gid);
```

### General Description

Sets the effective group ID (GID) of a process to *gid*, if *gid* is equal to the real GID or the saved set GID of the calling process, or if the process has appropriate privileges. The real GID, the saved set GID, and any supplementary GIDs are not changed.

### Returned Value

If successful, `setegid()` returns zero. If unsuccessful, it returns the value `-1` and sets `errno` to one of the following:

- EINVAL**      The value specified for *gid* is incorrect and is not supported by the implementation.
- EPERM**       The process does not have appropriate privileges, and *gid* does not match the real GID or the saved set GID.

### Example CBC3BS02

```
/* CBC3BS02
   This example changes your effective GID.
*/
#define _POSIX1_SOURCE 2
#include <unistd.h>
#include <stdio.h>

main() {
    printf("your effective group id is %d\n", (int) getegid());
    if (setegid(500) != 0)
        perror("setegid() error");
    else
        printf("your effective group id was changed to %d\n",
              (int) getegid());
}
```

### Output

```
your effective group id is 512
your effective group id was changed to 500
```

**Related Information**

- “unistd.h” on page 53
- “exec Functions” on page 322
- “getegid() — Get the Effective Group ID” on page 514
- “getgid() — Get the Real Group ID” on page 521
- “setgid() — Set the Group ID” on page 1220

setenv() — Add, Delete, and Change Environment Variables

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a Language Environment	both	

Format

POSIX - C only

```
#define _POSIX1_SOURCE 2
#include <stdlib.h>

int setenv(const char *var_name, const char *new_value, int change_flag)
```

Non-POSIX

```
#include <stdlib.h>

int setenv(const char *var_name, const char *new_value, int change_flag)
```

General Description

Adds, changes, and deletes environment variables.

To avoid infringing on the user's name-space, the non-POSIX version of this function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

| To provide an ASCII input/output format for applications using this function, define  
| feature test macro \_\_LIBASCII as described on page 22.

*var\_name* is a pointer to a character string that contains the name of the environment variable to be added, changed, or deleted. If setenv() is called with *var\_name* containing an equal sign ('='), setenv() will fail, and errno will be set to indicate that an invalid argument was passed to the function.

*new\_value* is a pointer to a character string that contains the value of the environment variable named in *var\_name*. If *new\_value* is a NULL pointer, it indicates that all occurrences of the environment variable named in *var\_name* be deleted.

*change\_flag* is a flag that can take any integer value:

Nonzero      Change the existing entry. If *var\_name* has already been defined and exists in the environment variable table, its value *will* be updated with *new\_value*. If *var\_name* was previously undefined, it will be appended to the table.

0 Do not change the existing entry. The new entry will be added even if the previous definition of *var\_name* exists. The current definition of *var\_name*, if it exists, is not changed.

If *var\_name* has already been defined and exists in the environment variable table, its value will *not* be updated with *new\_value*. However, if *var\_name* was previously undefined, it will be appended to the table.

#### Notes:

1. The value of the *change\_flag* is irrelevant if *new\_value*=NULL.
2. You should not define environment variables that begin with '\_BPXK\_' since they might conflict with variable names defined by OS/390 UNIX services. `setenv()` uses the BPX1ENV callable service to pass environment variables that begin with '\_BPXK\_' to the kernel.

Also, do not use '\_EDC\_' and '\_CEE\_'. They are used by the run-time library and the Language Environment.

Environment variables set with the `setenv()` function will only exist for the life of the program, and are not saved before program termination. Other ways to set environment variables are found in "Using Environment Variables" in the *OS/390 C/C++ Programming Guide*.

#### Special Behavior for POSIX C

Under POSIX, `setenv()` is available if one of the following is true:

- Code is compiled with the compiler option `LANGLV(ANSI)`, uses `#include <env.h>`, and has the POSIX feature tests turned on.
- Code is compiled with `LONGNAME` and prelinked with the OMVS option.

See "OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions" on page 9 for more information about using POSIX support.

#### Returned Value

Returns the value 0 if successful. Otherwise, it returns the value -1 and sets `errno` to indicate the type of failure that occurred.

#### Example CBC3BS03

```
/* CBC3BS03
   This example (program 1) sets the environment variable
   _EDC_ANSI_OPEN_DEFAULT.
   A child program (program 2) is then initiated via a system call.
   The example illustrates that environment variables are propagated
   forward to a child program, but not backward to the parent.
*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *x;

    /* set environment variable _EDC_ANSI_OPEN_DEFAULT to "Y" */
    setenv("_EDC_ANSI_OPEN_DEFAULT", "Y", 1);
```

```

/* set x to the current value of the _EDC_ANSI_OPEN_DEFAULT*/
x = getenv("_EDC_ANSI_OPEN_DEFAULT");

printf("program1 _EDC_ANSI_OPEN_DEFAULT = %s\n",
      (x != NULL) ? x : "undefined");

/* call the child program */
system("program2");

/* set x to the current value of the _EDC_ANSI_OPEN_DEFAULT*/
x = getenv("_EDC_ANSI_OPEN_DEFAULT");

printf("program1 _EDC_ANSI_OPEN_DEFAULT = %s\n",
      (x != NULL) ? x : "undefined");
}

```

### CBC3BS04

```

/* CBC3BS04
   Program 2:
   A child program of CBC3BS03, which is initiated via a system call.
*/
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char *x;

    /* set x to the current value of the _EDC_ANSI_OPEN_DEFAULT*/
    x = getenv("_EDC_ANSI_OPEN_DEFAULT");

    printf("program2 _EDC_ANSI_OPEN_DEFAULT = %s\n",
          (x != NULL) ? x : "undefined");

    /* clear the Environment Variables Table */
    setenv("_EDC_ANSI_OPEN_DEFAULT", NULL, 1);

    /* set x to the current value of the _EDC_ANSI_OPEN_DEFAULT*/
    x = getenv("_EDC_ANSI_OPEN_DEFAULT");

    printf("program2 _EDC_ANSI_OPEN_DEFAULT = %s\n",
          (x != NULL) ? x : "undefined");
}

```

### Output

```

program1 _EDC_ANSI_OPEN_DEFAULT = Y
program2 _EDC_ANSI_OPEN_DEFAULT = Y
program2 _EDC_ANSI_OPEN_DEFAULT = undefined
program1 _EDC_ANSI_OPEN_DEFAULT = Y

```

### Related Information

- “Using Environment Variables” in the *OS/390 C/C++ Programming Guide*
- “stdlib.h” on page 45
- “clearenv() — Clear Environment Variables” on page 184
- “getenv() — Get Value of Environment Variables” on page 515
- “\_\_getenv() — Get an Environment Variable” on page 517
- “putenv() — Change or Add an Environment Variable” on page 1054
- “system() — Execute a Command” on page 1488

## seteuid() — Set the Effective User ID

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a	both	MVS 4.3

### Format

```
#define _POSIX1_SOURCE 2
#include <unistd.h>
```

```
int seteuid(uid_t uid);
```

### General Description

Sets the effective user ID (UID) to *uid* if *uid* is equal to the real UID or the saved set user ID of the calling process, or if the process has appropriate privileges. The real UID and the saved set UID are not changed.

The seteuid() function is not supported from an address space running multiple processes, since it would cause all processes in the address space to have their security environment changed unexpectedly.

seteuid() can be used by daemon processes to change the identity of a process in order for the process to be used to run work on behalf of a user. In UNIX, changing the identity of a process is done by changing the real and effective UIDs and the auxiliary groups. In order to change the identity of the process on MVS completely, it is necessary to also change the MVS security environment. The identity change will only occur if the EUID value is specified, changing just the real UID will have no effect on the MVS environment.

The seteuid() function invokes MVS SAF services to change the MVS identity of the address space. The MVS identity that is used is determined as follows:

- If an MVS user ID is already known by the kernel from a previous call to a kernel function (for example, getpwnam()) and the UID for this user ID matches the UID specified on the seteuid() call, then this user ID is used.
- For nonzero target UIDs, if there is no saved user ID or the UID for the saved user ID does not match the UID requested on the seteuid() call, the seteuid() function queries the security database (for example, using getpwnam) to retrieve a user ID. The retrieved user ID is then used.
- If the target UID is 0 and a user ID is not known, the seteuid() function always sets the MVS user ID to BPXROOT or the value specified on the SUPERUSER parm in sysparms. BPXROOT is set up during system initialization as a super-user with a UID of 0. The BPXROOT user ID is not defined to the BPX.DAEMON FACILITY class profile. This special processing is necessary to prevent a superuser from gaining daemon authority.
- A nondaemon superuser that attempts to set a user ID to a daemon superuser UID fails with an EPERM.

When the MVS identity is changed, the auxiliary list of groups is also set to the list of groups for the new user ID.



If the seteuid() function is issued from multiple tasks within one address space, use synchronization to ensure that the seteuid() functions are not performed concurrently. The execution of seteuid() function concurrently within one address space can yield unpredictable results.

### Returned Value

If successful, seteuid() returns zero. If unsuccessful, it returns the value -1 and sets errno to one of the following:

EINVAL	The value specified for <i>uid</i> is incorrect and is not supported by the implementation.
EPERM	The process does not have appropriate privileges, and <i>uid</i> does not match the real UID or the saved set UID.

### Example CBC3BS05

```
/* CBC3BS05
   This example changes the effective UID.
*/
#define _POSIX1_SOURCE 2
#include <unistd.h>
#include <stdio.h>

main() {
    printf("your effective user id is %d\n", (int) geteuid());
    if (seteuid(25) != 0)
        perror("seteuid() error");
    else
        printf("your effective user id was changed to %d\n",
              (int) geteuid());
}
```

### Output

```
your effective user id is 0
your effective user id was changed to 25
```

### Related Information

- “unistd.h” on page 53
- “geteuid() — Get the Effective User ID” on page 519
- “getuid() — Get the Real User ID” on page 618
- “setreuid() — Set Real and Effective User IDs” on page 1262
- “setuid() — Set the Effective User ID” on page 1278

## setgid() — Set the Group ID

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX1_SOURCE 2
#include <unistd.h>
```

```
int setgid(gid_t gid);
```

### General Description

Sets one or more of the group IDs (GIDs) for the current process to *gid*.

If *gid* is the same as the process's real GID, setgid() always succeeds and sets the effective GID to be the same as the real GID.

If *gid* is not the same as the process's real GID, setgid() succeeds only if the process has appropriate privileges. If the process has such privileges, setgid() sets the real GID, the effective GID, and saved set GID to *gid*.

If \_POSIX\_SAVED\_IDS is defined in the sys/types.h header file, setgid() also sets the saved set GID to *gid*. To determine whether \_POSIX\_SAVED\_IDS is defined, you can use sysconf().

setgid() does not change any supplementary GIDs of the calling process.

### Returned Value

If successful, setgid() returns zero. If unsuccessful, it returns the value -1 and sets errno to one of the following:

EINVAL      The value of *gid* is incorrect.

EPERM        The process does not have appropriate privileges to set the GID.

### Example

#### CBC3BS06

```
/* CBC3BS06
   This example sets your GID.
*/
#define _POSIX1_SOURCE 2
#include <unistd.h>
#include <stdio.h>

main() {
    printf("your group id is %d\n", (int) getgid());
    if (setgid(500) != 0)
        perror("setgid() error");
    else
        printf("your group id was changed to %d\n",
              (int) getgid());
}
```

**Output**

```
your group id is 512  
your group id was changed to 500
```

**Related Information**

- “sys/types.h” on page 49
- “exec Functions” on page 322
- “getegid() — Get the Effective Group ID” on page 514
- “getgid() — Get the Real Group ID” on page 521
- “setuid() — Set the Effective User ID” on page 1278

## **setgrent() — Reset Group Database to First Entry**

The information for this function is included in “endgrent() — Group Database Entry Functions” on page 307.

## setgroups() — Set the Supplementary Group ID List for the Process

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 5.2.2

### Format

```
#define _OPEN_SYS
#include <sys/types.h>
#include <grp.h>
```

```
int setgroups(const int size, const gid_t list[ ]);
```

### General Description

setgroups() sets the supplementary group IDs for the process to the list provided in the *list* array. The argument *size* gives the number of *gid\_t* elements in *list* array. The maximum number of supplementary groups for a strictly conforming program is **NGROUPS\_MAX**, as defined in *<limits.h>*. Or, refer to the *sysconf()* function (see “*sysconf()* — Determine System Configuration Options” on page 1483) for information on dynamically determining the number of supplementary groups allowed.

The caller of this function must be a superuser.

### Returned Value

If successful, setgroups() returns 0.

If unsuccessful, setgroups() returns -1, and returns the error value in *errno*. The following are the possible values of *errno*:

- EFAULT**     The *list* and *size* specify an array that is partially or completely outside of addressable storage for the process.
- EINVAL**     The *size* parameter is greater than the maximum allowed.
- EMVSERR**    An MVS environmental or internal error occurred.
- EMVSSAF2ERR**  
                The System authorization facility (SAF) had an error.
- EPERM**       The caller is not authorized, only authorized users are allowed to alter the supplementary group IDs list.

### Related Information

- “getgroups() — Get a List of Supplementary Group IDs” on page 527
- “initgroups() — Initialize the Supplementary Group ID List for the Process” on page 669

## sethostent() — Open the Host Information Data Set

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>
```

```
void sethostent(int stayopen);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <netdb.h>
```

```
void sethostent(int stayopen);
```

### General Description

The `sethostent()` call opens and rewinds the `/etc/host` or `tcpip.HOSTS.SITEINFO` data set. If the `stayopen` flag is nonzero, the `/etc/host` or `tcpip.HOSTS.SITEINFO` data set remains open after each call.

You can use the **X\_SITE** environment variable to specify a data set other than `tcpip.HOSTS.SITEINFO`. For more information on these data sets and environment variables, see *OS/390 UNIX System Services Planning*.

**Note:** `tcpip.HOSTS.LOCAL`, `tcpip.HOSTS.ADDRINFO`, and `tcpip.HOSTS.SITEINFO` are described in *TCP/IP for MVS: Customization and Administration Guide*.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

No values are returned by the `sethostent()` function.

### Related Information

- “`endhostent()` — Work with a Host Entry” on page 308
- “`endnetent()` — Close Network Information Data Sets” on page 309
- “`gethostbyaddr()` — Get a Host Entry by Address” on page 531
- “`gethostbyname()` — Get a Host Entry by Name” on page 534
- “`gethostent()` — Get the Next Host Entry” on page 537

## setibmopt() — Set IBM TCP/IP Image

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#define _OPEN_SYS_SOCKET_EXT
#include <sys/socket.h>
```

```
int setibmopt(int cmd, struct ibm_tcpimage *bfrp);
```

### General Description

The `setibmopt()` function call is used to set TCP/IP options. Currently, the only supported command is `IBMTCP_IMAGE` which allows the `setibmopt()` to choose the active TCP/IP image stack the application will connect to.

To reset `ibm_tcpimage` to nothing chosen, set the *name* to all blanks.

The chosen transport is inherited over `fork()` and preserved over `exec()`. If this is not desired, the child process should call `setibmopt()` with a blank name to reset the TCP/IP image for the child.

### Parameter Description

<i>cmd</i>	The value in <i>cmd</i> must be set to the command to be performed. Currently, only <code>IBMTCP_IMAGE</code> is supported and must be paired with the <i>bfrp</i> parameter as described.
<i>bfrp</i>	The pointer to a <code>ibm_tcpimage</code> structure.

To set the TCP/IP image for a socket, the application should set values in the `ibm_tpcimage` structure as follows:

### Element Description

status	0 means is not known and need not be checked. Currently, this is the only value with meaning.
version	0 means the version is to be set on return if known.
name	The name must be left justified, uppercase, padded with blanks, and be the name of an active TCP stack.

### Returned Value

A 0 indicates success; the value -1 indicates a failure. The value of `errno` is set to one of the following on a failure:

### Errno Code Description

EFAULT	Using the <i>bfrp</i> supplied would result in access of a storage location that is inaccessible.
EIBMBADTCPNAME	A name of a PFS was specified that either is not configured or is not a Sockets PFS.
EOPNOTSUPP	The <i>cmd</i> is a function that is not supported.

### **Related Information**

- “sys/socket.h” on page 48
- “\_\_iptcpn() — Retrieve the Resolver Supplied Jobname or Userid” on page 693



## setibmssockopt() — Set Options Associated with a Bulk Mode Socket

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R3

### Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>
```

```
int setibmssockopt(int s,int level,int optname,
    char *optval, size_t optlen);
```

### General Description

Use `setibmssockopt()` with the *optname* `SO_BULKMODE` to place the UDP socket in bulk mode. Normally, UNIT transactions occur between the application and the TCP/IP address space for every receive (`read()`, `recv()`, `recvfrom()`, `recvmsg()`) or send (`send()`, `sendto()`, `sendmsg()`, `write()`) issued on a socket. The bulk mode socket option enables an application to queue multiple datagrams, sending all of the datagrams in one UNIT transaction. This reduces the CPU consumption for each datagram.

Bulk mode is only supported for receive-type socket calls. Currently, send-type socket calls are not supported for bulk mode. If a `setibmssockopt()` request is made to set the send queue size, the send queue size is set to zero automatically for the socket, and a zero return code is returned. Thus, no send-type socket calls will be queued.

For inbound datagrams, a queue of pending datagrams is maintained on the application side of the UNIT interface. As the application performs receive calls (`read()`, `recv()`, `recvfrom()`, `recvmsg()`), it draws from that queue. When the queue is empty, the receive call requests that TCP/IP pass over whatever datagrams are pending in one UNIT transaction. The oldest datagram is returned to the caller, and the application-side queue is replenished.

For outbound datagrams, a separate queue is kept on the application side of the UNIT interface. When the application performs send calls (`send()`, `sendto()`, `sendmsg()`, `write()`), the datagram is queued. When there are no more datagrams to send out, `ibmsflush()` is called to send the queued datagrams to the TCP/IP address space in one UNIT transaction.

### Notes:

1. Bulk mode is valid for send calls `send()`, `sendmsg()`, `sendto()`, and `write()` but is not valid for `writv()`. Bulk mode is valid for receive calls `read()`, `recv()`, `recvfrom()`, and `recvmsg()` but is not valid for `readv()`. For the `sendmsg()` and `recvmsg()` calls, only one datagram is processed per call (i.e. the buffer should be large enough to contain the entire message).
2. There is a restriction for applications that will pass socket descriptors between child and parent processes where one process intends to use the socket in bulk mode. The bulk mode send and receive buffers cannot be shared between processes. If the socket descriptor is to be passed to a child via `fork()` and `exec()` then the parent MUST issue `ibmsflush()` (if sending in bulk mode) to

clear the send buffer and *setibmssockopt()* with *b\_onoff* in the *ibm\_bulkmode\_struct* set to 0 to turn bulk mode off in the parent. The child can then issue *setibmssockopt()* against the same socket to turn bulk mode back on and begin operations in bulk mode in the new process. If the socket descriptor is to be passed back, then the child **MUST** follow the same sequence.

Parameter	Description
-----------	-------------

<i>s</i>	The socket descriptor.
<i>level</i>	The level for which the option is set.
<i>optname</i>	The name of a specified socket option.
<i>optval</i>	The pointer to option data. If <i>optname</i> is <i>SO_BULKMODE</i> , <i>optval</i> should point to an <i>ibm_bulkmode_struct</i> . If <i>optname</i> is <i>SO_NONBLOCKLOCAL</i> or <i>SO_IGNOREINCOMINGPUSH</i> , <i>optval</i> should point to an int.
<i>optlen</i>	The length of the option data.

To enable or disable bulk mode for a socket, the application should set values in the *ibm\_bulkmode\_struct* as follows:

Element	Description
---------	-------------

<i>b_onoff</i>	1 means bulk mode is on; 0 means bulk mode is off.
<i>b_max_receive_queue_size</i>	The maximum receiving queue size in bytes. Specifying a value of 0 prevents queuing for inbound datagrams. The maximum value allowed is 65768.
<i>b_max_send_queue_size</i>	The maximum sending queue size in bytes. Specifying a value of 0 prevents queuing for outbound datagrams. If a value greater than 0 is specified, the minimum value is 37. The maximum value allowed is 65768. This value is set to zero, since send-type socket calls are not currently supported for bulk mode.
<i>b_move_data</i>	For outbound sockets, if <i>b_move_data</i> is nonzero, the data is moved into buffers in the queue. The client's buffers can be reused right away. If <i>b_move_data</i> is zero, pointers to the data are saved in the queue. The buffers should not be reused until the queue has been flushed (generally by issuing an <i>ibmsflush()</i> ).
<i>b_teststor</i>	If this element is nonzero, the message buffer address and the message buffer are checked for addressability during each socket call. <i>errno</i> is set to <i>EFAULT</i> if either address or buffer is cannot be addressed. If this element is zero, no checking is performed.
<i>b_max_send_queue_size_avail</i>	The maximum send queue size in bytes that can be set by <i>b_max_send_queue_size</i> option on <i>setibmssockopt()</i> . This value is set to zero, since send-type socket calls are not currently supported for bulk mode.

**b\_num\_UNITS\_sent**

The number of actual UNITS issued in sending datagrams to TCP/IP. This value is set to zero, since send-type socket calls are not currently supported for bulk mode. This value is only used by `getibmsockopt()`.

**b\_num\_UNITS\_received**

The number of actual UNITS issued in receiving datagrams from TCP/IP. This value is only used by `getibmsockopt()`.

The fields `b_num_UNITS_sent` and `b_num_UNITS_received` represent cumulative totals for this socket since the time the application was started.

The socket calls that receive datagrams in your program (`recvfrom()`, `recv()`, `read()`, or `recvmsg()`) do not need to be changed. If you are using a queue for sending datagrams (by specifying a nonzero value for `b_max_send_queue_size` above), you should code `ibmsflush()` to flush the socket at appropriate points, such as after you send a burst of datagrams that is normally followed by a pause.

For option `SO_NONBLOCKLOCAL`, *optval* should point to an integer. If *optval* points to 1, the socket is placed in nonblocking mode. In nonblocking mode, when the application-side queue is empty, a value of -1 is returned on a receive and sets `errno` to `EWOULDBLOCK`. If *optval* points to 0, the socket is placed in blocking mode.

Before the application calls `SO_NONBLOCKLOCAL`, the socket is in blocking mode.

This option will enable blocking only for sockets that are operating in bulk mode. If the `SO_NONBLOCKLOCAL` option is used for a socket that is not in bulk mode, `EINVAL` will be returned. If blocking is desired for sockets that are not operating in bulk mode, *fcntl()* should be used with the file flag `O_NONBLOCK` as the command.

`SO_IGNOREINCOMINGPUSH` is another option to consider. This option is meaningful only for stream sockets. This option is effective only for connections established through an offload box. If *optval* points to 1, the option is set. If *optval* points to 0, the option is off.

The `SO_IGNOREINCOMINGPUSH` option causes a receive call to return when:

- The requested length is reached.
- The internal TCP/IP length is reached.
- The peer application closes the connection.

The amount of data returned for each call is maximized and the amount of CPU time consumed by your program and TCP/IP can be reduced.

This option is not appropriate for your operation if your program depends on receiving data before the connection is closed. For example, this option is appropriate for an FTP data connection, but not for a Telnet connection.

## Returned Value

The value 0 indicates success. The value -1 indicates an error. The value of `errno` indicates the specific error.

## Errno Code Description

EBADF	The <i>s</i> parameter is not a valid socket descriptor.
EFAULT	Using <i>optval</i> and <i>optlen</i> parameters would result in an attempt to access storage outside the caller's address space.
EINVAL	The specified option is invalid at the specified socket level or the socket has been shut down. The requested send or receive queue size is greater than the maximum allowed. The maximum size allowed is 65768. The requested send queue size is less than the minimum required, 37. The <code>SO_NONBLOCKALL</code> option is used for a socket that is not in bulk mode.
ENOMEM	There is not enough storage in the caller's address space to allocate the receive queue to the <code>b_max_receive_queue_size</code> specified in the <code>ibm_bulkmode_struct</code> . <code>b_max_receive_queue_size</code> is reset to 0, turning bulk mode off. Subsequent requests will pass through as normal receives. There is not enough storage in the caller's address space to allocate the send queue to the <code>b_max_send_queue_size</code> specified in the <code>ibm_bulkmode_struct</code> . <code>b_max_send_queue_size</code> is reset to 0, turning bulk mode off. Subsequent requests will pass through as normal sends.
ENOPROTOOPT	The <i>optname</i> parameter is unrecognized. The <i>level</i> parameter is not <code>SOL_SOCKET</code> . The domain of the socket descriptor is not <code>AF_INET</code> . The socket descriptor is not a datagram type socket.

## Example

The following is an example of the `setibmssockopt()` call.

```
#include <stdio.h>
#include <socket.h>

{ struct ibm_bulkmode_struct bulkstr;
  int optlen, rc, s;
  FILE *stream;

  /* Create, bind, etc done for socket s */
  .
  .
  .
  optlen = sizeof(bulkstr);
  rc = getibmssockopt(s, SOL_SOCKET, SO_BULKMODE, (char *) &bulkstr, &optlen);
  if (rc < 0) {
    tcperror("on getibmssockopt()");
    exit(-1);
  }
  fprintf(stream, "%d byte buffer available for outbound queue.\n",
    bulkstr.b_max_send_queue_size_avail);

  bulkstr.b_max_send_queue_size=bulkstr.b_max_send_queue_size_avail;
  bulkstr.b_onoff = 1;
  bulkstr.b_teststor = 0;
  bulkstr.b_move_data = 1;
  bulkstr.b_max_receive_queue_size = 65536;
  rc = setibmssockopt(s, SOL_SOCKET, SO_BULKMODE, (char *) &bulkstr, optlen);
  if (rc < 0) {
```

```
        tcperror("on setibmsockopt()");  
        exit(-1);  
    }  
}
```

## Related Information

- “sys/socket.h” on page 48
- “fcntl() — Control Open File Descriptors” on page 350
- “getibmsockopt() — Get the Options Associated with a Bulk Mode Socket” on page 543
- “getsockopt() — Get the Options Associated with a Socket” on page 606
- “ibmsflush() — Flush the Application-side Datagram Queue” on page 652
- “setsockopt() — Set Options Associated with a Socket” on page 1270

## setitimer() — Set Value of an Interval Timer

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON) MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/time.h>
```

```
int setitimer(int which, struct itimerval * value, struct itimerval * ovalue);
```

### General Description

setitimer() sets the value of an interval timer. An interval timer is a timer which sends a signal after each repetition (interval) of time.

The *which* argument indicates what kind of time is being controlled. Values for *which* are:

#### ITIMER\_REAL

This timer is marking real (clock) time. A SIGALRM signal is generated after each interval of time.

**Note:** alarm() also sets the real interval timer.

#### ITIMER\_VIRTUAL

This timer is marking process virtual time. Process virtual time is the amount of time spent while executing in the process, and can be thought of as a CPU timer. A SIGVTALRM signal is generated after each interval of time.

#### ITIMER\_PROF

This timer is marking process virtual time plus time spent while the system is running on behalf of the process. A SIGPROF signal is generated after each interval of time.

**Note:** In a multi-threaded environment, each of the above timers is specific to a thread of execution for both the generation of the time interval and the measurement of time. For example, an ITIMER\_VIRTUAL timer will mark execution time for just the thread, not the entire process.

The *value* argument points to an itimerval structure containing the timer value to be set. The structure contains:

```
it_interval timer interval
```

When it\_interval is nonzero, it is used as the value which it\_value is initialized to after each timer expiration. If it\_interval is zero, the timer is disabled **after the next expiration**, subject to the value in it\_value.

`it_value`      current timer value to be set

When `it_value` is nonzero, it is used as the initial value to establish the timer with, i.e. the time to the next timer expiration. If `it_value` is zero, the timer is **immediately** disabled.

The *ovalue* argument points to an `itimerval` structure in which the current value of the timer is returned. If *ovalue* is a NULL pointer, the current timer value is not returned. The structure contains:

`it_interval`    current timer interval

`it_value`      current timer value

For both `itimerval` structures, each of the fields (`it_interval` and `it_value`) is a `timeval` structure, and contains:

`tv_sec`        seconds since January 1, 1970

`tv_usec`      microseconds

### Returned Value

If successful, `setitimer()` returns zero, and if *ovalue* was non-NULL, *ovalue* points to the `itimerval` structure containing the old timer values.

If unsuccessful, `setitimer()` returns -1, and returns the error value in `errno`. The following are the possible values of `errno`:

**EINVAL**      *which* is not a valid timer type, or the *value* argument has an incorrect (non-canonical) form. The `tv_seconds` field must be a non-negative integer, and the `tv_usec` field must be a non-negative integer in the range of 0 to 1,000,000.

Usage of the `ITIMER_PROF` timer generates a `SIGPROF` signal which may interrupt an in-progress function. Thus, programs using this timer may need to be able to restart an interrupted function.

### Related Information

- “`sys/time.h`” on page 49
- “`alarm()` — Set an Alarm” on page 102
- “`getitimer()` — Get Value of an Interval Timer” on page 548
- “`gettimeofday()` — Get Date and Time” on page 616
- “`sleep()` — Suspend Execution of a Thread” on page 1364
- “`ualarm()` — Set the Interval Timer” on page 1640
- “`usleep()` — Suspend Execution for an Interval” on page 1663

## setjmp() — Preserve Stack Environment

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <setjmp.h>
```

```
int setjmp(jmp_buf env);
```

### General Description

Saves a stack environment that can subsequently be restored by `longjmp()`. The `setjmp()` and `longjmp()` functions provide a way to perform a nonlocal goto. They are often used in signal handlers.

A call to `setjmp()` causes it to save the current stack environment in *env*. A subsequent call to `longjmp()` restores the saved environment and returns control to a point corresponding to the `setjmp()` call. The values of all variables, except register variables and nonvolatile automatic variables, accessible to the function receiving control, contain the values they had when `longjmp()` was called. The values of register variables are unpredictable. Nonvolatile *auto* variables that are changed between calls to `setjmp()` and `longjmp()` are also unpredictable.

An invocation of `setjmp()` must appear in one of the following contexts only:

- The entire controlling expression of a selection or iteration statement
- One operand of a relational or equality operator with the other operand an integral constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement
- The operand of a unary “!” operator with the resulting expression being the entire controlling expression of a selection or iteration
- The entire expression of an expression statement (possibly cast to void).

**Note:** Ensure that the function that calls `setjmp()` does not return before you call the corresponding `longjmp()` function. Calling `longjmp()` after the function calling `setjmp()` returns causes unpredictable program behavior.

### Special Behavior for POSIX C

To save and restore a stack environment that includes a signal mask, use `sigsetjmp()` and `siglongjmp()`, instead of `setjmp()`. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information about using POSIX support.

The `sigsetjmp()`—`siglongjmp()` pair, the `setjmp()`—`longjmp()` pair, the `_setjmp()`—`_longjmp()` pair, and the `getcontext()`—`setcontext()` pair *cannot* be intermixed. A stack environment saved by `setjmp()` can only be restored by `longjmp()`.

### Special Behavior for C++



If `setjmp()` and `longjmp()` are used to transfer control in a OS/390 C++ program, the behavior in terms of the destruction of automatic objects is undefined. This applies both to C OS/390 and C/C++ OS/390 ILC modules. The use of `setjmp()` and `longjmp()` in conjunction with `try()`, `catch()`, and `throw()` is also undefined.

### Special Behavior for XPG4.2

In a program that was compiled with the feature test macro `_XOPEN_SOURCE_EXTENDED` defined, another pair of functions, `_setjmp()`—`_longjmp()`, are available. On this implementation, these calls are functionally identical to `setjmp()`—`longjmp()`. Therefore it is possible, but not recommended, to intermix the `setjmp()`—`longjmp()` pair with the `_setjmp()`—`_longjmp()` pair.

### Returned Value

Returns the value 0 after saving the stack environment. If `setjmp()` returns as a result of a `longjmp()` call, it returns the *value* argument of `longjmp()`, or the value 1 if the *value* argument of `longjmp()` is equal to 0.

### Example

This example stores the stack environment at the statement:

```
if(setjmp(mark) != 0) ...
```

When the system first performs the `if` statement, it saves the environment in *mark* and sets the condition to FALSE because `setjmp()` returns the value 0 when it saves the environment. The program prints the message: `setjmp has been called`.

The subsequent call to function *p* tests for a local error condition, which can cause it to perform the `longjmp()` function. Then control returns to the original `setjmp()` function using the environment saved in *mark*. This time the condition is TRUE because `-1` is the returned value from the `longjmp()` function. The program then performs the statements in the block and prints: `longjmp has been called`. Finally, the program calls the recover function and exits.

```
/* This example shows the effect of having set the stack environment. */
#include <stdio.h>
#include <setjmp.h>

jmp_buf mark;

void p(void);
void recover(void);

int main(void)
{
    if (setjmp(mark) != 0) {
        printf("longjmp has been called\n");
        recover();
        exit(1);
    }
    printf("setjmp has been called\n");
    :
    p();
    :
}

void p(void)
{
```

```
    int error = 0;
    :
    error = 9;
    :
    if (error != 0)
        longjmp(mark, -1);
    :
}

void recover(void)
{
    :
}
```

### Related Information

- “setjmp.h” on page 40
- “getcontext() — Get User Context” on page 505
- “longjmp() — Restore Stack Environment” on page 768
- “\_longjmp() — Non-Local Goto” on page 771
- “setcontext() — Restore User Context” on page 1210
- “sigsetjmp() — Save Stack Environment and Signal Mask” on page 1346
- “siglongjmp() — Restore the Stack Environment and Signal Mask” on page 1327
- “\_setjmp() — Set Jump Point for a Non-Local Goto” on page 1237
- “swapcontext() — Save and Restore User Context” on page 1472

## `_setjmp()` — Set Jump Point for a Non-Local Goto

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <setjmp.h>
```

```
int _setjmp(jmp_buf env);
```

### General Description

The `_setjmp()` function saves a stack environment that can subsequently be restored by `_longjmp()`. The `_setjmp()` and `_longjmp()` functions provide a way to perform a non-local goto. They are often used in signal handlers.

A call to `_setjmp()` causes it to save the current stack environment in *env*. A subsequent call to `_longjmp()` restores the saved environment and returns control to a point corresponding to the `_setjmp()` call. The values of all variables, except register variables, and except nonvolatile automatic variables, accessible to the function receiving control contain the values they had when `_longjmp()` was called. The values of register variables are unpredictable. Non-volatile *auto* variables that are changed between calls to `_setjmp()` and `_longjmp()` are also unpredictable.

An invocation of `_setjmp()` must appear in one of the following contexts only:

1. The entire controlling expression of a selection or iteration statement.
2. One operand of a relational or equality operator with the other operand an integral constant expression, with the resulting expression being the entire controlling expression of a selection or iteration statement.
3. The operand of a unary "!" operator with the resulting expression being the entire controlling expression of a selection or iteration.
4. The entire expression of an expression statement (possibly cast to void).

The X/Open standard states that `_setjmp()` and `_longjmp()` are functionally identical to `longjmp()` and `setjmp()`, respectively, with the addition restriction that `_setjmp()` and `_longjmp()` do not manipulate the signal mask. However, on this implementation `longjmp()` and `setjmp()` do not manipulate the signal mask. So on this implementation `_setjmp()` and `_longjmp()` are literally identical to `longjmp()` and `setjmp()`, respectively.

To save and restore a stack environment, including the current signal mask, use `sigsetjmp()` and `siglongjmp()` instead of `_setjmp()` and `_longjmp()`, or `setjmp()` and `longjmp()`.

The `_setjmp()`—`_longjmp()` pair, the `setjmp()`—`longjmp()` pair, the `sigsetjmp()`—`siglongjmp()` pair, and the `getcontext()`—`setcontext()` pair cannot be intermixed. A stack environment saved by `_setjmp()` can be restored only by `_longjmp()`.

### **Notes:**

1. However, on this implementation, since the `_setjmp()`—`_longjmp()` pair are functionally identical to the `setjmp()`—`longjmp()` pair it is possible to intermix them, but it is not recommended.
2. Ensure that the function that calls `_setjmp()` does not return before you call the corresponding `_longjmp()` function. Calling `_longjmp()` after the function calling `_setjmp()` returns causes unpredictable program behavior.

### **Special Behavior for C++**

If `_setjmp()` and `_longjmp()` are used to transfer control in a OS/390 C++ program, the behavior in terms of the destruction of automatic objects is undefined. This applies both to C/OS/390 and C/C++ OS/390 ILC modules. The use of `_setjmp()` and `_longjmp()` in conjunction with `try()`, `catch()`, and `throw()` is also undefined.

### **Returned Value**

The `_setjmp()` function returns the value 0 after saving the stack environment. If `_setjmp()` returns as a result of a `_longjmp()` call, it returns the *value* argument of `_longjmp()`, or 1 if the *value* argument of `_longjmp()` was 0.

### **Related Information**

- “setjmp.h” on page 40
- “getcontext() — Get User Context” on page 505
- “\_longjmp() — Non-Local Goto” on page 771
- “longjmp() — Restore Stack Environment” on page 768
- “setcontext() — Restore User Context” on page 1210
- “setjmp() — Preserve Stack Environment” on page 1234
- “siglongjmp() — Restore the Stack Environment and Signal Mask” on page 1327
- “sigsetjmp() — Save Stack Environment and Signal Mask” on page 1346
- “swapcontext() — Save and Restore User Context” on page 1472

# setkey() — Set Encoding Key

## Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

## Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>

void setkey(const char *key);
```

## General Description

The setkey() function transforms the *key* argument array into data encryption keys which are used by the encrypt() function to encode blocks of data.

The *key* argument of setkey() is an array of length 64 bytes containing only the bytes with numerical value of 0 and 1. If this 64 byte array is divided into groups of 8, the low-order byte of each group is ignored. The setkey() function transforms the remaining 56 bytes, each with values 0 or 1, into 16 48-bit keys according to the Data Encryption Standard (DES) key algorithm.

To provide an ASCII input/output format for applications using this function, define feature test macro \_\_LIBASCII as described on page 22.

## Special Behavior for OS/390 UNIX Services

When setkey() is called from a thread, the array of 16 bit-bit keys produced by setkey() is unique to the thread. Thus, for each thread from which the encrypt() function is called by a threaded application, the setkey() function must first be called from each thread.

## Returned Value

No values are returned by the setkey() function.

## Special Behavior for OS/390 UNIX Services

The setkey() function will fail if:

- ENOMEM     Unable to allocate storage for DES keys on thread from which setkey() invoked.
- EINVAL     64 byte input array contains bytes with values other than 0x00 or 0x01.

**Note:** Because setkey() does not return a value, applications wishing to check for errors should set errno to 0, call setkey(), then test errno and, if it is nonzero, assume an error has occurred.

**Related Information**

- “stdlib.h” on page 45

setlocale() — Set Locale

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 SAA Language Environment OS/390 UNIX	both	

Format

```
#include <locale.h>

char *setlocale(int category, const char *locale);
```

General Description

Sets, changes, or queries *locale* categories or groups of categories. It does this action according to values of the *locale* and *category* arguments.

A *locale* is the complete definition of the part of a user's program that depends on language and cultural conventions. You can accept the default value of *locale*, or you can set it to one of the supplied locales listed in the appendix, "Supplied Locales", in the *OS/390 C/C++ Programming Guide*. Some examples of the supplied locales are: "C", "POSIX", "SAA", "S370", "Fr\_BE.IBM-1047", "En\_GB.IBM-285", "En\_US.IBM\_1047", "Fr\_BE.IBM-1148@euro", and "Fr\_BE.IBM-1148".

Note that non-POSIX programs may exploit the POSIX style of locale support. This use of environment variables also applies to non-POSIX programs that use POSIX locale support.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

Effect of setlocale() on Language Environment

The current locale set with the `setlocale()` function affects only some C library functions. (See Table 30). It does not affect the new CEE locale set and query functions available under Language Environment and described in the *IBM Language Environment Programming Reference*.

The Category Argument

The category argument may be set to one of these values:

Table 30 (Page 1 of 3). Values for Category Arguments of `setlocale()`

Category	Purpose
LC_ALL	Specifies all categories associated with the program's locale.

Table 30 (Page 2 of 3). Values for Category Arguments of setlocale()

Category	Purpose
LC_COLLATE	<p>Defines the collation sequence, that is, the relative order of collation elements (characters and multi-character collation elements) in the program's locale. The collation sequence definition is used by regular expression, pattern matching, and sorting functions.</p> <p>These string functions are affected by the defined collation sequence: <code>strcoll()</code>, <code>strxfrm()</code>, <code>wscoll()</code>, and <code>wcsxfrm()</code>.</p> <p>LC_CTYPE, LC_COLLATE, and LC_SYNTAX should refer to the same locale. Changing just one of them may invalidate another.</p>
LC_CTYPE	<p>Defines character classification and case conversion for characters in the program's locale. Affects the behavior of character-handling functions defined in the <code>ctype.h</code> header file: <code>csid()</code>, <code>isalnum()</code>, <code>isalpha()</code>, <code>isblank()</code>, <code>iswblank()</code>, <code>iscntrl()</code>, <code>isdigit()</code>, <code>isgraph()</code>, <code>islower()</code>, <code>isprint()</code>, <code>ispunct()</code>, <code>isspace()</code>, <code>isupper()</code>, <code>iswalnum()</code>, <code>iswalpha()</code>, <code>iswcntrl()</code>, <code>iswctype()</code>, <code>iswdigit()</code>, <code>iswgraph()</code>, <code>iswlower()</code>, <code>iswprint()</code>, <code>iswpunct()</code>, <code>iswspace()</code>, <code>iswupper()</code>, <code>iswxdigit()</code>, <code>isxdigit()</code>, <code>tolower()</code>, <code>toupper()</code>, <code>towlower()</code>, <code>towupper()</code>, <code>wcsid()</code>, and <code>wctype()</code>.</p> <p>Affects behavior of the <code>printf()</code> and <code>scanf()</code> families of functions: <code>fprintf()</code>, <code>printf()</code>, <code>sprintf()</code>, <code>fscanf()</code>, <code>scanf()</code>, and <code>sscanf()</code>.</p> <p>Affects the behavior of wide character input/output functions: <code>fgetwc()</code>, <code>fgetws()</code>, <code>getwc()</code>, <code>getwchar()</code>, <code>fputwc()</code>, <code>fputws()</code>, <code>putwc()</code>, <code>putwchar()</code>, and <code>ungetwc()</code>.</p> <p>Affects the behavior of multibyte and wide character functions: <code>mblen()</code>, <code>mbtowc()</code>, <code>mbstowcs()</code>, <code>wctomb()</code>, <code>wcstombs()</code>, <code>mbrlen()</code>, <code>mbrtowc()</code>, <code>mbsrtowcs()</code>, <code>wcrtomb()</code>, <code>wcsrtombs()</code>, <code>wcswidth()</code>, <code>wcwidth()</code>, <code>wcstod()</code>, <code>wcstol()</code>, and <code>wcstoul()</code>.</p> <p>LC_CTYPE, LC_COLLATE, and LC_SYNTAX should refer to the same locale. Changing just one of them may invalidate another.</p>
LC_MESSAGES	<p>Under OS/390 C/C++ support, it affects the messages returned by the <code>nl_langinfo()</code> function and it also has an effect on <code>rpmatch()</code>.</p> <p>The LC_MESSAGES category will <i>not</i> affect the messages for the following functions: <code>perror()</code>, <code>strerror()</code>, and <code>regerror()</code>.</p> <p>In the locale of a C program running with POSIX(ON), it defines affirmative and negative response patterns.</p>
LC_MONETARY	<p>Affects monetary information returned by <code>localeconv()</code> and the <code>strfmon()</code> function. It defines the rules and symbols used to format monetary numeric information in the program's locale. The formatting rules and symbols are strings. <code>localeconv()</code> returns pointers to these strings with names found in the <code>locale.h</code> header file.</p>



Table 30 (Page 3 of 3). Values for Category Arguments of setlocale()

Category	Purpose
LC_NUMERIC	<p>Affects the decimal-point character for the formatted input/output and string conversion functions, and the non-monetary formatting information returned by the localeconv() function, specifically:</p> <ul style="list-style-type: none"> <li>• The printf() family of functions</li> <li>• The scanf() family of functions</li> <li>• strtod()</li> <li>• atof()</li> </ul> <p>The formatting rules and symbols are strings. localeconv() returns pointers to the strings with names found in the locale.h header file.</p>
LC_TIME	<p>Defines time and date format information in the program's locale used by the strftime(), strptime(), and wcsftime() functions.</p>
LC_SYNTAX	<p>Affects the behavior of functions that use encoded values to format characters:</p> <ul style="list-style-type: none"> <li>• printf() family of functions</li> <li>• scanf() family of functions</li> <li>• regcomp()</li> <li>• strfmon()</li> </ul> <p>LC_SYNTAX also affects values that may be retrieved using the getsyntax() function.</p> <p>LC_CTYPE, LC_COLLATE, and LC_SYNTAX should refer to the same locale. Changing just one of them may invalidate another.</p>
LC_TOD	<p>Affects the behavior of the functions related to time zone and Daylight Savings Time information in the program's locale, when time zone and Daylight Savings Time information is not defined by the TZ environment variable. This information is used by ctime(), localtime(), mktime(), and strftime().</p>

For a POSIX program, the functions ctime(), localtime(), mktime(), setlocale(), and strftime() call the tzset() function to override LC\_TOD category information when TZ is defined and valid. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information about using POSIX support.

## The Locale Argument

Identifies the locale. For a list of locales provided by IBM refer to the appendix, “Supplied Locales”, in the *OS/390 C/C++ Programming Guide*.

If the value of an environment variable is used, it must be a valid locale name. If this is the case, setlocale() sets the specified category to the named locale, and returns a string giving the name of the locale. Otherwise, setlocale() does not change the program's locale and returns a NULL pointer. Valid category names include names of locales provided by IBM. Also, names of locales, which are created using the OS/390 C/C++ locale definition mechanism, are valid.

## The Null-String ("" ) Locale Value

If "" is specified, the locale-related environment variables are checked. If the locale name is not defined by the environment variables, the default is "S370" when

running POSIX(OFF) and "C" when running POSIX(ON). See "OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions" on page 9 for more information.

For both C and C++ languages, and whether you are using POSIX or not, if a program using POSIX-style locale support specifies "" for the value of *locale*, then `setlocale()` interrogates locale-related environment variables in the program's environment to find a locale name or names to use. The locale name is chosen according to the first of the following conditions that applies:

1. If the environment variable `LC_ALL` is defined and is not null, the value of `LC_ALL` is used. That value is applied to all categories.
2. If individual environmental variables are defined, then their values are used for the categories.
3. If the environment variable `LANG` is defined and is not null, the value of `LANG` is used.
4. If no non-null environment variable is present to supply a value, "C" is used.

If a program using POSIX-style locale support specifies `LC_ALL` for the value of *category* and "" for the value of *locale*, `setlocale()` searches environment variables in the way just described to obtain a locale name for each category. If all the locale names obtained identify valid locales, `setlocale()` sets each category to the appropriate locale and returns a string naming the locale associated with each category. Otherwise, `setlocale()` does not change the program's locale and returns a NULL pointer.

### Default Locale

The relationship between the POSIX C and SAA C locales is as follows.

Using C or C++ languages with the runtime option POSIX(OFF):

1. The SAA C locale definition is the default. "C", "SAA", and "S370" are synonyms for the SAA C locale definition, which is prebuilt into the library.

The source file `EDC$SAAC_LOCALE` is provided for reference, but cannot be used to alter the definition of this prebuilt locale.

2. Issuing `setlocale(category, "")` has the following effect:
  - Locale-related environment variables are checked to find the name of locales) to use to set the *category* specified. Querying the locale with `setlocale(category, NULL)` returns the name of the locales specified by the appropriate environment variables.
  - If no non-null environment variable is present, it is the equivalent of having issued `setlocale(category, "S370")`. That is, the locale chosen is the SAA C locale definition, and querying the locale with `setlocale(category, NULL)`, returns "S370" as the locale name.
3. If no `setlocale()` function is issued or `setlocale(LC_ALL, "C")` is used, then the locale chosen is the prebuilt SAA C locale, and querying the locale with `setlocale(category, NULL)`, returns "C" as the locale name.
4. For `setlocale(LC_ALL, "SAA")`, the locale chosen is the prebuilt SAA C locale, and querying the locale with `setlocale(category, NULL)`, returns "SAA" as the locale name.

5. For `setlocale(LC_ALL, "S370")`, the locale chosen is the prebuilt SAA C locale, and querying the locale with `setlocale(category, NULL)`, returns "S370" as the locale name.
6. For `setlocale(LC_ALL, "POSIX")`, the locale chosen is the prebuilt POSIX C locale, and querying the locale with `setlocale(category, NULL)`, returns "POSIX" as the locale name.

Using OS/390 C with the runtime option POSIX(ON):

1. The POSIX C locale definition is the default. "C" and "POSIX" are synonyms for the POSIX C locale definition, which is prebuilt into the library.

The source file `EDC$POSX LOCALE` is provided for reference, but cannot be used to alter the definition of this prebuilt locale.

2. Issuing `setlocale(category, "")` has the following effect:
  - Locale-related environment variables are checked to find the name of locales that can set the *category* specified. Querying the locale with `setlocale(category, NULL)` returns the name of the locale specified by the appropriate environment variables.
  - If no non-null environment variable is present, the result is equivalent to having issued `setlocale(category, "C")`. That is, the locale chosen is the POSIX C locale definition, and querying the locale with `setlocale(category, NULL)`, returns "C" as the locale name.
3. If no `setlocale()` function is issued or if `setlocale(LC_ALL, "C")` is used, the locale chosen is the prebuilt POSIX C locale. Querying the locale with `setlocale(category, NULL)` returns "C" as the locale name.
4. For `setlocale(LC_ALL, "POSIX")` the locale chosen is the prebuilt POSIX C locale. Querying the locale with `setlocale(category, NULL)` returns "POSIX" as the locale name.
5. For `setlocale(LC_ALL, "SAA")` the locale chosen is the prebuilt SAA C locale. Querying the locale with `setlocale(category, NULL)` returns "SAA" as the locale name.
6. For `setlocale(LC_ALL, "S370")` the locale chosen is the prebuilt SAA C locale. Querying the locale with `setlocale(category, NULL)` returns "S370" as the locale name.

The `setlocale()` function supports locales by using the `localedef` utility, as well as locales built using the assembler language source and produced by the `EDCLOC` macro. Find more information about old format locales in "Internationalization: Locales and Character Sets", in the *OS/390 C/C++ Programming Guide*.

### Special Behavior for OS/390 UNIX Services

The `LOCPATH` environment variable specifies a colon separated list of HFS directories. If `LOCPATH` is defined, `setlocale()` searches HFS directories in the order specified by `LOCPATH` for locale object files it requires. Locale object files in the HFS are produced by the `localedef` utility running under the OS/390 UNIX services Shell and Utilities. If `LOCPATH` is not defined and `setlocale()` is called by a POSIX program, `setlocale()` looks in the default HFS locale directory, `/usr/lib/nls/locale`, for locale object files it requires. If `setlocale()` does not find a locale object it requires in the HFS, it converts the locale name to a PDS member name as described in the

*OS/390 C/C++ Programming Guide* and searches locale PDS load libraries associated with the program calling `setlocale()`.

Locale names may be filenames, relative pathnames, or absolute pathnames. `LOCPATH` is used if filename rather than pathname is specified. Also, `//` preceding a filename tells `setlocale()` to skip HFS search, to convert the name to a load module name of the form `EDC$xxxx`, and to search MVS load libraries for a member to load with this name.

### Invocation Sequence for `setlocale()`

In all three variations of the `setlocale()` function call, a pointer to a string that represents the locale value is returned. Also, in all variations, if the value for either category or locale is invalid, `setlocale()` returns a NULL pointer and the operating environment is not changed.

Each variation causes a different function to be performed:

1. `setlocale(category, locale);`

When an explicit locale is named, the category named in the call is set according to the named locale.

2. `setlocale(category, "");`

When the locale argument of the `setlocale()` function is given as a null string (`""`), the `setlocale()` function sets the locale environment according to the environment variables. If these are not set, the default locale "S370" is used. This locale may be customized when the OS/390 C/C++ product is installed. See "Using Environment Variables" in the *OS/390 C/C++ Programming Guide*.

The environment variables are not currently supported under all OS/390 C/C++ environments. The processing above will allow the `setlocale()` function to use the environment variables if they are available, and to use the "S370" locale otherwise.

3. `setlocale(category, (char *) 0);`

When a null pointer is given as a locale, a pointer to a string that represents the current locale for the specified category is returned. The string has the property that if it were specified as the locale of a subsequent `setlocale()` call of the same category, the current locale would be restored. For example, the following sequence is effectively a no-op:

```
setlocale(category, setlocale(category, (char *) 0));
```

When called with a null string (for example, `setlocale(LC_ALL, "")`), `setlocale()` determines the locale to be set, using the environment variables, and checking them in this order:

1. `LC_ALL`. If set, it specifies the name for all categories; it can override the values in the other environmental variables.
2. `LC_COLLATE`, `LC_CTYPE`, `LC_MESSAGES`, `LC_MONETARY`, `LC_NUMERIC`, `LC_TIME`, `LC_SYNTAX`, and `LC_TOD`. If set, these variables specify the locale name for the given category.
3. `LANG`.

The `setlocale()` function uses the `getenv()` function to retrieve the environment variables if the system supports the `getenv()` function. Under CICS it is not supported.

## Querying the Locale

When *locale* is set to a NULL pointer, `setlocale()` returns a string indicating the program's locale without changing it. This provides a means to query the program's current locale. To query the locale, give a NULL pointer as the second parameter. For example, to query all the categories of your locale, use a statement like the following:

```
char *string = setlocale(LC_ALL, NULL);
```

## Returned Value

Returns a pointer to the string associated with the specified category for the new locale. The string can be used on a subsequent call to restore that part of the program's locale.

**Note:** Because the string that a successful call to `setlocale()` points to may be overwritten by subsequent calls to the `setlocale()` function, you should copy the string if you plan to use it later.

On error, the `setlocale()` function returns a NULL pointer and the program's locale is not changed.

If successful, `setlocale()` function returns a string whose contents depend on the values of the *category* and *locale* arguments as shown in the following table.

*Table 31. Return String as Determined by Category and Locale Values*

Category Value	Locale Value	Return String
Specific category	NULL pointer	current-locale-name for category
	new-locale-name	new-locale-name for category
	" " (Null string)	If the environmental variables are set, new-locale-name: environment-variable-value or C.  If the environmental variables are not set, and if non-POSIX Program, then S370 or SAA.
LC_ALL	NULL pointer	One of these: <ul style="list-style-type: none"> <li>• locale-name</li> <li>• locale-name-list: locale-name1, locale-name2, ..., if different names for one or more categories.</li> </ul>
	new-locale-name	new-locale-name (same for all categories)
	" " (Null string)	One of these: <ul style="list-style-type: none"> <li>• new-locale-name: environment-variable-value or C</li> <li>• locale-name-list: environment-variable-value-list if different names for one or more categories.</li> </ul> If environmental variables are not set, and if non-POSIX program, then S370 (same for all categories).

If the string returned contains a locale name list, the names have the following order:

1. LC\_COLLATE locale-name
2. LC\_CTYPE locale-name
3. LC\_MONETARY locale-name
4. LC\_NUMERIC locale-name
5. LC\_TIME locale-name
6. LC\_TOD locale-name
7. LC\_MESSAGES locale-name
8. LC\_SYNTAX locale-name

If it fails, `setlocale()` returns a NULL pointer, and does not change the program's locale. Failure can result if:

- An incorrect *category* value is used.
- An incorrect *locale* value is used.
- The value of the environment variable used by `setlocale()` when the value of *locale* is "" is an undefined or incorrect locale name.

**Note:** If `setlocale()` is called and an application has called `pthread_create()` to create another thread, `setlocale()` returns a NULL pointer and does not change the current locale.

### Example CBC3BS07

```
/* CBC3BS07
   This example sets the locale of the program to be Fr_FR.IBM-1047 and
   prints the string that is associated with the locale.
*/
#include <stdio.h>
#include <locale.h>

char *string;

int main(void)
{
    string = setlocale(LC_ALL, "Fr_FR.IBM-1047");
    if (string != NULL)
        printf(" %s \n", string);
}
```

### CBC3BS08

```
/* EDCJS08
   This example uses setenv() to set the value of the environment variable
   LC_TIME to FRAN, calls setlocale() to set all categories to default
   values, uses setlocale() to query all categories, and uses printf() to
   print results.
*/
#include <stdio.h>
#include <stdlib.h>
#include <env.h>
#include <locale.h>

int main(void)
{
    char *string;
    setenv("LC_TIME", "FRAN", 1);
    setlocale(LC_ALL, "");
```

```

    string = setlocale(LC_ALL, NULL);
    printf("string = %s \n", string);
}

```

### Output

If the example is run with POSIX(OFF), the result of printf() is:

```
string = "S370,S370,S370,S370,FRAN,S370,S370,S370"
```

If the example is run with POSIX(ON), the result of printf() is:

```
string = "C,C,C,C,FRAN,C,C,C"
```

### Example

The following example shows euro currency support:

```

/* EUROSAMP
   This example sets the locale of the program to be
   Fr_BE.IBM-1148 and Fr_BE.IBM-1148@euro and prints
   the string associated with each locale.
*/

#include <stdio.h>
#include <locale.h>

int main(void)
{
    char *string;

    string = setlocale(LC_ALL,"Fr_BE.IBM-1148");
    if (string != NULL)
        printf("String = %s \n",string);

    string = setlocale(LC_ALL,"Fr_BE.IBM-1148@euro");
    if (string != NULL)
        printf("String = %s \n",string);
}

```

### Output

```
String = Fr_BE.IBM-1148
String = Fr_BE.IBM-1148@euro
```

### Related Information

- “locale.h” on page 33
- “getenv() — Get Value of Environment Variables” on page 515
- “nl\_langinfo() — Retrieve Locale Information” on page 866
- “localeconv() — Query Numeric Conventions” on page 756

## setlogmask() — Set the Mask for the Control Log

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <syslog.h>
```

```
int setlogmask(int maskpri);
```

### General Description

The `setlogmask()` function sets the log priority mask for the current process to *maskpri* and returns the previous mask. If the *maskpri* argument is 0 (zero), the current log mask is not modified. Calls by the current process to the `syslog()` function with a priority not set in *maskpri* are rejected. The mask for an individual priority *pri* is calculated by the macro `LOG_MASK(pri)`; The mask for all priorities up to and including *toppri* is given by the macro `LOG_UPTO(toppri)`. The default log mask allows all priorities to be logged.

### Returned Value

If successful, `setlogmask()` function returns the value of the previous mask setting.

No errors are defined.

### Related Information

- “syslog.h” on page 47
- “closelog() — Close the Control Log” on page 196
- “openlog() — Open the System Control Log” on page 882
- “syslog() — Send a Message to the Control Log” on page 1486



## setnetent() — Open the Network Information Data Set

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>
```

```
void setnetent(int stayopen);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <netdb.h>
```

```
void setnetent(int stayopen);
```

### General Description

The `setnetent()` call opens and rewinds the `tcpip.HOSTS.ADDRINFO` data set, which contains information about known networks. If the `stayopen` flag is nonzero, the `tcpip.HOSTS.ADDRINFO` remains open after each call to `setnetent()`.

You can use the **X\_ADDR** environment variable to specify a data set other than `tcpip.HOSTS.ADDRINFO`. For more information on these data sets and environment variables, see *OS/390 UNIX System Services Planning*.

**Note:** `tcpip.HOSTS.ADDRINFO` is described in *TCP/IP for MVS: Customization and Administration Guide*.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

No value is returned for the `setnetent()` function.

### Related Information

- “`endhostent()` — Work with a Host Entry” on page 308
- “`endnetent()` — Close Network Information Data Sets” on page 309
- “`getnetbyaddr()` — Get a Network Entry by Address” on page 558
- “`getnetbyname()` — Get a Network Entry by Name” on page 560
- “`getnetent()` — Get the Next Network Entry” on page 562

## set\_new\_handler() — Register a Function for set\_new\_handler()

### Standards

Standards / Extensions	C or C++	Dependencies
	C++ only	

### Format

```
#include <new.h>
```

```
typedef void(*PNH)();
PNH set_new_handler(PNH);
```

### General Description

The `set_new_handler()` function is part of the OS/390 C++ error handling mechanism. If you have registered a new-handler function with `set_new_handler()`, that new-handler function will be called by the new operator if it is unable to allocate storage. If you have not registered a new-handler function, a default new-handler function will be used instead.

The argument supplied to `set_new_handler()` is a pointer to a function with a void return type and no arguments.

In a multi-threaded environment, the new-handler function created by the issuance of a `set_new_handler()` call applies to all threads in the (POSIX) process.

You cannot issue a `set_new_handler()` from a DLL if the DLL application is to invoke the new handler function.

### Returned Value

Returns a pointer to a function with a void return type and no arguments. The function pointed to is the function that was previously called by the `set_new_handler()` function.

Refer to the *OS/390 C/C++ Language Reference* for more information about OS/390 C++ error handling, including the new operator and the `set_new_handler()` functions.

### Related Information

None

## setpeer() — Preset the Socket Peer Address

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 5.1

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>
int setpeer(int socket, struct sockaddr *address, int length, char *name);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/socket.h>
int setpeer(int socket, struct sockaddr *address, int length, char *name);
```

### General Description

The `setpeer()` call presets the peer address associated with a socket.

**Note:** Neither `AF_INET` nor `AF_UNIX` support this function.

#### Parameter Description

<i>socket</i>	The socket descriptor.
<i>address</i>	The address of the socket peer.
<i>length</i>	The length of the socket address.
<i>name</i>	The name of a field indicating the conditions of the peer request.

#### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

This function always returns a value of `-1`.

#### Error Code Description

<code>EINVAL</code>	The request is invalid or not supported.
---------------------	--

### Related Information

None

## setpgid() — Set Process Group ID for Job Control

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
int setpgid(pid_t pid, pid_t pgid);
```

### General Description

Sets the process group ID (PGID) of a process within the session of the calling process, so you can reassign a process to a different process group, or start a new process group with the specified process as its group leader.

*pid\_t pid* is the process ID (PID) of the process whose PGID you want to change. This must either be the caller of `setpgid()` or one of its children, and it must be in the caller's session. It cannot be the PID of a session leader. If *pid* is zero, the system uses the PID of the process calling `setpgid()`.

*pid\_t pgid* is the new PGID you want to assign to the process identified by *pid*. If *pgid* indicates an existing process group, it must be in the caller's session. If *pgid* is zero, the system uses the PID of the process indicated by *pid* as the ID for the new process group. The new group is created in the caller's session.

### Returned Value

If successful, `setpgid()` returns zero. If unsuccessful, it returns a value of `-1` and sets `errno` to one of the following:

- |        |   |
|--------|---|
| EACCES | The value of <i>pid</i> matches the PID of a child of the calling process, but the child has successfully run one of the EXEC functions.  |
| EINVAL | <i>pgid</i> is less than zero or has some other unsupported value.  |
| EPERM  | The caller cannot change the PGID of the specified process. Some possible reasons are: <ul style="list-style-type: none"> <li>• The specified process is a session leader.</li> <li>• <i>pid</i> matches the PID of a child of the calling process, but the child is not in the same session as the caller.</li> <li>• <i>pgid</i> does not match the PID of the process specified by <i>pid</i>, and it does not match the PGID of any other process in the caller's session.</li> </ul> |
| ESRCH  | <i>pid</i> does not match the PID of the calling process or any of its children.  |

## Example

### CBC3BS09

```

/* CBC3BS09
   This example sets the PGID.
*/
#define _POSIX_SOURCE
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

main() {
    pid_t pid;
    int p1[2], p2[2];
    char c='?';

    if (pipe(p1) != 0)
        perror("pipe() #1 error");
    else if (pipe(p2) != 0)
        perror("pipe() #2 error");
    else
        if ((pid = fork()) == 0) {
            printf("child's process group id is %d\n", (int) getpgrp());
            write(p2[1], &c, 1);
            puts("child is waiting for parent to complete task");
            read(p1[0], &c, 1);
            printf("child's process group id is now %d\n", (int) getpgrp());
            exit(0);
        }
        else {
            printf("parent's process group id is %d\n", (int) getpgrp());
            read(p2[0], &c, 1);
            printf("parent is performing setpgid() on pid %d\n", (int) pid);
            if (setpgid(pid, 0) != 0)
                perror("setpgid() error");
            write(p1[1], &c, 1);
            printf("parent's process group id is now %d\n", (int) getpgrp());
            sleep(5);
        }
    }
}

```

## Output

```

parent's process group id is 5767174
child's process group id is 5767174
parent is performing setpgid() on pid 131084
parent's process group id is now 5767174
child is waiting for parent to complete task
child's process group id is now 131084

```

## Related Information

- “unistd.h” on page 53
- “exec Functions” on page 322
- “getpgrp() — Get the Process Group ID” on page 571
- “setpgrp() — Set Process Group ID” on page 1256
- “setsid() — Create Session, Set Process Group ID” on page 1268
- “tcsetpgrp() — Set the Foreground Process Group ID” on page 1546

## setpgrp() — Set Process Group ID

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
```

```
pid_t setpgrp(void);
```

### General Description

If the calling process is not already a session leader, `setpgrp()` sets the process group ID of the calling process to the process ID of the calling process. If a new process group is created, it is created within the session of the calling process.

### Returned Value

If successful, `setpgrp()` returns the new process group ID.

If unsuccessful, `setpgrp()` returns -1 and the error value is placed in `errno`. The following are the possible values of `errno`:

`EPERM`      The calling process is a session leader.

### Related Information

- “`unistd.h`” on page 53
- “`exec Functions`” on page 322
- “`getpgrp()` — Get the Process Group ID” on page 571
- “`setpgid()` — Set Process Group ID for Job Control” on page 1254
- “`setsid()` — Create Session, Set Process Group ID” on page 1268
- “`tcsetpgrp()` — Set the Foreground Process Group ID” on page 1546

# setpriority() — Set Process Scheduling Priority

## Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

## Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/resource.h>

int setpriority(int which, id_t who, int priority);
```

## General Description

setpriority() sets the scheduling priority of a process, process group or user.

Processes are specified by the values of the *which* and *who* arguments. The *which* argument may be any one of the following set of symbols defined in the *sys/resource.h* include file:

- PRIO\_PROCESS  
indicates that the *who* argument is to be interpreted as a process ID
- PRIO\_PGRP  
indicates that the *who* argument is to be interpreted as a process group ID
- PRIO\_USER  
indicates that the *who* argument is to be interpreted as a user ID

The *who* argument specifies the ID (process, process group, or user). A 0 (zero) value for the *who* argument specifies the current process, process group or user ID.

The *priority* argument specifies the scheduling priority. It is specified as a signed integer in the range, -20 to 19. Negative priorities cause more favorable scheduling. The default priority is 0. If the value specified to setrlimit() is less than the system's lowest supported priority value, the system's lowest supported value is used; if it is greater than the system's highest supported value, the system's highest supported value is used. The setting of a process' scheduling priority value has the equivalent effect on a process' nice value, since they both represent the process' relative CPU priority. For example, setting one's scheduling priority value to its maximum value (19) has the equivalent effect of increasing one's nice value to its maximum value ((2\*NZERO)-1), and will be reflected on the nice(), getpriority() and setpriority() functions.

If more than one process is specified, setpriority() sets the priorities of all of the specified processes to the specified value.

Only a process with appropriate privilege can lower its priority.

**Returned Value**

If successful, `setpriority()` return a value of 0.

If unsuccessful, `setpriority()` returns -1, and returns the error value in `errno`. The following are the possible values of `errno`:

ESRCH	No process could be located using the <i>which</i> and <i>who</i> argument values specified.
EINVAL	The symbol specified in the <i>which</i> argument was not recognized, or the value of the <i>who</i> argument is not a valid process ID, process group ID or user ID.
EPERM	A process was located, but neither the real nor effective user ID of the executing process match the effective user ID of the process whose priority is to be changed.
EACCES	The priority is being changed to a lower value and the current process does not have the appropriate privilege.
ENOSYS	The system does not support this function.

**Related Information**

- “`sys/resource.h`” on page 48
- “`getpriority()` — Get Process Scheduling Priority” on page 578
- “`nice()` — Change Priority of a Process” on page 864



## setprotoent() — Open the Protocol Information Data Set

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>
```

```
void setprotoent(int stayopen);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <netdb.h>
```

```
void setprotoent(int stayopen);
```

### General Description

The `setprotoent()` call opens and rewinds the */etc/protocol* or the *tcpip.ETC.PROTO* data set. If the *stayopen* flag is nonzero, the */etc/protocol* or the *tcpip.ETC.PROTO* data set remains open after each call.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

No value is returned for the `setprotoent()` function.

### Related Information

- “`endprotoent()` — Work with a Protocol Entry” on page 310
- “`getprotobyname()` — Get a Protocol Entry by Name” on page 580
- “`getprotobynumber()` — Get a Protocol Entry by Number” on page 582
- “`getprotoent()` — Get the Next Protocol Entry” on page 584

## **setpwent() — Reset User Database Search**

The information for this function is included in “endpwent() — User Database Functions” on page 311.

## setregid() — Set Real and Effective Group IDs

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
int setregid(gid_t rgid, gid_t egid);
```

### General Description

The `setregid()` function sets the real and/or effective GIDs for the calling process to the values specified by the input real and effective GID values. If a specified value is equal to -1, the corresponding real or effective GID of the calling process is left unchanged.

A process with appropriate privileges can set the real and effective GID to any valid GID value. An unprivileged process can only set the effective GID if the EGID argument is equal to either the real, effective, or saved GID of the process. An unprivileged process can only set the real GID if the RGID argument is equal to either the real, effective, or saved GID of the process.

If the `setregid()` function is issued from multiple tasks within one address space, use synchronization to ensure that the `setregid()` functions are not performed concurrently. The execution of `setregid()` function concurrently within one address space can yield unpredictable results.

The `setregid()` function does not change any supplementary GIDs of the calling process.

### Returned Value

Upon successful completion, 0 is returned. Otherwise, -1 is returned and `errno` is set to indicate the error and neither of the group IDs will be changed.

The `setregid()` function will fail if:

- |             |  |
|-------------|--|
| EINVAL      | The value of the <i>rgid</i> or <i>egid</i> argument is invalid or out-of-range.   |
| EPERM       | The processes does not have appropriate privileges and a change other than changing the real group ID to the saved set-group-ID, or changing the effective group ID to the real group ID or the saved group ID, was requested. |
| EMVSSAF2ERR | The SAF call IRRSSU00 incurred an error.   |

### Related Information

- “exec Functions” on page 322
- “getuid() — Get the Real User ID” on page 618
- “setreuid() — Set Real and Effective User IDs” on page 1262
- “setuid() — Set the Effective User ID” on page 1278

## setreuid() — Set Real and Effective User IDs

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);
```

### General Description

The `setreuid()` function sets the real and/or effective UIDs for the calling process to the values specified by the input real and effective UID values. If a specified value is equal to -1, the corresponding real or effective UID of the calling process is left unchanged.

A process with appropriate privileges can set the real and effective UID to any valid UID value. An unprivileged process can only set the effective UID if the EUID argument is equal to either the real, effective, or saved UID of the process. An unprivileged process can only set the real UID if the RUID argument is equal to either the real, effective, or saved UID of the process.

The `setreuid()` function is not supported from an address space running multiple processes, since it would cause all processes in the address space to have their security environment changed unexpectedly.

`setreuid()` can be used by daemon processes to change the identity of a process in order for the process to be used to run work on behalf of a user. In UNIX, changing the identity of a process is done by changing the real and effective UIDs and the auxiliary groups. In order to change the identity of the process on MVS completely, it is necessary to also change the MVS security environment. The identity change will only occur if the EUID value is specified, changing just the real UID will have no effect on the MVS environment.

The `setreuid()` function invokes MVS SAF services to change the MVS identity of the address space. The MVS identity that is used is determined as follows:

- If an MVS user ID is already known by the kernel from a previous call to a kernel function (for example, `getpwnam()`) and the UID for this user ID matches the UID specified on the `setreuid()` call, then this user ID is used.
- For nonzero target UIDs, if there is no saved user ID or the UID for the saved user ID does not match the UID requested on the `setreuid()` call, the `setreuid()` function queries the security database (for example, using `getpwnam()`) to retrieve a user ID. The retrieved user ID is then used.
- If the target UID is 0 and a user ID is not known, the `setreuid()` function always sets the MVS user ID to BPXROOT or the value specified on the SUPERUSER parm in sysparms. BPXROOT is set up during system initialization as a super-user with a UID of 0. The BPXROOT user ID is not defined to the BPX.DAEMON FACILITY class profile. This special processing is necessary to prevent a superuser from gaining daemon authority.

- A nondaemon superuser that attempts to set a user ID to a daemon superuser UID fails with an EPERM.

When the MVS identity is changed, the auxiliary list of groups is also set to the list of groups for the new user ID.

If the setreuid() function is issued from multiple tasks within one address space, use synchronization to ensure that the setreuid() functions are not performed concurrently. The execution of setreuid() function concurrently within one address space can yield unpredictable results.

### Returned Value

Upon successful completion, 0 is returned. Otherwise, -1 is returned and errno is set to indicate the error and neither of the group IDs will be changed.

The setreuid() function will fail if:

- |        |  |
|--------|--|
| EINVAL | The value of the <i>rgid</i> or <i>egid</i> argument is invalid or out-of-range.   |
| EPERM  | The processes does not have appropriate privileges and a change other than changing the real group ID to the saved set-group-ID, or changing the effective group ID to the real group ID or the saved group ID, was requested. |

EMVSSAF2ERR

The SAF call IRRSSU00 incurred an error.

### Related Information

- “exec Functions” on page 322
- “getuid() — Get the Real User ID” on page 618
- “seteuid() — Set the Effective User ID” on page 1218
- “setuid() — Set the Effective User ID” on page 1278

## setrlimit() — Control Maximum Resource Consumption

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/resource.h>
```

```
int setrlimit(int resource, const struct rlimit *rlp);
```

### General Description

The `setrlimit()` function sets resource limits for the calling process. A resource limit is a pair of values; one specifying the current (soft) limit, the other a maximum (hard) limit.

The soft limit may be modified to any value that is less than or equal to the hard limit. For certain *resource* values, (`RLIMIT_CPU`, `RLIMIT_NOFILE`, `RLIMIT_AS`), the soft limit cannot be set lower than the existing usage.

The hard limit may be lowered to any value that is greater than or equal to the soft limit. The hard limit can be raised only by a process which has superuser authority. Both the soft limit and hard limit can be changed by a single call to `setrlimit()`.

The value `RLIM_INFINITY` defined in `<sys/resource.h>`, is considered to be larger than any other limit value. If a call to `getrlimit()` returns `RLIM_INFINITY` for a resource, it means the implementation does not enforce limits on that resource. Specifying `RLIM_INFINITY` as any resource limit values on a successful call to `setrlimit()` inhibits enforcement of that resource limit.

The *resource* argument specifies which resource to set the hard and/or soft limits for, and may be one of the following values:

#### RLIMIT\_CORE

The maximum size of a dump of memory (in bytes) allowed for the process. A value of 0 (zero) prevents file creation. Dump file creation will stop at this limit.

#### RLIMIT\_CPU

The maximum amount of CPU time (in seconds) allowed for the process. If the limit is exceeded, a `SIGXCPU` signal is sent to the process and the process is granted a small CPU time extension to allow for signal generation and delivery. If the extension is used up, the process is terminated with a `SIGKILL` signal. An attempt to set the CPU limit lower than that already used will result in an `EINVAL` error.

#### RLIMIT\_DATA

The maximum size of the break value for the process, in bytes. In this implementation, this resource always has a hard and soft limit value of `RLIM_INFINITY`. A call to `setrlimit()` to set this resource to any value other than `RLIM_INFINITY` will fail with an error of `EINVAL`.

**RLIMIT\_FSIZE**

The maximum file size (in bytes) allowed for the process. A value of 0 (zero) prevents file creation. If the size is exceeded, a SIGXFSZ signal is sent to the process. If the process is blocking, catching, or ignoring SIGXFSZ, continued attempts to increase the size of a file beyond the limit will fail with an errno of EFBIG.

**RLIMIT\_NOFILE**

The maximum number of open file descriptors allowed for the process. This number is one greater than the maximum value that may be assigned to a newly created descriptor. (That is, it is one-based.) Any function that attempts to create a new file descriptor beyond the limit will fail with an EMFILE errno. An attempt to set the open file descriptors limit lower than that already used will result in an EINVAL errno.

**RLIMIT\_STACK**

The maximum size of the stack for a process, in bytes. Note that in OS/390 UNIX services, the stack is a per-thread resource. In this implementation, this resource always has a hard and soft limit value of RLIM\_INFINITY. A call to setrlimit() to set this resource to any value other than RLIM\_INFINITY will fail with an errno of EINVAL.

**RLIMIT\_AS** The maximum address space size for the process, in bytes. If the limit is exceeded, malloc() and mmap() functions will fail with an errno of ENOMEM. Also, automatic stack growth will fail.

The *r/p* argument points to a `rlimit` structure. This structure contains the following members:

<code>rlim_cur</code>	The current (soft) limit
<code>rlim_max</code>	The maximum (hard) limit

Refer to the `<sys/resource.h>` header for more detail.

The resource limit values are propagated across exec and fork.

**Special Behavior for OS/390 UNIX Services**

An exception exists for exec processing in conjunction with daemon support. If a daemon process invokes exec and it had previously invoked setuid() prior to exec, the RLIMIT\_CPU, RLIMIT\_AS, RLIMIT\_CORE, RLIMIT\_FSIZE, and RLIMIT\_NOFILE limit values are set based on the limit values specified in the kernel parmlib member BPXPRMxx.

For processes which are not the only process within an address space, the RLIMIT\_CPU and RLIMIT\_AS limits are shared with all the processes within the address space. For RLIMIT\_CPU, when the soft limit is exceeded, action will be taken on the first process within the address space. If the action is termination, all processes within the address space will be terminated.

In addition to the RLIMIT\_CORE limit values, the dump file defaults are set by SYSMDUMP defaults. Refer to the *OS/390 MVS Initialization and Tuning Reference* for more information on setting up SYSMDUMP defaults via the IEADMR00 parmlib member.

Core dumps are taken in 4160 byte increments. Therefore, RLIMIT\_CORE values affect the size of core dumps in 4160 byte increments. For example, if the RLIMIT\_CORE soft limit value is 4000, the dump will contain no data. If the RLIMIT\_CORE soft limit value is 8000, the maximum size of a core dump is 4160 bytes.

When setting RLIMIT\_NOFILE, the hard limit cannot exceed the system defined limit of 65535.

### Returned Value

Upon successful completion, a 0 is returned. Otherwise, a -1 is returned, and the error value in `errno` is set. The following are the possible values of `errno`:

- |        |  |
|--------|--|
| EINVAL | An invalid <i>resource</i> was specified, or the soft limit to set exceeds the hard limit to set, the soft limit to set is below the current usage, or the resource does not allow any value other than RLIM_INFINITY. |
| EPERM  | The limit specified to <code>setrlimit()</code> would have raised the maximum limit value, and the calling process does not have appropriate privileges.   |

### Related Information

- “`stropts.h`” on page 46
- “`sys/resource.h`” on page 48
- “`getrlimit()` — Control Maximum Resource Consumption” on page 591
- “`brk()` — Change Space Allocation” on page 133
- “`rexec()` — Execute Commands One at a Time on a Remote Host” on page 1143
- “`fork()` — Create a New Process” on page 422
- “`getdtablesize()` — Get the File Descriptor Table Size” on page 513
- “`malloc()` — Reserve Storage Block” on page 786
- “`open()` — Open a File” on page 872
- “`sigaltstack()` — Set and/or Get Signal Alternate Stack Context” on page 1314
- “`sysconf()` — Determine System Configuration Options” on page 1483
- “`ulimit()` — Get/Set Process File Size Limits” on page 1645



## setservent() — Open the Network Services Information Data Set

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <netdb.h>
```

```
void setservent(int stayopen);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <netdb.h>
```

```
void setservent(int stayopen);
```

### General Description

The `setservent()` call opens and rewinds the `/etc/services` or the `tcpip.ETC.SERVICES` data set. For more information on `/etc/services` or the `tcpip.ETC.SERVICES` data set, see *TCP/IP for MVS: Customization and Administration Guide*. If the `stayopen` flag is nonzero, the `tcpip.ETC.SERVICES` data set remains open after each call.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

The `setservent()` function returns no value.

### Related Information

- “endservent() — Close Network Services Information Data Sets” on page 312
- “getservbyname() — Get a Server Entry by Name” on page 597
- “getservent() — Get the Next Service Entry” on page 601

## setsid() — Create Session, Set Process Group ID

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
pid_t setsid(void);
```

### General Description

Creates a new session with the calling process as its session leader. The caller becomes the process group leader of a new process group. The calling process must not be a process group leader already. The caller does not have a controlling terminal.

The process group ID (PGID) of the new process group is equal to the process ID (PID) of the caller. The caller starts as the only process in the new process group and in the new session.

### Returned Value

If successful, setsid() returns the value of the caller's new PGID. If unsuccessful, it returns the value `-1` and sets `errno` to `EPERM`, which indicates either that the caller is already a process group leader, or that the caller's PID matches the PGID of some other process.

### Example

#### CBC3BS10

```
/* CBC3BS10
   This example creates a new session.
*/
#define _POSIX_SOURCE
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

main() {
    pid_t pid;
    int p[2];
    char c='?';

    if (pipe(p) != 0)
        perror("pipe() error");
    else
        if ((pid = fork()) == 0) {
            printf("child's process group id is %d\n", (int) getpgrp());
            write(p[1], &c, 1);
            setsid();
            printf("child's process group id is now %d\n", (int) getpgrp());
            exit(0);
        }
    else {
```

```

        printf("parent's process group id is %d\n", (int) getpgrp());
        read(p[0], &c, 1);
        sleep(5);
    }
}

```

### Output

```

child's process group id is 262152
child's process group id is now 262150
parent's process group id is 262152

```

### Related Information

- “sys/types.h” on page 49
- “exec Functions” on page 322
- “\_exit() — End a Process and Bypass the Cleanup” on page 332
- “fork() — Create a New Process” on page 422
- “getpid() — Get the Process ID” on page 573
- “kill() — Send a Signal to a Process” on page 728
- “setpgid() — Set Process Group ID for Job Control” on page 1254
- “sigaction() — Examine or Change a Signal Action” on page 1295

setsockopt() — Set Options Associated with a Socket

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

Format

X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int setsockopt(int socket, int level, int option_name, const void
*option_value,
               size_t option_length);
```

Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/types.h>
#include <sys/socket.h>

int setsockopt(int socket, int level, int option_name, char
*option_value,
               int *option_length);
```

ip\_mreq Structure

To include the ip\_mreq structure in your program, add the following code:

```
#define _XOPEN_SOURCE 500
#include <netinet/in.h>
```

General Description

The setsockopt() call sets options associated with a socket. Options can exist at multiple protocol levels; they are always present at the highest socket level.

Parameter	Description
socket	The socket descriptor.
level	The level for which the option is being set. Only SOL_SOCKET and IPPROTO_IP are supported..
option_name	The name of a specified socket option.
option_value	The pointer to option data.
option_length	The length of the option data.

When manipulating socket options, you must specify the level at which the option resides and the name of the option. To manipulate options at the socket or IP level, the level parameter must be set to SOL\_SOCKET or IPPROTO\_IP as defined in **sys/socket.h**. To manipulate options at any other level, such as the TCP level, supply the appropriate protocol number for the protocol controlling the option. Currently, only the SOL\_SOCKET and IPPROTO\_IP levels are supported. The getprotobyname() call can be used to return the protocol number for a named protocol.

The *option\_value* and *option\_length* parameters are used to pass data used by the particular set command. The *option\_value* parameter points to a buffer containing the data needed by the set command. The *option\_value* parameter is optional and can be set to the NULL pointer, if data is not needed by the command. The *option\_length* parameter must be set to the size of the data pointed to by *option\_value*.

All of the socket-level options except SO\_LINGER expect *option\_value* to point to an integer and *option\_length* to be set to the size of an integer. When the integer is nonzero, the option is enabled. When it is zero, the option is disabled. The SO\_LINGER option expects *option\_value* to point to a **linger** structure, as defined in **sys/socket.h**. This structure is defined in the following example:

```
struct linger
{
    int    l_onoff;           /* option on/off */
    int    l_linger;         /* linger time */
};
```

The *l\_onoff* field is set to 0 if the SO\_LINGER option is begin disabled. A nonzero value enables the option. The *l\_linger* field specifies the amount of time to linger on close. The units of *l\_linger* are seconds.

The following options are recognized at the IP level:

#### Option Description

##### IP\_MULTICAST\_TTL

Sets the IP time-to-live of outgoing multicast datagrams. Default value is 1 (that is, multicast only to the local subnet). The TTL value is passed in as *u\_char*.

##### IP\_MULTICAST\_LOOP

Enables/disables loopback of outgoing multicast datagrams. Default is enable. When it is enabled, multicast applications that have joined the outgoing multicast group can receive a copy of the multicast datagrams destined for that address/port pair. The loopback indicator is passed in as *u\_char*. 0 is specified to disable loopback. 1 is specified to enable loopback.

##### IP\_MULTICAST\_IF

Sets the interface for sending outbound multicast datagrams from this socket application. Multicast datagrams will be transmitted only on one interface at a time. An IP address is passed using struct *in\_addr*.

If INADDR\_ANY is specified for the interface address passed, a default interface will be chosen as follows:

- If 224.0.0.0 was specified on GATEWAY statement, use that interface.
- If DEFAULTNET was specified and is multicast capable, use that interface.

##### IP\_ADD\_MEMBERSHIP

This option is used to join a multicast group on a specific interface (an interface has to be specified with this option). Only applications that want to receive multicast datagrams need to join multicast groups. Applications that only transmit will not need to do so.

The multicast IP address and the interface IP address will be passed in the following structure available in **netinet/in.h**:

```
struct ip_mreq
{
    struct in_addr imr_multiaddr; /* IP multicast addr of group */
    struct in_addr imr_interface; /* local IP addr of interface */
};
```

If **INADDR\_ANY** is specified on the interface address of the **mreq** structure passed, a default interface will be chosen as follows:

- If the group address specified in the **mreq** structure was specified on a **GATEWAY** statement, use that interface.
- If **224.0.0.0** was specified on **GATEWAY** statement, use that interface.
- If **DEFAULTNET** was specified and is multicast capable use that interface.

#### IP\_DROP\_MEMBERSHIP

This option is used to leave a multicast group.

The multicast IP address and the interface IP address will be passed in the following structure available in **netinet/in.h**:

```
struct ip_mreq
{
    struct in_addr imr_multiaddr; /* IP multicast addr of group */
    struct in_addr imr_interface; /* local IP addr of interface */
};
```

If **INADDR\_ANY** is specified on the interface address of the **mreq** structure passed, the system will drop the first group that matches the group (class D) address without regard to the interface.

The following options are recognized at the socket level:

Option	Description
SO_BROADCAST	Toggles the ability to broadcast messages. If enabled, this option allows the application program to send broadcast messages over <i>socket</i> , if the interface specified in the destination supports broadcasting of packets. This option has no meaning for stream sockets.
SO_DEBUG	Turns on recording of debugging information. This option enables or disables debugging in the underlying protocol modules. This option takes an int value.
SO_KEEPALIVE	Toggles the TCP keep-alive mechanism for a stream socket. When activated, the keep-alive mechanism periodically sends a packet on an otherwise idle connection. If the remote TCP does not respond to the packet or to retransmissions of the packet, the connection is terminated with the error ETIMEDOUT.
SO_LINGER	Lingers on close if data is present. When this option is enabled and there is unsent data present when <code>close()</code> is called, the calling application program is blocked during the <code>close()</code> call, until the data is transmitted or the connection has timed out. If this option is disabled, the TCP/IP address space waits to try to send the data. Although the data transfer is usually successful, it cannot be guaranteed, because the TCP/IP address space waits only a finite amount of time trying to send the data. The

`close()` call returns without blocking the caller. This option has meaning only for stream sockets.

**SO\_OOBINLINE** Toggles the reception of out-of-band data. When this option is enabled, it causes out-of-band data to be placed in the normal data input queue as it is received, making it available to `recv()`, `recvfrom()`, and `recvmsg()` without having to specify the `MSG_OOB` flag in those calls. When this option is disabled, it causes out-of-band data to be placed in the priority data input queue as it is received, making it available to `recv()`, `recvfrom()`, and `recvmsg()` only by specifying the `MSG_OOB` flag in those calls. This option has meaning only for stream sockets.

**\_SO\_PROPAGATEUSERID**

Toggles propagating a user ID (UID) over a socket. When enabled, user (UID) information is extracted from the system when the `connect()` function is invoked and presented over the socket when the `accept()` function is invoked.

**SO\_RCVBUF** Sets receive buffer size. This option takes an int value.

**SO\_REUSEADDR** Toggles local address reuse. When enabled, this option allows local addresses that are already in use to be bound. This alters the normal algorithm used in the `bind()` call. The system checks at connect time to ensure that the local address and port do not have the same foreign address and port. The error `EADDRINUSE` is returned if the association already exists.

**SO\_SNDBUF** Sets send buffer size. This option takes an int value.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

The value 0 indicates success; the value -1 indicates an error. The value of the error code indicates the specific error.

### Error Code Description

**EBADF** The *socket* parameter is not a valid socket descriptor.

**EFAULT** Using *option\_value* and *option\_length* parameters would result in an attempt to access storage outside the caller's address space.

**EINVAL** The specified option is invalid at the specified socket level or the socket has been shut down.

**ENOBUFS** Insufficient system resources are available to complete the call.

**ENOPROTOOPT**

The *option\_name* parameter is unrecognized, or the *level* parameter is not `SOL_SOCKET`.

**ENOSYS** The function is not implemented. You attempted to use a function that is not yet available.

**ENOTSOCK**

The descriptor is for a file, not for a socket.

## Example

The following are examples of the `setsockopt()` call. See “`getsockopt()` — Get the Options Associated with a Socket” on page 606 for examples of how the `getsockopt()` options set are queried.

```
int rc;
int s;
int option_value;
struct linger l;
int setsockopt(int s, int level, int option_name, char *option_value, int option_len);
:
/* I want out of band data in the normal input queue */
option_value = 1;
rc = setsockopt(s, SOL_SOCKET, SO_OOBINLINE, (char *) &option_value, sizeof(int));

:
/* I want to linger on close */
l.l_onoff = 1;
l.l_linger = 100;
rc = setsockopt(s, SOL_SOCKET, SO_LINGER, (char *) &l, sizeof(l));
```

## Related Information

- “`fcntl()` — Control Open File Descriptors” on page 350
- “`getprotobyname()` — Get a Protocol Entry by Name” on page 580
- “`getsockopt()` — Get the Options Associated with a Socket” on page 606
- “`ioctl()` — Control Device” on page 672
- “`socket()` — Create a Socket” on page 1371



## setstate() — Change Generator for random()

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>
```

```
char *setstate(const char *state);
```

### General Description

The `setstate()` function allows switching between state arrays used by the `random()` function once a state has been initialized. The array defined by the *state* argument is used for further random-number generation by the calling thread until `initstate()` is called or `setstate()` is called again. The `setstate()` function returns a pointer to the previous state array.

After initialization, a state array can be restarted at a different point by calling `setstate()` with the desired state, followed by `srandom()` with the desired seed.

### Returned Value

If `setstate()` is successful, it returns a pointer to the previous state array. Otherwise, a null pointer is returned. The function will fail and write a message to standard error if it detects that the state information has been damaged.

### Related Information

- “`stdlib.h`” on page 45
- “`random()` — A Better Random-Number Generator” on page 1079
- “`initstate()` — Initialize Generator for Random()” on page 670
- “`srandom()` — Use Seed to Initialize Generator for random()” on page 1400
- “`drand48()` — Pseudo-random Number Generator” on page 286
- “`rand()` — Generate Random Number” on page 1077

## set\_terminate() — Register a Function for terminate()

### Standards

Standards / Extensions	C or C++	Dependencies
	C++ only	

### Format

```
#include <terminate.h>
```

```
typedef void(*PFV)();
PFV set_terminate(PFV);
```

### General Description

Is part of the OS/390 C++ error handling mechanism. The argument supplied to `set_terminate()` is a pointer to a function with a void return type and no arguments. The function specified will be called by the `terminate()` function.

Note that the function registered for `terminate()` should not return to `terminate()`. If it does, `terminate()` will call `abort()`.

In a multi-threaded environment, the `terminate` function created by the issuance of a `set_terminate()` call applies to all threads in the (POSIX) process. If a thread throws an exception which is not caught by that thread of execution, then `terminate()` is called. The default `terminate()` action calls `abort()` which by default cause a SIGABRT signal. If there is no signal handler, then SIGABRT terminates the process. You can override this with a thread-level termination by supplying a function which invokes `pthread_exit()` as a `terminate` function. This terminates the thread but not the process.

### Returned Value

Returns a pointer to a function with a void return type and no arguments. The function pointed to is the function that was previously registered by `set_terminate()`.

Refer to the *OS/390 C/C++ Language Reference* for more information about OS/390 C++ exception handling, including the `set_terminate()` function.

### Related Information

- “`terminate()` — Terminate After Failures in C++ Error Handling” on page 1560

## \_SET\_THLIIPADDR() — Set the Client's IP Address

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R5

### Format

```
#include <__ussos.h>
```

```
int _SET_THLIIPADDR(ln,ipaddr);
```

### General Description

The \_SET\_THLIIPADDR() macro provides a way for daemons to set a client's IP address.

\_SET\_THLIIPADDR() takes the following arguments:

*ln*            The length of the IP address as specified by *ipaddr*. The IP address length can be between 1 and 16 inclusive. The argument is specified as an unsigned int.

*ipaddr*       Pointer to the IP address.

### Usage Notes

The intent of the \_SET\_THLIIPADDR() macro is to provide a way for daemons to set a client's IP address for security facility authorization (SAF) exits when performing security related functions.

### Restrictions

Results are unpredictable if \_SET\_THLIIPADDR() is issued outside of the OS/390 UNIX environment.

### Returned Value

\_SET\_THLIIPADDR() returns a non-zero value if the client's IP address is set.

\_SET\_THLIIPADDR() returns a zero value and does not set the client's IP address when:

- The base level of OS/390 UNIX is not OS/390 R5.
- The setting of the IP address is not supported.
- The length of the IP address is less than 1 or greater than 16.

## setuid() — Set the Effective User ID

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
int setuid(uid_t uid);
```

### General Description

Sets the real, effective, or saved set user IDs (UIDs) for the current process to *uid*.

If *uid* is the same as the real UID of the process, `setuid()` always succeeds and sets the effective UID to be the same as the real UID. If `_POSIX_SAVED_IDS` is defined in the `sys/types.h` header file, `setuid()` also sets the saved set UID to *uid*.

If *uid* is not the same as the real UID of the process, `setuid()` succeeds only if the process has appropriate privileges. If the process has such privileges, `setuid()` sets the real group ID (UID), effective UID, and saved set UID to *uid*.

The `setuid()` function is not supported from an address space running multiple processes, since it would cause all processes in the address space to have their security environment changed unexpectedly.

`setuid()` can be used by daemon processes to change the identity of a process in order for the process to be used to run work on behalf of a user. In UNIX, changing the identity of a process is done by changing the real and effective UIDs and the auxiliary groups. In order to change the identity of the process on MVS completely, it is necessary to also change the MVS security environment. The identity change will only occur if the EUID value is specified, changing just the real UID will have no effect on the MVS environment.

The `setuid()` function invokes MVS SAF services to change the MVS identity of the address space. The MVS identity that is used is determined as follows:

- If an MVS user ID is already known by the kernel from a previous call to a kernel function (for example, `getpwnam()`) and the UID for this user ID matches the UID specified on the `setuid()` call, then this user ID is used.
- For nonzero target UIDs, if there is no saved user ID or the UID for the saved user ID does not match the UID requested on the `setuid()` call, the `setuid()` function queries the security database (for example, using `getpwnam()`) to retrieve a user ID. The retrieved user ID is then used.
- If the target UID is 0 and a user ID is not known, the `setuid()` function always sets the MVS user ID to BPXROOT or the value specified on the SUPERUSER parm in sysparms. BPXROOT is set up during system initialization as a super-user with a UID of 0. The BPXROOT user ID is not defined to the

BPX.DAEMON FACILITY class profile. This special processing is necessary to prevent a superuser from gaining daemon authority.

**Note:** When running under UID 0, some servers will issue `setuid(0)` in order to test whether they are running UID 0. The problem with this is that the `setuid` function will change the `userid` to `BPXROOT` which will likely cause the daemon to fail on subsequent function requests.

- When changing from a non-zero UID to a zero UID, the MVS `userid` is not changed. When using the `su` shell command to become a superuser, the shell retains the original MVS `userid`.
- A nondaemon superuser that attempts to set a user ID to a daemon superuser UID fails with an `EPERM`.

When the MVS identity is changed, the daemon must make a call to `initgroups()` to set the auxiliary list of groups to the list of groups for the new user ID.

If the `setuid()` function is issued from multiple tasks within one address space, use synchronization to ensure that the `setuid()` functions are not performed concurrently. The execution of `setuid()` function concurrently within one address space can yield unpredictable results.

## Returned Value

If successful, `setuid()` returns zero. If unsuccessful, it returns the value `-1` and sets `errno` to one of the following:

<code>EAGAIN</code>	The process is currently not able to change UIDs.
<code>EINVAL</code>	The value of <i>uid</i> is incorrect.
<code>EPERM</code>	The process does not have appropriate privileges to set the UID to <i>uid</i> .

## Example

### CBC3BS11

```
/* CBC3BS11
   This example changes the effective UID.
*/
#define _POSIX_SOURCE
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

main() {
    printf("prior to setuid(), uid=%d, effective uid=%d\n",
           (int) getuid(), (int) geteuid());
    if (setuid(25) != 0)
        perror("setuid() error");
    else
        printf("after setuid(), uid=%d, effective uid=%d\n",
               (int) getuid(), (int) geteuid());
}
```

## Output

```
prior to setuid(), uid=0, effective uid=0
after setuid(), uid=25, effective uid=25
```

### Related Information

- “sys/types.h” on page 49
- “exec Functions” on page 322
- “geteuid() — Get the Effective User ID” on page 519
- “getuid() — Get the Real User ID” on page 618
- “seteuid() — Set the Effective User ID” on page 1218
- “setgid() — Set the Group ID” on page 1220
- “setreuid() — Set Real and Effective User IDs” on page 1262

## set\_unexpected() — Register a Function for unexpected()

### Standards

Standards / Extensions	C or C++	Dependencies
	C++ only	

### Format

```
#include <unexpected.h>
```

```
typedef void(*PFV)();
PFV set_unexpected(PFV);
```

### General Description

Is part of the OS/390 C++ error handling mechanism. The argument supplied to `set_unexpected()` is a pointer to a function with a void return type and no arguments. The function specified will be called by the `unexpected()` function.

Note that the function registered for `unexpected()` should not return to `unexpected()`. If it does, `unexpected()` will call `terminate()`.

In a multi-threaded environment, the `unexpected` function created by the issuance of a `set_unexpected()` call applies to all threads in the (POSIX) process.

You cannot issue a `set_unexpected()` from a DLL if the DLL application is to invoke the `unexpected` function.

### Returned Value

Returns a pointer to a function with a void return type and no arguments. The function pointed to is the function that was previously called by the `unexpected()` function.

Refer to the *OS/390 C/C++ Language Reference* for more information about OS/390 C++ exception handling, including the `set_unexpected()` function.

### Related Information

- “`unexpected()` — Handle Exception Not Listed in Exception Specification” on page 1654

## setutxent() — Reset to Start of utmpx Database

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <utmpx.h>
```

```
void setutxent(void);
```

### General Description

The `setutxent()` function resets the input to the beginning of the utmpx database opened by previous calls to `getutxid()`, `getutxent()`, `getutxline()`, or `pututxline()` calls from the current thread. This should be done before each `getutxid()` and `getutxline()` search for a new entry if it is desired that the entire database be examined.

Because the `setutxent()` function processes thread specific data the `setutxent()` function can be used safely from a multi-threaded application. If multiple threads in the same process open the database, then each thread opens the database with a different file descriptor. The thread's database file descriptor is closed when the calling thread terminates or the `endutxent()` function is called by the calling thread.

### Returned Value

The `setutxent()` function returns no value.

### Related Information

- “`__utmpxname()` — Change the utmpx Database Name” on page 1669
- “`getutxent()` — Read Next Entry in utmpx Database” on page 620
- “`getutxline()` — Search by Line utmpx Database” on page 624
- “`getutxid()` — Search by ID utmpx Database” on page 622
- “`pututxline()` — Write Entry to utmpx Database” on page 1061
- “`endutxent()` — Close the utmpx Database” on page 313



## setvbuf() — Control Buffering

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

### General Description

Controls the buffering strategy and buffer size for a specified stream. The *stream* pointer must refer to an open file, and `setvbuf()` must be the first operation on the file.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

The location pointed to by *buf* designates an area that you provide that the OS/390 C/C++ Run-Time Library can choose to use as a buffer for the stream. A *buf* value of NULL indicates that no such area is supplied and that the OS/390 C/C++ Run-Time Library is to assume responsibility for managing its own buffers for the stream. If you supply a buffer, it must exist until the stream is closed.

If *type* is `_IOFBF` or `_IOLBF`, *size* is the size of the supplied buffer. If *buf* is NULL, the C library will take *size* as the suggested size for its own buffer. If *type* is `_IONBF`, both *buf* and *size* are ignored. Unbuffered I/O is allowed for memory files and hierarchical file system (HFS) files. However, it is not permitted for Hiperspace memory files.

Value	Meaning
<code>_IONBF</code>	No buffer is used.
<code>_IOFBF</code>	Full buffering is used for input and output. Use <i>buf</i> as the buffer and <i>size</i> as the size of the buffer.
<code>_IOLBF</code>	Line buffering is used for text stream I/O and terminal I/O. The buffer is flushed when a new-line character is used (text stream), when the buffer is full, or when input is requested (terminal).

The value for *size* must be greater than 0.

**Attention:** If you use `setvbuf()` or `setbuf()` to define your own buffer for a stream, you must ensure that either the buffer is available after program termination, or the stream is closed or flushed, before you call `exit()`. This can be done by defining the array with file scope or by dynamically allocating the storage for the array using `malloc()`.

For example, if the buffer is declared within the scope of a function block, the *stream* must be closed before the function is terminated. This prevents the storage allocated to the buffer from being freed.

### Returned Value

Returns the value 0 if successful, or even if it chooses not to use your buffer. It returns nonzero if an invalid value was specified in the parameter list, or if the request cannot be performed.

### Example

```
/* This example sets up a buffer of buf for stream1 and specifies that
   input from stream2 is to be unbuffered.
*/
#include <stdio.h>
#define BUF_SIZE 1024

char buf[BUF_SIZE];

int main(void)
{
    FILE *stream1, *stream2;

    stream1 = fopen("myfile1.dat", "r");
    stream2 = fopen("myfile2.dat", "r");

    /* stream1 uses a user-assigned buffer of BUF_SIZE bytes */
    if (setvbuf(stream1, buf, _IOFBF, sizeof(buf)) != 0)
        printf("Incorrect type or size of buffer 1");

    /* stream2 is unbuffered */
    if (setvbuf(stream2, NULL, _IONBF, 0) != 0)
        printf("Incorrect type or size of buffer 2");
    :
}
```

### Related Information

- One of the sections about I/O Operations in the *OS/390 C/C++ Programming Guide*
- “stdio.h” on page 43
- “fclose() — Close File” on page 348
- “fflush() — Write Buffer to File” on page 385
- “fopen() — Open a File” on page 417
- “setbuf() — Control Buffering” on page 1208

## shmat() — Shared Memory Attach Operation

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE
#include <sys/shm.h>
```

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

### General Description

The `shmat()` function attaches the shared memory segment associated with the shared memory identifier, `shmid`, to the address space of the calling process. The segment is attached at the address specified by one of the following criteria:

- If `shmaddr` is a null pointer, the segment is attached at the first available address as selected by the system.
- If `shmaddr` is not a null pointer, and the flag, **SHM\_RND** was specified, the segment is attached at the address given by `(shmaddr - ((ptrdiff_t)shmaddr % SHMLBA))` where `%` is the 'C' language remainder operator.
- If `shmaddr` is not a null pointer, and the flag, **SHM\_RND** was not specified, the segment is attached at the address given by `shmaddr`.
- The segment is attached for reading if the flag, **SHM\_RDONLY**, is specified with `shmflg` and the calling process has read permission. If the flag is not set and the process has both read and write permission, the segment is attached for reading and writing.

The first attach of newly created **\_\_IPC\_MEGA** segment, as well as subsequent attaches, will have write access to the segment, regardless of the **SHM\_RDONLY** option.

- All attaches to an **\_\_IPC\_MEGA** shared memory segment have the same Write or Read access authority. If a segment is enabled for writes then all attaches have the ability to read and write to the segment. If the segment is disabled for writes, then all attaches have the ability to read from the segment and cannot write to the segment

The first attach of newly created **\_\_IPC\_MEGA** segment, as well as subsequent attaches, will have write access to the segment, regardless of the **SHM\_RDONLY** option. Write/Read access can be changed by the `shmctl()` function, Shared Memory Control Operations.

An **\_\_IPC\_MEGA** shared memory segment is attached as follows:

- If `shmaddr` is zero and **\_\_IPC\_MEGA** segment, then the segment will be attached at the first available address selected by the system on a segment boundary.
- If `shmaddr` is not zero and **SHM\_RND** is specified and **\_\_IPC\_MEGA** segment, the segment address will be truncated to the segment boundary (last 20 bits zero).

- If `shmaddr` is not zero and `SHM_RND` is not specified and `__IPC_MEGA` segment, the segment address must be a megabyte multiple (segment boundary).

### Returned Value

If successful, the `shmat()` function increments the value of `shm_nattach` in the data structure associated with the shared memory ID of the attached shared memory segment and returns the segment's starting address.

If unsuccessful, `shmat()` returns `-1`, and returns the error value in `errno`. The following are the possible values of `errno`:

- |        |   |
|--------|---|
| EACCES | Operation permission is denied to the calling process.  |
| EINVAL | The value of <i>shmid</i> is not a valid shared memory identifier; the <i>shmaddr</i> is not a null pointer and the value of $(shmaddr - ((ptrdiff\_t)shmaddr \% SHMLBA))$ is an illegal address for attaching shared memory segments; or the <i>shmaddr</i> is not a null pointer, <b>SHM_RND</b> was specified, and the value of <i>shmaddr</i> is an illegal address for attaching shared memory segments.<br><br>The shared memory address, <i>*shmaddr</i> , is not zero, is not on a megabyte boundary, and <code>SHM_RND</code> was not specified. |
| EMFILE | The number of shared memory segments attached to the calling process would exceed the system-imposed limit.   |
| ENOMEM | The available data space is not large enough to accommodate the shared memory segment.  |

### Related Information

- “sys/shm.h” on page 48
- “rexec() — Execute Commands One at a Time on a Remote Host” on page 1143
- “exit() — End Program” on page 330
- “\_exit() — End a Process and Bypass the Cleanup” on page 332
- “fork() — Create a New Process” on page 422
- “shmctl() — Shared Memory Control Operations” on page 1287
- “shmdt() — Shared Memory Detach Operation” on page 1289
- “shmget() — Get a Shared Memory Segment” on page 1290

shmctl() — Shared Memory Control Operations

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 5.2.2

Format

```
#define _XOPEN_SOURCE
#include <sys/shm.h>
```

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

General Description

The shmctl() function provides a variety of shared memory control operations on the shared memory segment identified by the argument, *shmid*.

The argument, *cmd*, specifies the shared memory control operation and may be any of the following values:

- IPC\_STAT This command obtains status information for the shared memory segment specified by the shared memory identifier, *shmid*. It places the current value of each member of the *shmid\_ds* data structure associated with *shmid* into the structure pointed to by *buf*. The contents of this structure is defined in <sys/shm.h>. This command requires read permission.
- IPC\_SET Set the value of the following members of the *shmid\_ds* data structure associated with *shmid* to the corresponding value in the structure pointed to by *buf*.

```
shm_perm.uid
shm_perm.gid
shm_perm.mode (only the low-order 9 bits)
```

This command can only be executed by a process that has an effective user ID equal to either that of a process with appropriate privileges or to the value of *shm\_perm.cuid* or *shm\_perm.uid* in the *shmid\_ds* data structure associated with *shmid*.

Using the IPC\_SET function to change the IPC\_MODE for an \_\_IPC\_MEGA shared memory segment will have an immediate effect on all attaches to the target segment. That is, the read and write access of all current attachers is immediately affected by the permissions specified in the new IPC\_MODE. To determine how the new mode affects access, you must consider the effect of all three parts of the mode field (the owner permissions, group permissions and other permissions). If all three read and all three write permissions in the new mode are set off, then the access for all attachers is changed to read. If any of the three read permission bits is set on but the corresponding write permission bit is off, then the access for all attachers is changed to read. Otherwise, the access of all attachers is changed to write.

**IPC\_RMID** Remove the shared memory identified specified by *shmid* from the system and destroy the shared memory segment and *shmid\_ds* data structure associated with *shmid*. This command can only be executed by a process that has an effective user ID equal to either that of a process with appropriate privileges or to the value of *shm\_perm.cuid* or *shm\_perm.uid* in the *shmid\_ds* data structure associated with *shmid*. The remove will be completed asynchronous to the return from the *shmctl()* function, when the last attachment is detached. When **IPC\_RMID** is processed, no further attaches will be allowed.

### Returned Value

If successful, *shmctl()* returns a value of 0 (zero).

If unsuccessful, *shmctl()* returns -1, and returns the error value in *errno*. The following are the possible values of *errno*:

<b>EACCES</b>	The argument, <i>cmd</i> , is equal to <b>IPC_STAT</b> but the calling process does not have read permission.
<b>EINVAL</b>	The value of <i>shmid</i> is not a valid shared memory identifier or the value of <i>cmd</i> is not a valid command.
<b>EPERM</b>	The argument, <i>cmd</i> , is equal to either <b>IPC_RMID</b> or <b>IPC_SET</b> and the effective user ID of the calling process is not equal to that of a process with appropriate privileges and it is not equal to the value of <i>shm_perm.cuid</i> or <i>shm_perm.uid</i> in the data structure associated with <i>shmid</i> .

### Related Information

- “sys/shm.h” on page 48
- “sys/ipc.h” on page 47
- “shmat() — Shared Memory Attach Operation” on page 1285
- “shmdt() — Shared Memory Detach Operation” on page 1289
- “shmget() — Get a Shared Memory Segment” on page 1290

## shmdt() — Shared Memory Detach Operation

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE
#include <sys/shm.h>
```

```
int shmdt(const void *shmaddr);
```

### General Description

The `shmdt()` function detaches from the calling process's address space the shared memory segment located at the address specified by the argument, *shmaddr*.

### Returned Value

If successful, `shmdt()` decrements the value of `shm_nattach` in the data structure associated with the shared memory ID of the attached shared memory segment and returns zero.

If unsuccessful, `shmdt()` returns `-1`, and returns the error value in `errno`. The following are the possible values of `errno`:

**EINVAL**      The value of *shmaddr* is not the data segment start address of a shared memory segment.

### Related Information

- “`sys/shm.h`” on page 48
- “`rexec()` — Execute Commands One at a Time on a Remote Host” on page 1143
- “`exit()` — End Program” on page 330
- “`_exit()` — End a Process and Bypass the Cleanup” on page 332
- “`fork()` — Create a New Process” on page 422
- “`shmat()` — Shared Memory Attach Operation” on page 1285
- “`shmctl()` — Shared Memory Control Operations” on page 1287
- “`shmget()` — Get a Shared Memory Segment” on page 1290

## shmget() — Get a Shared Memory Segment

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE
#include <sys/shm.h>
```

```
int shmget(key_t key, size_t size, int shmflg);
```

### General Description

The `shmget()` function returns the shared memory identifier associated with *key*.

A shared memory identifier, associated data structure and shared memory segment of at least *size* bytes, see `<sys/shm.h>`, are created for *key* if one of the following is true:

1. Argument *key* has a value of **IPC\_PRIVATE**
2. Argument *key* does not already have a shared memory identifier associated with it and the flag **IPC\_CREAT** was specified

Specify `__IPC_MEGA` to request segment level sharing. The resulting shared memory segment will be allocated in units of segments instead of units of pages. The shared memory size parameter still reflects the number of bytes required but must be in megabyte multiples. A shared memory size parameter of 0 or one which is not a megabyte multiple will result in the request failing.

The first `shmget` to define the shared memory segment determines whether the segment has the `__IPC_MEGA` attribute or not. Subsequent `shmgets`, those that use existing shared memory segments, will use the `__IPC_MEGA` attribute defined by that segment. The `__IPC_MEGA` option will have no effect for these `shmgets` and will be ignored.

Specification of the `__IPC_MEGA` option for large segments will result in significant real storage savings and reduced ESQA usage, especially as the number of shares increases.

Valid values for the argument *shmflg* include any combination of the following constants defined in `<sys/ipc.h>` and `<sys/modes.h>`:

<b>IPC_CREAT</b>	Create a shared memory segment if the <i>key</i> specified does not already have an associated ID. <b>IPC_CREAT</b> is ignored when <b>IPC_PRIVATE</b> is specified.
<b>IPC_EXCL</b>	Causes the <code>shmget()</code> function to fail if the <i>key</i> specified has an associated ID. <b>IPC_EXCL</b> is ignored when <b>IPC_CREAT</b> is not specified or <b>IPC_PRIVATE</b> is specified.



__IPC_MEGA	Requests a shared memory segment with the size in megabyte multiples. Use of this option requires that the size parameter, <code>size_t</code> , be in a megabyte multiple. The <code>__IPC_MEGA</code> option is required to create the shared memory segment but the <code>__IPC_MEGA</code> option is not required to acquire access to a previously defined/created shared memory segment that has the <code>__IPC_MEGA</code> attribute.
S_IRUSR	Permits read access when the effective user ID of the caller matches either <code>shm_perm.cuid</code> or <code>shm_perm.uid</code> .
S_IWUSR	Permits write access when the effective user ID of the caller matches either <code>shm_perm.cuid</code> or <code>shm_perm.uid</code> .
S_IRGRP	Permits read access when the effective group ID of the caller matches either <code>shm_perm.cgid</code> or <code>shm_perm.gid</code> .
S_IWGRP	Permits write access when the effective group ID of the caller matches either <code>shm_perm.cgid</code> or <code>shm_perm.gid</code> .
S_IROTH	Permits other read access.
S_IWOTH	Permits other write access.

When a shared memory segment associated with argument *key* already exists, setting **IPC\_EXCL** and **IPC\_CREAT** in argument *shmflg* will force `shmget()` to fail.

The following fields are initialized when a `shmid_ds` data structure is created:

- The fields `shm_perm.cuid` and `shm_perm.uid` are set equal to the effective user ID of the calling process
- The fields `shm_perm.cgid` and `shm_perm.gid` are set equal to the effective group ID of the calling process
- The low order 9 bits of `shm_perm.mode` are set to the value in the low order 9 bits of *shmflg*
- The field `shm_segsz` is set equal to the value of the argument *size*
- The field `shm_lpid`, `shm_nattach`, `shm_atime`, and `shm_dtime` are set equal to zero
- The value of `shm_ctime` is set equal to the current time

## Returned Value

If successful, `shmget()` returns a non-negative integer, namely a shared memory identifier.

If unsuccessful, `shmget()` returns `-1`, and returns the error value in `errno`. The following are the possible values of `errno`:

EACCES	A shared memory identifier exists for the argument <i>key</i> , but operation permission as specified by the low order 9 bits of <i>shmflg</i> could not be granted
EEXIST	A shared memory identifier exists for the argument <i>key</i> and both <b>IPC_CREAT</b> and <b>IPC_EXCL</b> are specified in <i>shmflg</i>
EINVAL	A shared memory identifier does not exist for the argument <i>key</i> specified and the value of argument, <i>size</i> , is less than the system-imposed minimum or greater than the system-imposed maximum.

**OR** a shared memory identifier exists for the argument, *key*, but the size of the segment associated with it is less than that specified by argument *size*.

**OR** `__IPC_MEGA` is specified and the segment size, *size\_t*, is not in megabyte multiples.

- |        |  |
|--------|--|
| ENOENT | A shared memory identifier does not exist for the argument, <i>key</i> , and <b>IPC_CREAT</b> is not specified.  |
| ENOMEM | A shared memory identifier and associated shared memory segment are to be created but the amount of available system storage was insufficient to fill the request.     |
| ENOSPC | A shared memory identifier is to be created but the system-imposed limit on the maximum number of allocated shared memory identifiers, system-wide, would be exceeded. |

When *shmflg* equals 0, the following applies:

- If a shared memory identifier has already been created with *key* earlier, and the calling process of this `shmget()` has read and/or write permissions to it, then `shmget()` returns the associated semaphore identifier.
- If a semaphore identifier has already been created with *key* earlier, and the calling process of this `shmget()` does not have read and/or write permissions to it, then `shmget()` returns -1 and sets `errno` to `EACCES`.
- If a semaphore identifier has not been created with *key* earlier, then `shmget()` returns -1 and sets `errno` to `ENOENT`.

### Related Information

- “`sys/shm.h`” on page 48
- “`sys/ipc.h`” on page 47
- “`sys/types.h`” on page 49
- “`shmat()` — Shared Memory Attach Operation” on page 1285
- “`shmctl()` — Shared Memory Control Operations” on page 1287
- “`shmdt()` — Shared Memory Detach Operation” on page 1289
- “`ftok()` — Generate an Interprocess Communication (IPC) key” on page 488

## shutdown() — Shut Down All or Part of a Duplex Connection

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int shutdown(int socket, int how);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/socket.h>

long shutdown(int *s, int how);
```

### General Description

The `shutdown()` call shuts down all or part of a duplex connection.

#### Parameter Description

*socket*      The socket descriptor.

*how*          The condition of the shutdown. The values 0, 1, or 2 set the condition. *how* sets the condition for shutting down the connection to socket *socket*.

*how* can have a value of:

- **SHUT\_RD**, which ends communication from socket *socket*.
- **SHUT\_WR**, which ends communication to socket *socket*.
- **SHUT\_RDWR**, which ends communication both to and from socket *socket*.

**Note:** You should issue a `shutdown()` call before you issue a `close()` call for a socket.

#### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

The value 0 indicates success; the value -1 indicates an error. The value of the error code indicates the specific error.

#### Error Code Description

**EBADF**      *socket* is not a valid socket descriptor.

**EINVAL**      The *how* parameter was not set to one of the valid values.

**ENOBUFS**    Insufficient system resources are available to complete the call.

### ENOTCONN

The socket is not connected.

### ENOTSOCK

The descriptor is for a file, not for a socket.

### Related Information

- “accept() — Accept a New Connection on a Socket” on page 75
- “close() — Close a File” on page 191
- “connect() — Connect a Socket” on page 214
- “socket() — Create a Socket” on page 1371

## sigaction() — Examine or Change a Signal Action

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	POSIX(ON) MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <signal.h>
```

```
int sigaction(int sig, const struct sigaction *new, struct sigaction *old);
```

### General Description

Examines and changes the action associated with a specific signal.

`int sig` is the number of a recognized signal. `sigaction()` examines and sets the action to be associated with this signal. Refer to Table 32 for the values of `sig`, as well as the signals supported by OS/390 UNIX services. The `sig` argument must be one of the macros defined in the `signal.h` header file.

`const struct sigaction *new` may be a NULL pointer. If so, `sigaction()` merely determines the action currently defined to handle `sig`. It does not change this action. If `new` is not NULL, it should point to a `sigaction` structure. The action specified in this structure becomes the new action associated with `sig`.

`struct sigaction *old` points to a memory location where `sigaction()` can store a `sigaction` structure. `sigaction()` uses this memory location to store a `sigaction` structure describing the action currently associated with `sig`. `old` can also be a NULL pointer, in which case `sigaction()` does not store this information.

This function is supported only in a POSIX program. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information.

### Special Behavior for C++

- The behavior when mixing signal-handling with C++ exception handling is undefined. Also, the use of signal-handling with constructors and destructors is undefined.
- C++ and C language linkage conventions are incompatible, and therefore `sigaction()` cannot receive C++ function pointers. If you attempt to pass a C++ function pointer to `sigaction()`, the compiler will flag it as an error. Therefore, to use the `sigaction()` function in the C++ language, you must ensure that signal handler routines established have C linkage, by declaring them as `extern "C"`.

### Signals

Table 32 (Page 1 of 3). Signals

Value	Default Action	Meaning
SIGABND	1	Abend.

Table 32 (Page 2 of 3). Signals

Value	Default Action	Meaning
SIGABRT	1	Abnormal termination (sent by abort()).
SIGALRM	1	A timeout signal (sent by alarm()).
SIGBUS	1	Bus error (available only when running on MVS 5.2 or higher)
SIGFPE	1	Arithmetic exceptions that are not masked, for example, overflow, division by zero, and incorrect operation.
SIGHUP	1	A controlling terminal is suspended, or the controlling process ended.
SIGILL	1	Detection of an incorrect function image.
SIGINT	1	Interactive attention.
SIGKILL	1	A termination signal that cannot be caught or ignored.
SIGPIPE	1	A write to a pipe that is not being read.
SIGPOLL	1	Pollable event occurred (available only when running on MVS 5.2 or higher)
SIGPROF	1	Profiling timer expired (available only when running on MVS 5.2 or higher)
SIGQUIT	1	A quit signal for a terminal.
SIGSEGV	1	Incorrect access to memory.
SIGSYS	1	Bad system call issued (available only when running on MVS 5.2 or higher)
SIGTERM	1	Termination request sent to the program.
SIGTRAP	1	Internal for use by dbx or ptrace.
SIGURG	2	High bandwidth data is available at a socket (available only when running on MVS 5.2 or higher)
SIGUSR1	1	Intended for use by user applications.
SIGUSR2	1	Intended for use by user applications.
SIGVTALRM	1	Virtual timer has expired (available only when running on MVS 5.2 or higher)
SIGXCPU	1	CPU time limit exceeded (available only when running on MVS 5.2 or higher). If a process runs out of CPU time and SIGXCPU is caught or ignored, OE generates a SIGKILL."
SIGXFSZ	1	File size limit exceeded (available only when running on MVS 5.2 or higher)
SIGCHLD	2	An ended or stopped child process. (SIGCLD is an alias name for this signal.)
SIGDCE	2	Signal is used by DCE (available only when running MVS 5.1 or higher).
SIGIO	2	Completion of input or output.

Table 32 (Page 3 of 3). Signals

Value	Default Action	Meaning
SIGIOERR	2	A serious I/O error was detected. (When running on MVS 4.3, this signal is not supported by the kernel. It will be mapped to SIGIO. An application that uses SIGIO and SIGIOERR may have undesirable results. This limitation is removed when running on MVS 5.1, as SIGIO and SIGIOERR are both supported.)
SIGWINCH	2	Window size has changed (available only when running on MVS 5.2 or higher)
SIGSTOP	3	A stop signal that cannot be caught or ignored.
SIGTSTP	3	A stop signal for a terminal.
SIGTTIN	3	A background process attempted to read from a controlling terminal.
SIGTTOU	3	A background process attempted to write to a controlling terminal.
SIGCONT	4	If stopped, continue.

The **Default Actions** in Table 32 on page 1295 are:

- 1 Normal termination of the process.
- 2 Ignore the signal.
- 3 Stop the process.
- 4 Continue the process if it is currently stopped. Otherwise, ignore the signal.

If the main program abends in a way that is not caught or handled by the operating system or application, OS/390 UNIX terminates the running application with a KILL -9. If OS/390 UNIX gets control in EOT or EOM and the terminating status has not been set, OS/390 UNIX sets it to appear as if a KILL -9 occurred.

If a signal catcher for a SIGABND, SIGFPE, SIGILL or SIGSEGV signal runs as a result of a program check or an ABEND, and the signal catcher executes a RETURN statement, the process will be terminated.

## sigaction Structure

The sigaction structure is defined as follows:

```
struct sigaction {
    void      (*sa_handler)(int);
    sigset_t  sa_mask;
    int       sa_flags;
    void      (*sa_sigaction)(int, siginfo_t *, void *);
};
```

The following are members of the structure:

`void (*)(int) sa_handler`

A pointer to the function assigned to handle the signal. The value of this member can also be SIG\_DFL (indicating the default action) or SIG\_IGN (indicating that the signal is to be ignored).

**Special Behavior for XPG4.2:**

This member and `sa_sigaction` are mutually exclusive of each other. When the `SA_SIGINFO` flag is set in `sa_flags` then `sa_sigaction` is used. Otherwise, `sa_handler` is used.

`sigset_t sa_mask`

A signal set identifies a set of signals that are to be added to the signal mask of the calling process before the signal handling function `sa_handler` or **`sa_sigaction`** (in XPG4.2) is invoked. For more on signal sets, see “`sigemptyset()` — Initialize a Signal Mask to Exclude All Signals” on page 1318. You cannot use this mechanism to block `SIGKILL` or `SIGSTOP`. If `sa_mask` includes these signals, they will simply be ignored; `sigaction()` will not return an error.

`sa_mask` must be set by using one or more of the signal set manipulation functions: `sigemptyset()`, `sigfillset()`, `sigaddset()`, or `sigdelset()`

`int sa_flags`

A collection of flag bits that affect the behavior of signals. The following flag bits can be set in `sa_flags`:

**`SA_NOCLDSTOP`**

Tells the system not to issue a `SIGCHLD` signal when child processes stop. This is relevant only when the `sig` argument of `sigaction()` is `SIGCHLD`.

**`SA_NOCLDWAIT`**

Tells the system not to create 'zombie' processes when a child process dies. This is relevant only when the `sig` argument of `sigaction()` is `SIGCHLD`. If the calling process subsequently waits for its children, and the process has no unwaited for children that were transformed into zombie processes, it will block until all of its children terminate. The `wait()`, `waitid()`, or `waitpid()` will fail and set `errno` to **`ECHILD`**.

**`SA_NODEFER`**

Tells the system to bypass automatically blocking this signal when invoking a signal handler function.

**`_SA_OLD_STYLE`**

Tells the C runtime library to use ANSI signal delivery rules, instead of POSIX rules. It is supported for compatibility with applications that use `signal()` to set signal action. (See “`signal()` — Handle Interrupts” on page 1330.) For a description of ANSI and POSIX.1 signal delivery rules, find “Handling Error Conditions and Signals” in the *OS/390 C/C++ Programming Guide*.

**`SA_ONSTACK`**

Tells the system to use the alternate signal stack (see “`sigaltstack()` — Set and/or Get Signal Alternate Stack Context” on page 1314 or “`sigstack()` — Set and/or Get Signal Stack Context” on page 1349) when invoking a signal handler function. If an alternate signal stack has not been declared, the signal handler function will be invoked with the current stack.



## SA\_RESETHAND

Tells the system to reset the signal's action to SIG\_DFL and clear the SA\_SIGINFO flag before invoking a signal handler function (Note: SIGILL and SIGTRAP cannot be automatically reset when delivered. However, no error will be generated should this situation exist). Otherwise, the disposition of the signal will not be modified on entry to the signal handler.

In addition, if this flag is set, sigaction() behaves as if the SA\_NODEFER flag were also set.

## SA\_RESTART

Tells the system to restart certain library functions if they should be interrupted by a signal. The functions that this restartability applies to are all of those that are defined as interruptible by signals and set errno to **EINTR** (except pause(), sigpause(), and sigsuspend()).

This is the list of restartable functions:

- |                |                 |                   |
|----------------|-----------------|-------------------|
| 1) accept()    | 24) fstatvfs()  | 46) recvfrom()    |
| 2) catclose()  | 25) fsync()     | 47) recvmsg()     |
| 3) catgets()   | 26) ftruncate() | 48) select()      |
| 4) chmod()     | 27) getgrgid()  | 49) semop()       |
| 5) chown()     | 28) getgrnam()  | 50) send()        |
| 6) close()     | 29) getmsg()    | 51) sendmsg()     |
| 7) closedir()  | 30) getpass()   | 52) sendto()      |
| 8) connect()   | 31) getpwnam()  | 53) statvfs()     |
| 9) creat()     | 32) getpwuid()  | 54) tcdrain()     |
| 10) dup2()     | 33) ioctl()     | 55) tcflow()      |
| 11) endgrent() | 34) lchown()    | 56) tcflush()     |
| 12) fchmod()   | 35) lockf()     | 57) tcgetattr()   |
| 13) fchown()   | 36) mkfifo()    | 58) tcgetpgrp()   |
| 14) fclose()   | 37) msgrcv()    | 59) tcsendbreak() |
| 15) fcntl()    | 38) msgxrcv()   | 60) tcsetattr()   |
| 16) fflush()   | 39) msgsnd()    | 61) tcsetpgrp()   |
| 17) fgetc()    | 40) open()      | 62) tmpfile()     |
| 18) fgetwc()   | 41) poll()      | 63) umount()      |
| 19) fopen()    | 42) putmsg()    | 64) wait()        |
| 20) fputc()    | 43) read()      | 65) waitid()      |
| 21) fputwc()   | 44) readv()     | 66) waitpid()     |
| 22) freopen()  | 45) recv()      | 67) write()       |
| 23) fseek()    |                 |                   |

## SA\_SIGINFO

Tells the system to use the signal action specified by sa\_sigaction instead of sa\_handler.

When this flag is off and the action is to catch the signal, the signal handler function specified by sa\_handler is invoked as:

```
void function(int signo);
```

Where *signo* is the only argument to the signal handler and it specifies the type of signal that has caused the signal handler function to be invoked.

When this flag is on and the action is to catch the signal, the signal handler function specified by `sa_sigaction` is invoked as:

```
void function(int signo, siginfo_t *info, void *context);
```

Where two additional arguments are passed to the signal handler function. If the second argument is not a NULL pointer, it will point to an object of type `siginfo_t` which provides additional information about the source of the signal. A `siginfo_t` object is a structure contains the following members:

si_signo	Contains the system-generated signal number
si_errno	Contains the implementation-specific error information (it is not used on this implementation)
si_code	Contains a code identifying the cause of the signal (refer to the <code>&lt;signal.h&gt;</code> include file for a list of these codes and their meanings, see“signal() — Handle Interrupts” on page 1330).  If si_signo contains SIGPOLL then si_code can be set to SI_ASYNCIO. Otherwise, if the value of si_code is less than or equal to zero, then the signal was generated by another process and the si_pid and si_uid members respectively indicate the process ID and the real user ID of the sender of this signal.  If the value of si_code is less than or equal to zero, then the signal was generated by another process and the si_pid and si_uid members respectively indicate the process ID and the real user ID of the sender of this signal.
si_pid	If the value of si_code is less than or equal to zero, then this member will indicate the process ID of the sender of this signal. Otherwise, this member is meaningless.
si_uid	If the value of si_code is less than or equal to zero, then this member will indicate the real user ID of the sender of this signal. Otherwise, this member is meaningless.
si_value	If si_code is SI_ASYNCIO, the si_value contains the application specified value. Otherwise, the contents of si_value are undefined

The third argument will point to an object of type `ucontext_t` (refer to the `<ucontext.h>` include file for a description of the contents of this object).

**Note:** The remaining flag bits are reserved for system use. There is no guarantee that the integer value of 'int sa\_flags' will be the same upon return from `sigaction()`. However, all flag bits defined above will remain unchanged.

```
void (*)(int, siginfo_t *, void *) sa_sigaction
```

A pointer to the function assigned to handle the signal, or `SIG_DFL`, or `SIG_IGN`. This function will be invoked passing three parameters. The first is of type 'int' that contains the signal type for which this function is being invoked. The second is of type 'pointer to siginfo\_t' where the `siginfo_t` contain additional information about the source of the signal. The third is of type 'pointer to void' but will actually point to a `ucontext_t` containing the context information at the time of the signal interrupt.

**Notes:**

1. The user must cast `SIG_IGN` or `SIG_DFL` to match the `sa_sigaction` definition. (indicating that the signal is to be ignored).
2. **Special Behavior for XPG4.2:** This member and `sa_handler` are mutually exclusive of each other. When the `SA_SIGINFO` flag is set in `sa_flags` then `sa_sigaction` is used. Otherwise, `sa_handler` is used.

When a signal handler installed by `sigaction()`, with the `_SA_OLD_STYLE` flag set off, catches a signal, the system calculates a new signal mask by taking the union of the current signal mask, the signals specified by `sa_mask`, and the signal that was just caught (if the `SA_NODEFER` flag is not set). This new mask stays in effect until the signal handler returns, or `sigprocmask()`, `sigsuspend()`, `siglongjmp()`, `sighold()`, `sigpause()`, or `sigrelse()` is called. When the signal handler ends, the original signal mask is restored.

After an action has been specified for a particular signal, using `sigaction()` or `signal()`, it remains installed until it is explicitly changed with another call to `sigaction()`, `signal()`, one of the exec functions, `bsd_signal()`, `sigignore()`, `sigset()`, or until the `SA_RESETHAND` flag causes it to be reset to `SIG_DFL`.

After an action has been specified for a particular signal, using `sigaction()` with the `_SA_OLD_STYLE` flag not set, it remains installed until it is explicitly changed with another call to `sigaction()`, `signal()`, or one of the exec functions.

After an action has been specified for a particular signal, using `sigaction()` with the `_SA_OLD_STYLE` flag set or using `signal()`, it remains installed until it is explicitly changed with another call to `sigaction()`, `signal()`, or one of the exec functions, or a signal catcher is driven, where it will be reset to `SIG_DFL`.

Successful setting of signal action to `SIG_IGN` for a signal that is pending causes the pending signal to be discarded, whether or not it is blocked. This provides the ability to discard signals that are found to be blocked and pending by `sigpending()`.

### Special Behavior for XPG4.2

- If a process sets the action of the SIGCHLD signal to SIG\_IGN, child processes of the calling process will not be transformed into 'zombie' processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited for children that were transformed into 'zombie' processes, it will block until all of its children terminate. The wait(), waitid(), or waitpid() function will fail and set errno to **ECHILD**.
- If the **SA\_SIGINFO** flag is set, the signal-catching function specified by sa\_sigaction is invoked as:

```
void function(int signo, siginfo_t *info, void *context);
```

Where *function* is the specified signal-catching function, *signo* is the signal number of the signal being delivered, *info* points to an object of type `siginfo_t` associated with the signal being delivered, and *context* points to an object of type `ucontext_t`.

### Considerations for Asynchronous Signal-Catching Functions

Some of the functions have been restricted to be serially reusable with respect to asynchronous signals. That is, the library will not allow an asynchronous signal to interrupt the execution of one of these functions until it has completed.

This restriction needs to be taken into consideration when a signal-catching function is invoked asynchronously because it causes the behavior of some of the library functions to become unpredictable.

Thus, when you are producing a strictly compliant POSIX C or X/Open application, only the following functions should be assumed to be reentrant with respect to asynchronous signals. Use only these functions in your signal-catching functions:

access()	alarm()	cfgetispeed()
cfgetospeed()	cfsetispeed()	cfsetospeed()
chdir()	chmod()	chown()
close()	creat()	dup()
dup2()	execle()	execve()
_exit()	fcntl()	fork()
fstat()	getegid()	geteuid()
getgid()	getgroups()	getpgrp()
getpid()	getppid()	getuid()
kill()	link()	lseek()
mkdir()	mkfifo()	open()
pathconf()	pause()	pipe()
pthread_cond_broadcast()	pthread_cond_signal()	pthread_mutex_trylock()
read()	rename()	rmdir()
setgid()	setpgid()	setsid()
setuid()	sigaction()	sigaddset()
sigdelset()	sigemptyset()	sigfillset()
sigismember()	sigpending()	sigprocmask()
sigsuspend()	sleep()	stat()
sysconf()	tcdrain()	tcflow()
tcflush()	tcgetattr()	tcgetpgrp()
tcsendbreak()	tcsetattr()	tcsetpgrp()
time()	times()	umask()
uname()	unlink()	utime()
wait()	waitpid()	write()

### Special Behavior for XPG4.2

Adds the following functions to the list of functions above that may be used in signal-catching functions in strictly compliant X/Open applications:

- fpathconf()
- raise()
- signal()

The macro versions of `getc()` and `putc()` are not reentrant, even though the library versions of these functions are.

For nonportable POSIX applications, most of the library functions can be used in a signal-catching function. However, do not use the following functions:

- `getenv()`
- `getgrent()`
- `getgrgid()`
- `getgrnam()`
- `getpwent()`
- `getpwnam()`
- `getpwuid()`
- `ttyname()`

## Returned Value

If successful, `sigaction()` returns zero. If unsuccessful, it returns the value `-1`. No new signal handler is installed. `sigaction()` sets `errno` to `EINVAL`, which indicates that the value of *sig* is not a valid signal for one of the following reasons:

- The *sig* is not recognized
- The process tried to ignore a signal that cannot be ignored
- The process tried to catch a signal that cannot be caught

The default action for `SIGCHLD` and `SIGIO` is for the signal to be ignored. A `sigaction()` to set the action to `SIG_IGN` for `SIGIO` will result in an error, with `errno` equal to `EINVAL`.

## Example

### CBC3BS13

```
/* CBC3BS13
   The first part of this example determines whether the SIGCHLD signal is
   currently being ignored. With a NULL pointer for the new argument, the
   current signal handler action is not changed.
*/
struct sigaction info;
if (sigaction(SIGCHLD, NULL, &info) != -1)
    if (info.sa_handler == SIG_IGN)
        printf("SIGCHLD being ignored.\n");
```

### CBC3BS14

```
/* CBC3BS14
   This fragment initializes a sigaction structure to specify mysig as a
   signal handler and then sets the signal handler for SIGCHLD.
   Information on the previous signal handler for SIGCHLD is stored in info.
*/
extern void mysig();
struct sigaction info, newhandler;
newhandler.sa_handler = &mysig;
sigemptyset(&(newhandler.sa_mask));
```

```
newhandler.sa_flags = 0;
if (sigaction(SIGCHLD,&newhandler,&info) != -1)
    printf("New handler set.\n");
```

### Related Information

- “signal.h” on page 41
- “alarm() — Set an Alarm” on page 102
- “bsd\_signal() — BSD Version of signal()” on page 135
- “exec Functions” on page 322
- “getcontext() — Get User Context” on page 505
- “kill() — Send a Signal to a Process” on page 728
- “makecontext() — Modify User Context” on page 783
- “setcontext() — Restore User Context” on page 1210
- “raise() — Raise Signal” on page 1074
- “\_\_sigactionset() — Examine and/or Change Signal Actions” on page 1305
- “sigaddset() — Add a Signal to the Signal Mask” on page 1312
- “sigaltstack() — Set and/or Get Signal Alternate Stack Context” on page 1314
- “sigdelset() — Delete a Signal from the Signal Mask” on page 1316
- “sigemptyset() — Initialize a Signal Mask to Exclude All Signals” on page 1318
- “sigignore() — Set Disposition to Ignore a Signal” on page 1323
- “siginterrupt() — Allow Signals to Interrupt Functions” on page 1324
- “sigfillset() — Initialize a Signal Mask to Include All Signals” on page 1320
- “signal() — Handle Interrupts” on page 1330
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “sigset() — Change a Signal Action and/or a Thread” on page 1343
- “sigstack() — Set and/or Get Signal Stack Context” on page 1349
- “siglongjmp() — Restore the Stack Environment and Signal Mask” on page 1327
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351
- “swapcontext() — Save and Restore User Context” on page 1472
- “wait() — Wait for a Child Process to End” on page 1687
- “wait3() — Wait for Child Process to Change State” on page 1696

## \_\_sigactionset() — Examine and/or Change Signal Actions

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	POSIX(ON) OS/390 R6

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>
```

```
int __sigactionset(size_t newct, const __sigactionset_t new[],
                  size_t *oldct, __sigactionset_t old[],
                  int options);
```

### General Description

Examines and changes the actions associated with one or more signals. This function is equivalent to using `sigaction()` one or more times.

The parameters are:

`size_t newct`

*newct* is the number of `__sigactionset_t` structures to be processed in the *new* array. The value of *newct* must be from 0 to 64. If this parameter is 0, the *new* parameter is ignored, and may be NULL. If the *newct* parameter is not 0, *new* must be an array containing at least *newct* `__sigactionset_t` structures.

`const __sigactionset_t new[]`

*new* is an optional array of `__sigactionset_t` structures. When *newct* is 0, *new* may be NULL, and no signal actions will be changed.

When *newct* is not 0, the data in the *new* array of `__sigactionset_t` structures will cause the actions associated with one or more signals to be changed. The system will change the signal actions as if `sigaction()` were called multiple times. The first *newct* `__sigactionset_t` structures in the *new* array are processed in order, and may cause the actions for one or more signals to be set. For each array entry, the effect is the same as calling `sigaction()` once for each signal whose bit is on in the `__sa_signals` signal set. The fields `__sa_handler`, `__sa_mask`, `__sa_flags`, and `__sa_sigaction` correspond to the `sa_handler`, `sa_mask`, `sa_flags`, and `sa_sigaction` fields in the `sigaction` structure for `sigaction()`.

If a signal appears in more than one `__sa_signals` signal set in the *new* array, the last action specified for that signal will be in effect when `__sigactionset()` returns. If all bits in all `__sa_signals` signal sets in the *new* parameter are off, no signal actions will be changed.

`size_t *oldct`

*oldct* is both an input and output parameter. It points to a word containing the number of output entries allowed, used, or needed in the *old* array.

When `__sigactionset()` is called, *\*oldct* is the maximum number of `__sigactionset_t` structures in the *old* array that the system can fill in.

The value of *\*oldct* must be from 0 to 64. If this parameter is 0, the *old* parameter is ignored, and may be NULL. If the *\*oldct* parameter is not 0, *old* must be an array of `__sigactionset_t` structures that the system can fill in. The number of array entries in *old* must be at least *\*oldct*. If not 0, *\*oldct* must be large enough to allow the system to pass back all the unique actions currently associated with all signals. If *\*oldct* is not large enough, `__sigactionset()` will fail and the `errno` will be set to `ENOMEM`.

If `__sigactionset()` returns with no error and *\*oldct* was not 0, *\*oldct* is set to the number of `__sigactionset_t` array entries in *old* that are filled in. If *\*oldct* was too small, causing an `ENOMEM` error, *\*oldct* is set to the number of `__sigactionset_t` structures the system would need in order to fill in all the distinct current signal actions. If *\*oldct* was 0 when `__sigactionset()` was called, it is not updated.

`__sigactionset_t old[]`

*old* is an optional array of `__sigactionset_t` structures.

When *\*oldct* is not 0, the structures in the *old* array will be filled in with the signal actions currently in effect before any changes are made. The `__sigactionset_t` structure entries in *old* are filled in with all the distinct signal actions currently in effect, starting with the first array entry. Each `__sigactionset_t` structure in the array will contain information about one or more signals. Bits in the `__sa_signals` signal set in each array entry will indicate which signals that entry applies to. The system will try to use as few array entries as possible when passing back the different signal actions. The signal actions for `SIGKILL` and `SIGSTOP` will not be returned.

The output information in each array entry is similar to that returned from `sigaction()`. In the `__sigactionset_t` structure, the fields `__sa_handler`, `__sa_mask`, `__sa_flags`, and `__sa_sigaction` correspond to the `sa_handler`, `sa_mask`, `sa_flags`, and `sa_sigaction` fields in the `sigaction` structure filled in by `sigaction()`. The signal action as described by these fields applies to all signals whose bits are on in the `__sa_signals` signal set in the array entry.

If *old* is not large enough to contain information about all distinct signal actions currently in effect, `__sigactionset()` fails, and `ENOMEM` is returned. There is no way to obtain the current signal actions for a specified subset of signals.

When *old* is NULL, the system does not return any information about the current signal actions.

`int options`

*options* is a collection of flag bits that affects the operation `__sigactionset()`. The following flag bit can be set in *options*:

`__SSET_IGINVALID`

Tells the system to ignore invalid bits in the `__sa_signals` field in all `__sigactionset_t` array entries in the *new* parameter. Also, the system will ignore attempts to set `SIGKILL` or `SIGSTOP` to an action other than `SIG_DFL`, or `SIGIO` to `SIG_IGN`.

If this option bit is off, the system will fail the `__sigactionset()` request if any invalid bits are found in any `__sa_signals`



signal set in any *new* array entry. Also, \_\_sigactionset() will fail if an attempt it made to set SIGKILL or SIGSTOP to something other than SIG\_DFL, or to set SIGIO to SIG\_IGN.

This function is supported only in a POSIX(ON) program.

### Special Behavior for C++

The behavior when mixing signal-handling with C++ exception handling is undefined. Also, the use of signal handling with constructors and destructors is undefined.

C++ and C language linkage conventions are incompatible, and therefore \_\_sigactionset() cannot receive C++ function pointers. If you attempt to pass a C++ function pointer to \_\_sigactionset(), the compiler will flag it as an error. Therefore to use the \_\_sigactionset() function in the C++ language, you must ensure that signal handler routines have C linkage, by declaring them as extern "C".

### \_\_sigactionset\_t type

The \_\_sigactionset\_t type is defined as follows:

```
typedef struct __sigactionset_s
{
    sigset_t    __sa_signals;
    int         __sa_flags;
    void        (*__sa_handler)(int);
    sigset_t    __sa_mask;

    void        (*__sa_sigaction)(int, siginfo_t *, void *);
} __sigactionset_t;
```

The following are members of the structure:

sigset\_t \_\_sa\_signals

This is a signal set. It contains the signals whose actions are described by the other members in this structure. For more information on signal sets, see “sigemptyset() — Initialize a Signal Mask to Exclude All Signals” on page 1318.

In the *new* array of \_\_sigactionset\_t structures, the caller sets bits in this signal set. The signal action for each signal in the signal set will be set as described by the other members of the structure.

\_\_sa\_signals must be set using one or more of the signal set manipulation functions: sigaddset(), sigdelset(), sigemptyset(), or sigfillset().

In the *old* array of \_\_sigactionset\_t structures, the system sets the bits in the \_\_sa\_signals field. The current signal action for each member of the signal set is described by the other members of the structure. All signals in the set have the same signal action.

int \_\_sa\_flags

A collection of flag bits that affect the behavior of the specified signal.

The flag bits in the \_\_sa\_flags field are the same as those in the sa\_flags member of the sigaction structure. See “sigaction() —

Examine or Change a Signal Action” on page 1295 for a detailed description of these flag bits.

`void (* __sa_handler)(int)`

A pointer to the function assigned to handle the signals in the `__sa_signals` signal set. This function will be invoked passing one parameter of type `int` that contains the signal type for which this function is being invoked. The value of this member can also be `SIG_DFL` (indicating the default action) or `SIG_IGN` (indicating that the signal is to be ignored).

**Note:** This member and `__sa_sigaction` are mutually exclusive. When the `SA_SIGINFO` flag is set in `__sa_flags`, `__sa_sigaction` is used. Otherwise, `__sa_handler` is used.

`sigset_t __sa_mask`

This signal set identifies a set of signals that are to be added to the signal mask of the calling thread before the signal handling function `__sa_handler` or `__sa_sigaction` is invoked. For more information on signal sets, see “`sigemptyset()` — Initialize a Signal Mask to Exclude All Signals” on page 1318. You cannot use this mechanism to block `SIGKILL` or `SIGSTOP`. If `__sa_mask` includes these signals, they will simply be ignored; `__sigactionset()` will not return an error.

`__sa_mask` must be set by using one or more of the signal set manipulation functions: `sigaddset()`, `sigdelset()`, `sigemptyset()`, or `sigfillset()`.

`void (*__sa_sigaction)(int, siginfo_t *, void *)`

A pointer to the function assigned to handle the signal, or `SIG_DFL`, or `SIG_IGN`. This function will be invoked passing three parameters. The first is of type `int` that contains the signal type for which this function is being invoked. The second is of type `siginfo_t*` where the `siginfo_t` contain additional information about the source of the signal. The third is of type `void*` but will actually point to a `ucontext_t` containing the context information at the of time the signal interrupt.

**Notes:**

1. The user must cast `SIG_DFL` or `SIG_IGN` to match the `__sa_sigaction` definition.
2. This member and `sa_handler` are mutually exclusive. When the `SA_SIGINFO` flag is set in `__sa_flags`, `__sa_sigaction` is used. Otherwise, `__sa_handler` is used.

When a signal handler installed by `__sigactionset()`, with the `_SA_OLD_STYLE` flag set off, catches a signal, the system calculates a new signal mask by taking the union of the current signal mask at the time of the signal interrupt, the signals specified by `__sa_mask`, and the signal that was just caught (if the `SA_NODEFER` flag is not set). This new mask stays in effect until the signal handler returns, or `sigprocmask()`, `sigsuspend()`, `siglongjmp()`, `sighold()`, `sigpause()`, or `sigrelse()` is called. When the signal handler ends, the original signal mask is restored.

After an action has been specified for a particular signal, using `__sigactionset()` with the `_SA_OLD_STYLE` flag not set, it remains installed until it is explicitly changed with another call to `__sigactionset()`, `sigaction()`, `signal()`, `bsd_signal()`, `sigset()`, `sigignore()`, one of the `exec` functions, or until the `SA_RESETHAND` flag causes it to be reset to `SIG_DFL`.

After an action has been specified for a particular signal, using `__sigactionset()` with the `_SA_OLD_STYLE` flag set or using `signal()`, it remains installed until it is explicitly changed with another call to `__sigactionset()`, `sigaction()`, `bsd_signal()`, `sigset()`, `signal()`, `sigignore()`, one of the `exec` functions, or a signal catcher is driven, where it will be reset to `SIG_DFL`.

Successful setting of a signal action to `SIG_IGN` for a signal that is pending causes the pending signal to be discarded, whether or not it is blocked. This provides the ability to discard signals that are found to be blocked and pending by `sigpending()`. A signal is discarded across a call to `__sigactionset()` if any `__sigactionset_t` structure in the *new* array causes the action for that signal to be set to `SIG_IGN`. This happens even if a later `__sigactionset_t` structure in the *new* array sets the signal action to something other than `SIG_IGN` before `__sigactionset()` returns.

If a process sets the action of the `SIGCHLD` signal to `SIG_IGN`, child processes of the calling process will not be transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited from children that were transformed into zombie processes, it will block until all of its children terminate. The `wait()`, `waitid()`, or `waitpid()` function will fail and set `errno` to `ECHILD`.

If the `SA_SIGINFO` flag is set, the signal catching function specified by `__sa_sigaction` is invoked as:

```
void function(int signo, siginfo_t *info, void * context);
```

where *function* is the specified signal-catching function, *signo* is the signal number of the signal being delivered, *info* points to an object of type `siginfo_t` associated with the signal being delivered, and *context* points to an object of type `ucontext_t`.

For a signal catcher that has been loaded by `fetch()` or `fetchep()`, the address returned by `__sigactionset()` in the `__sa_handler` or `__sa_sigaction` fields may be different than the value originally passed in to `sigaction()` or `__sigactionset()` (when the signal action was first set). This signal catcher address can be passed in again to `sigaction()` or `__sigactionset()` to re-establish the same signal catcher. The effect will be similar to passing in the original catcher address obtained from `fetch()` or `fetchep()`. However, this address should not be used for any other purpose, such as directly calling the signal catcher. Always use the original address obtained from `fetch()` or `fetchep()` when calling the catcher directly.

### Considerations for Asynchronous Signal-Catching Functions

Some of the functions have been restricted to be serially reusable with respect to asynchronous signals. For more information on these functions, see “`sigaction()` — Examine or Change a Signal Action” on page 1295.

## Returned Value

If successful, `__sigactionset()` returns a zero.

If unsuccessful, it returns `-1`, and no signal actions are changed. When `__sigactionset()` is unsuccessful, one of the following `errno` values is returned:

- EINVAL** This error can occur if:
- An unsupported signal bit was on in the `__sa_signals` signal set in the *new* parameter. This error will not be reported if the `__SSET_IGINVALID` flag is set in *options*. To obtain more information in this case, use `__errno2()`.
  - An attempt was made to set the signal action for `SIGSTOP` or `SIGKILL` to something other than `SIG_DFL`. This error will not be reported if the `__SSET_IGINVALID` flag is set in *options*. To obtain more information in this case, use `__errno2()`.
  - The *newct* or *oldct* parameters are not in the range from 0 to 64.
  - *newct* was not 0 and *new* was `NULL`.
  - *oldct* was not 0 and *old* was `NULL`.
- EMVSERR** An MVS environmental or internal error has occurred. Use `__errno2()` to obtain more information about this error.
- ENOMEM** The input value in *oldct* was not 0, and was too small to let the system pass back all distinct current signal actions. When `__sigactionset()` returns, *\*oldct* will be set to the number of array entries needed by the system.

## Example

```
/*
 * Note: This is just a code fragment
 */

void catch_sigchld(int, siginfo_t *, void *);

...

__sigactionset_t new[2], old[64];
int options;
int rc;
size_t oldct = 64;

/*
 * Set SIGUSR1 and SIGUSR2 to SIG_IGN
 * Set SIGCHLD to new-style catcher catch_sigchld()
 * Save original signal setup in variable old
 */

bzero(new, sizeof new);

(void)sigemptyset(&(new[0].__sa_signals) );
(void)sigaddset (&(new[0].__sa_signals), SIGUSR1);
(void)sigaddset (&(new[0].__sa_signals), SIGUSR2);

(void)sigemptyset(&(new[1].__sa_signals) );
(void)sigaddset (&(new[1].__sa_signals), SIGCHLD);

new[0].__sa_handler = SIG_IGN;
```

```

new[1].__sa_sigaction = &catch_sigchld;
new[1].__sa_flags      = SA_SIGINFO;

rc = __sigactionset((size_t)2, new, &oldct, old, __SSET_IGINVALID);

```

## Related Information

- “signal.h” on page 41
- “alarm() — Set an Alarm” on page 102
- “bsd\_signal() — BSD Version of signal()” on page 135
- “exec Functions” on page 322
- “getcontext() — Get User Context” on page 505
- “kill() — Send a Signal to a Process” on page 728
- “makecontext() — Modify User Context” on page 783
- “setcontext() — Restore User Context” on page 1210
- “raise() — Raise Signal” on page 1074
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sigaddset() — Add a Signal to the Signal Mask” on page 1312
- “sigaltstack() — Set and/or Get Signal Alternate Stack Context” on page 1314
- “sigdelset() — Delete a Signal from the Signal Mask” on page 1316
- “sigemptyset() — Initialize a Signal Mask to Exclude All Signals” on page 1318
- “sigignore() — Set Disposition to Ignore a Signal” on page 1323
- “siginterrupt() — Allow Signals to Interrupt Functions” on page 1324
- “sigfillset() — Initialize a Signal Mask to Include All Signals” on page 1320
- “signal() — Handle Interrupts” on page 1330
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “sigset() — Change a Signal Action and/or a Thread” on page 1343
- “sigstack() — Set and/or Get Signal Stack Context” on page 1349
- “siglongjmp() — Restore the Stack Environment and Signal Mask” on page 1327
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351
- “swapcontext() — Save and Restore User Context” on page 1472
- “wait() — Wait for a Child Process to End” on page 1687
- “wait3() — Wait for Child Process to Change State” on page 1696

## sigaddset() — Add a Signal to the Signal Mask

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	

### Format

```
#define _POSIX_SOURCE
#include <signal.h>
```

```
int sigaddset(sigset_t *set, int signal);
```

### General Description

Adds a signal to the set of signals already recorded in *set*.

`sigaddset()` is part of a family of functions that manipulate signal sets. *Signal sets* are data objects that let a process keep track of groups of signals. For example, a process can create one signal set to record which signals it is blocking, and another signal set to record which signals are pending. In general, signal sets are used to manipulate groups of signals used by other functions (such as `sigprocmask()`) or to examine signal sets returned by other functions (such as `sigpending()`).

### Returned Value

If the signal is successfully added to the signal set, `sigaddset()` returns zero. If *signal* is not supported, `sigaddset()` returns the value `-1` and sets `errno` `EINVAL`.

### Example

#### CBC3BS15

```
/* CBC3BS15
   This example adds a set of signals.
*/
#define _POSIX_SOURCE
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void catcher(int signum) {
    puts("catcher() has gained control");
}

main() {
    struct sigaction sact;
    sigset_t sigset;

    sigemptyset(&sact.sa_mask);
    sact.sa_flags = 0;
    sact.sa_handler = catcher;
    sigaction(SIGUSR1, &sact, NULL);

    puts("before first kill()");
    kill(getpid(), SIGUSR1);
    puts("before second kill()");
```

```

sigemptyset(&sigset);
sigaddset(&sigset, SIGUSR1);
sigprocmask(SIG_SETMASK, &sigset, NULL);

kill(getpid(), SIGUSR1);
puts("after second kill()");
}

```

### Output

```

before first kill()
catcher() has gained control
before second kill()
after second kill()

```

### Related Information

- “signal.h” on page 41
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sigdelset() — Delete a Signal from the Signal Mask” on page 1316
- “sigemptyset() — Initialize a Signal Mask to Exclude All Signals” on page 1318
- “sigfillset() — Initialize a Signal Mask to Include All Signals” on page 1320
- “sighold() — Add a Signal to a Thread” on page 1322
- “sigismember() — Test If a Signal Is in a Signal Mask” on page 1325
- “sigpending() — Examine Pending Signals” on page 1337
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “sigpause() — Unblock a Signal and Wait for a Signal” on page 1336
- “sigpending() — Examine Pending Signals” on page 1337
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351
- “signal() — Handle Interrupts” on page 1330
- “sigset() — Change a Signal Action and/or a Thread” on page 1343
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351

## sigaltstack() — Set and/or Get Signal Alternate Stack Context

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON)

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>
```

```
int sigaltstack(const stack_t *ss, stack_t *oss);
```

### General Description

The `sigaltstack()` function allows a thread to define and examine the state of an alternate stack for signal handlers. Signals that have been explicitly declared to execute on the alternate stack will be delivered on the alternate stack.

**Note:** To explicitly declare that a signal catcher is to run on the alternate signal stack, the `SA_ONSTACK` flag must be set in the `sa_flags` when the signal action is set via `sigaction()`.

If `ss` is not a null pointer, it points to a `stack_t` structure that specifies the alternate signal stack that will take effect upon return from `sigaltstack()`. The `ss_flags` member specifies the new stack state. If it is set to `SS_DISABLE`, the stack is disabled and `ss_sp` and `ss_size` are ignored. Otherwise the stack will be enabled, and the `ss_sp` and `ss_size` members specify the new address and size of the stack.

The range of addresses starting at `ss_sp`, up to but not including `ss_sp + ss_size`, is available to the implementation for use as the stack. This interface makes no assumptions regarding which end is the stack base and in which direction the stack grows as items are pushed.

If `oss` is not a null pointer, on successful completion it will point to a `stack_t` structure that specifies the alternate signal stack that was in effect prior to the call to `sigaltstack()`. The `ss_sp` and `ss_size` members specify the address and size of that stack. The `ss_flags` member specifies the stack's state, and may contain one of the following values:

#### SS\_ONSTACK

The thread is currently executing on the alternate signal stack. Attempts to modify the alternate signal stack while the thread is executing on it fails. This flag must not be modified by threads.

#### SS\_DISABLE

The alternate signal stack is currently disabled.

The value `SIGSTKSZ` is a system default specifying the number of bytes that would be used to cover the usual case when manually allocating an alternate stack area. The value `MINSIGSTKSZ` is defined to be the minimum stack size for a signal handler. In computing an alternate signal stack size, a program should add that amount to its stack requirements to allow for the system implementation overhead. The constants `SS_ONSTACK`, `SS_DISABLE`, `SIGSTKSZ`, and `MINSIGSTKSZ` are defined in `<signal.h>`.



After a successful call to one of the exec functions, there are no alternate signal stacks in the new process image.

### **Returned Value**

If successful, sigaltstack() return 0.

If unsuccessful, sigaltstack() returns -1, and returns the error value in errno. The following are the possible values of errno:

- EINVAL      ss argument is not a null pointer, and the ss\_flags member pointed to by ss contains flags other than SS\_DISABLE.
- ENOMEM     The size of the alternate stack area is less than MINSIGSTKSZ.
- EPERM      An attempt was made to modify an active stack.

### **Related Information**

- “signal.h” on page 41
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sigsetjmp() — Save Stack Environment and Signal Mask” on page 1346
- “sigstack() — Set and/or Get Signal Stack Context” on page 1349

## sigdelset() — Delete a Signal from the Signal Mask

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	

### Format

```
#define _POSIX_SOURCE
#include <signal.h>
```

```
int sigdelset(sigset_t *set, int signal);
```

### General Description

Removes the specified *signal* from the list of signals recorded in *set*.

The `sigdelset()` function is part of a family of functions that manipulate signal sets. *Signal sets* are data objects that let a process keep track of groups of signals. For example, a process can create one signal set to record which signals it is blocking, and another signal set to record which signals are pending. In general, signal sets are used to manipulate groups of signals used by other functions (such as `sigprocmask()`) or to examine signal sets returned by other functions (such as `sigpending()`).

### Returned Value

If the signal is successfully deleted from the signal set, `sigdelset()` returns zero. If *signal* is not supported, `sigdelset()` returns the value `-1` and sets `errno` to `EINVAL`.

### Example

#### CBC3BS16

```
/* CBC3BS16
   This example deletes specific signals.
*/
#define _POSIX_SOURCE
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void catcher(int signum) {
    puts("catcher() has gained control");
}

main() {
    struct sigaction sact;
    sigset_t sigset;

    sigemptyset(&sact.sa_mask);
    sact.sa_flags = 0;
    sact.sa_handler = catcher;
    sigaction(SIGUSR1, &sact, NULL);

    sigfillset(&sigset);
    sigprocmask(SIG_SETMASK, &sigset, NULL);

    puts("before kill());
```

```

kill(getpid(), SIGUSR1);

puts("before unblocking SIGUSR1");
sigdelset(&sigset, SIGUSR1);
sigprocmask(SIG_SETMASK, &sigset, NULL);
puts("after unblocking SIGUSR1");
}

```

### Output

```

before kill()
before unblocking SIGUSR1
catcher() has gained control
after unblocking SIGUSR1

```

### Related Information

- “signal.h” on page 41
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sigaddset() — Add a Signal to the Signal Mask” on page 1312
- “sigemptyset() — Initialize a Signal Mask to Exclude All Signals” on page 1318
- “sigfillset() — Initialize a Signal Mask to Include All Signals” on page 1320
- “sighold() — Add a Signal to a Thread” on page 1322
- “sigismember() — Test If a Signal Is in a Signal Mask” on page 1325
- “sigpending() — Examine Pending Signals” on page 1337
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351
- “signal() — Handle Interrupts” on page 1330
- “sigpause() — Unblock a Signal and Wait for a Signal” on page 1336
- “sigset() — Change a Signal Action and/or a Thread” on page 1343
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351

## sigemptyset() — Initialize a Signal Mask to Exclude All Signals

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	

### Format

```
#define _POSIX_SOURCE
#include <signal.h>
```

```
int sigemptyset(sigset_t *set);
```

### General Description

Initializes a signal set *set* to the empty set. All recognized signals are excluded.

sigemptyset() is part of a family of functions that manipulate signal sets. *Signal sets* are data objects that let a process keep track of groups of signals. For example, a process can create one signal set to record which signals it is blocking, and another signal set to record which signals are pending. Signal sets are used to manipulate groups of signals used by other functions (such as sigprocmask()) or to examine signal sets returned by other functions (such as sigpending()).

### Returned Value

Returns zero to indicate successful completion. There are no documented errno's for this function.

### Example

#### CBC3BS17

```
/* CBC3BS17
   This example initializes a set of signals to an empty set.
*/
#define _POSIX_SOURCE
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

main() {
    struct sigaction sact;

    sigemptyset(&sact.sa_mask);
    sact.sa_flags = 0;
    sact.sa_handler = SIG_IGN;

    sigaction(SIGUSR2, &sact, NULL);

    puts("before kill()");
    kill(getpid(), SIGUSR2);
    puts("after kill()");
}
```

### Output

```
before kill()
after kill()
```

**Related Information**

- “signal.h” on page 41
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sigaddset() — Add a Signal to the Signal Mask” on page 1312
- “sigdelset() — Delete a Signal from the Signal Mask” on page 1316
- “sigfillset() — Initialize a Signal Mask to Include All Signals” on page 1320
- “sigismember() — Test If a Signal Is in a Signal Mask” on page 1325
- “sighold() — Add a Signal to a Thread” on page 1322
- “sigpending() — Examine Pending Signals” on page 1337
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351
- “signal() — Handle Interrupts” on page 1330
- “sigpause() — Unblock a Signal and Wait for a Signal” on page 1336
- “sigpending() — Examine Pending Signals” on page 1337
- “sigset() — Change a Signal Action and/or a Thread” on page 1343
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351

## sigfillset() — Initialize a Signal Mask to Include All Signals

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	

### Format

```
#define _POSIX_SOURCE
#include <signal.h>
```

```
int sigfillset(sigset_t *set);
```

### General Description

Initializes a signal set *set* to the complete set of supported signals.

sigfillset() is part of a family of functions that manipulate signal sets. *Signal sets* are data objects that let a process keep track of groups of signals. For example, a process can create one signal set to record which signals it is blocking, and another signal set to record which signals are pending. Signal sets are used to manipulate groups of signals used by other functions (such as sigprocmask()) or to examine signal sets returned by other functions (such as sigpending()).

### Returned Value

Returns zero to indicate successful completion. There are no documented errno's for this function.

### Example CBC3BS18

```
/* CBC3BS18
   This example initializes a set of signals to a complete set.
*/
#define _POSIX_SOURCE
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

main() {
    sigset_t sigset;
    sigfillset(&sigset);
    sigprocmask(SIG_SETMASK, &sigset, NULL);

    puts("before kill()");
    kill(getpid(), SIGSEGV);
    puts("after kill()");
}
```

### Output

```
before kill()
after kill()
```

## Related Information

- “signal.h” on page 41
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sigaddset() — Add a Signal to the Signal Mask” on page 1312
- “sigdelset() — Delete a Signal from the Signal Mask” on page 1316
- “sigemptyset() — Initialize a Signal Mask to Exclude All Signals” on page 1318
- “sigismember() — Test If a Signal Is in a Signal Mask” on page 1325
- “sigpending() — Examine Pending Signals” on page 1337
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351
- “signal() — Handle Interrupts” on page 1330
- “sighold() — Add a Signal to a Thread” on page 1322
- “sigpause() — Unblock a Signal and Wait for a Signal” on page 1336
- “sigpending() — Examine Pending Signals” on page 1337
- “sigset() — Change a Signal Action and/or a Thread” on page 1343
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351

## sighold() — Add a Signal to a Thread

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>
```

```
int sighold(int sig);
```

### General Description

The sighold() function provides a simplified method for adding the signal specified by the argument, *sig*, to the calling thread's signal mask.

### Returned Value

If successful, sighold() return a zero (0)

If unsuccessful, sighold() returns -1, and returns the error value in errno. The following are the possible values of errno:

**EINVAL**      The value of the argument, *sig*, is not a valid signal type or it is SIGKILL or SIGSTOP.

### Related Information

- “signal.h” on page 41
- “sigaddset() — Add a Signal to the Signal Mask” on page 1312
- “sigpause() — Unblock a Signal and Wait for a Signal” on page 1336
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “sigrelse() — Remove a Signal from a Thread” on page 1342



## sigignore() — Set Disposition to Ignore a Signal

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON) MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>
```

```
int sigignore(int sig);
```

### General Description

The `sigignore()` function provides a simplified method for setting the signal action of the signal specified by the argument, *sig*, to `SIG_IGN`.

### Returned Value

If successful, `sigignore()` return a zero (0).

If unsuccessful, `sigignore()` returns `-1`, and returns the error value in `errno`. The following are the possible values of `errno`:

**EINVAL**      The value of the argument, *sig*, is not a valid signal type or it is `SIGKILL` or `SIGSTOP`.

### Related Information

- “`signal.h`” on page 41
- “`bsd_signal()` — BSD Version of `signal()`” on page 135
- “`sigaction()` — Examine or Change a Signal Action” on page 1295
- “`signal()` — Handle Interrupts” on page 1330
- “`sigset()` — Change a Signal Action and/or a Thread” on page 1343

## siginterrupt() — Allow Signals to Interrupt Functions

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON) MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>
```

```
int siginterrupt(int sig, int flag);
```

### General Description

The `siginterrupt()` function provides a simplified method for changing the restart behavior when a function is interrupted by the signal specified in the argument, *sig*.

The argument, *flag*, serves as a binary switch to enable or disable restart behavior. When *flag* is nonzero, restart behavior will be disabled. Otherwise it is enabled.

### Returned Value

If successful, `siginterrupt()` returns a zero (0).

If unsuccessful, `siginterrupt()` returns `-1`, and returns the error value in `errno`. The following are the possible values of `errno`:

`EINVAL`      The value of the argument, *sig*, is not a valid signal type.

### Related Information

- “`signal.h`” on page 41
- “`sigaction()` — Examine or Change a Signal Action” on page 1295

## sigismember() — Test If a Signal Is in a Signal Mask

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	

### Format

```
#define _POSIX_SOURCE
#include <signal.h>
```

```
int sigismember(const sigset_t *set, int signal);
```

### General Description

Tests whether a specified signal number *signal* is a member of a signal set *set*.

`sigismember()` is part of a family of functions that manipulate signal sets. *Signal sets* are data objects that let a process keep track of groups of signals. For example, a process can create one signal set to record which signals it is blocking, and another signal set to record which signals are pending. Signal sets are used to manipulate groups of signals used by other functions (such as `sigprocmask()`) or to examine signal sets returned by other functions (such as `sigpending()`).

### Returned Value

Returns the value 1 if *signal* is in *set*, and it returns zero if it is not. If *signal* is not a valid signal, `sigismember()` returns the value -1. When unsuccessful, `sigismember()` sets `errno` to `EINVAL`, which indicates that the value of *signal* is not one of the supported signals.

### Example

#### CBC3BS19

```
/* CBC3BS19 */
#define _POSIX_SOURCE
#include <stdio.h>
#include <signal.h>

void check(sigset_t set, int signum, char *signame) {
    printf("%-8s is ", signame);
    if (!sigismember(&set, signum))
        printf("not ");
    puts("in the set");
}

main() {
    sigset_t sigset;

    sigemptyset(&sigset);
    sigaddset(&sigset, SIGUSR1);
    sigaddset(&sigset, SIGKILL);
    sigaddset(&sigset, SIGCHLD);

    check(sigset, SIGUSR1, "SIGUSR1");
    check(sigset, SIGUSR2, "SIGUSR2");
    check(sigset, SIGFPE, "SIGFPE");
}
```

```
    check(sigset, SIGKILL, "SIGKILL");  
}
```

**Output**

```
SIGUSR1  is in the set  
SIGUSR2  is not in the set  
SIGFPE   is not in the set  
SIGKILL  is in the set
```

**Related Information**

- “signal.h” on page 41
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sigaddset() — Add a Signal to the Signal Mask” on page 1312
- “sigdelset() — Delete a Signal from the Signal Mask” on page 1316
- “sigemptyset() — Initialize a Signal Mask to Exclude All Signals” on page 1318
- “sigfillset() — Initialize a Signal Mask to Include All Signals” on page 1320
- “sighold() — Add a Signal to a Thread” on page 1322
- “sigpending() — Examine Pending Signals” on page 1337
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351
- “signal() — Handle Interrupts” on page 1330
- “sigpause() — Unblock a Signal and Wait for a Signal” on page 1336
- “sigpending() — Examine Pending Signals” on page 1337
- “sigset() — Change a Signal Action and/or a Thread” on page 1343
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351

# siglongjmp() — Restore the Stack Environment and Signal Mask

## Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

## Format

```
#define _POSIX_SOURCE
#include <setjmp.h>

void siglongjmp(sigjmp_buf env, int val);
```

## General Description

For a stack environment previously saved in *env* by `sigsetjmp()`, the `siglongjmp()` function restores all the stack environment and, optionally, the signal mask, depending on whether it was saved by `sigsetjmp()`. The `sigsetjmp()` and `siglongjmp()` functions provide a way to perform a nonlocal goto.

*env* is an address for a `sigjmp_buf` structure.

*val* is the return value from `siglongjmp()`.

`siglongjmp()` is similar to `longjmp()`, except for the optional capability of restoring the signal mask. The `sigsetjmp()`—`siglongjmp()` pair, the `setjmp()`—`longjmp()` pair, the `_setjmp()`—`_longjmp()` pair, and the `getcontext()`—`setcontext()` pair cannot be inter-mixed. A stack environment and signal mask saved by `sigsetjmp()` can be restored only by `siglongjmp()`.

A call to `sigsetjmp()` causes the current stack environment including, optionally, the signal mask to be saved in *env*. A subsequent call to `siglongjmp()` restores the saved environment and signal mask (if saved by `sigsetjmp()`) and returns control to a point in the program corresponding to the `sigsetjmp()` call. Execution resumes as if the `sigsetjmp()` call had just returned the given *value*. All variables (except register variables) that are accessible to the function that receives control contain the values they had when you called `siglongjmp()`. The values of register variables are unpredictable. Nonvolatile auto variables that are changed between calls to `sigsetjmp()` and `siglongjmp()` are also unpredictable.

**Note:** If you call `siglongjmp()`, the function in which the corresponding call to `sigsetjmp()` was made must not have returned first. After the function calling `sigsetjmp()` returns, calling `siglongjmp()` causes unpredictable program behavior.

The *value* argument passed to `siglongjmp()` must be nonzero. If you give a zero argument for *value*, `siglongjmp()` substitutes the value 1 in its place.

`siglongjmp()` does not use the normal function call and return mechanisms. `siglongjmp()` restores the saved signal mask only if the *env* parameter was initialized by a call to `sigsetjmp()` with a nonzero *savemask* argument.

## Special Behavior for C++

If `sigsetjmp()` and `siglongjmp()` are used to transfer control in a OS/390 C++ program, the behavior in terms of the destruction of automatic objects is undefined. This applies to both OS/390 C++ and C/C++ OS/390 ILC modules. The use of `sigsetjmp()` and `siglongjmp()` in conjunction with `try()`, `catch()`, and `throw()` is also undefined.

## Returned Value

Returns no value. There are no documented errno's for this function.

## Example

This example saves the stack environment and signal mask at the statement:  
`if(sigsetjmp(mark,1) != 0) ...`

When the system first performs the `if` statement, it saves the environment and signal mask in *mark* and sets the condition to false, because `sigsetjmp()` returns the value 0 when it saves the environment. The program prints the message:  
`sigsetjmp()` has been called

The subsequent call to function `p()` tests for a local error condition, which can cause it to perform `siglongjmp()`. Then control returns to the original `sigsetjmp()` function using the environment saved in *mark* and restores the signal mask. This time the condition is true because `-1` is the return value from `siglongjmp()`. The example then performs the statements in the block and prints: `siglongjmp()` has been called Then it performs your `recover()` function and leaves the program.

```
#define _POSIX_SOURCE
#include <stdio.h>
#include <setjmp.h>

sigjmp_buf mark;

void p(void);
void recover(void);

int main(void)
{
    if (sigsetjmp(mark) != 0) {
        printf("siglongjmp() has been called\n");
        recover();
        exit(1);
    }
    printf("sigsetjmp() has been called\n");
    :
    p();
    :
}

void p(void) {
    int error = 0;
    :
    error = 9;
    :
    if (error != 0)
        siglongjmp(mark, -1);
    :
}

void recover(void) {
    :
}
```

```
}
```

## Related Information

- “setjmp.h” on page 40
- “getcontext() — Get User Context” on page 505
- “longjmp() — Restore Stack Environment” on page 768
- “\_longjmp() — Non-Local Goto” on page 771
- “setcontext() — Restore User Context” on page 1210
- “setjmp() — Preserve Stack Environment” on page 1234
- “\_setjmp() — Set Jump Point for a Non-Local Goto” on page 1237
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “sigsetjmp() — Save Stack Environment and Signal Mask” on page 1346
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351
- “swapcontext() — Save and Restore User Context” on page 1472

## signal() — Handle Interrupts

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 POSIX.4a XPG4 XPG4.2	both	

### Format

```
#include <signal.h>
```

```
void(*signal (int sig, void(*func)(int)))(int);
```

### General Description

Allows a process to choose one of several ways to handle an interrupt signal *sig* from the operating system or from the `raise()` function.

The *sig* argument must be one of the macros defined in the `signal.h` header file. See Table 33 on page 1332.

The *func* argument must be one of the macros, `SIG_DFL` or `SIG_IGN`, defined in the `signal.h` header file, or a function address.

If the value of *func* is `SIG_DFL`, default handling for that signal will occur. If the value of *func* is `SIG_IGN`, the signal will be ignored. Otherwise, *func* points to a function to be called when that signal occurs. Such a function is called a *signal handler*.

When a signal occurs, if *func* points to a function:

1. First the equivalent of `signal(sig, SIG_DFL)`; is executed or an implementation-defined blocking of the system is performed. (If the value of *sig* is `SIGILL`, the occurrence of the reset to `SIG_DFL` is implementation defined.)
2. Next, the equivalent of `(*func)(sig)`; is executed. The function *func* may terminate by executing a return statement or by calling the `abort()`, `exit()`, or `longjmp()` function. If *func* executes a return statement and the value of *sig* was `SIGFPE` or any other implementation-defined value corresponding to a computational exception, the behavior is undefined. Otherwise, the program will resume execution at the point it was interrupted.

If a signal occurs for a reason other than having called the `abort()` or `raise()` function, the behavior is undefined if the signal handler calls any function in the standard library other than the `signal()` function itself (with a first argument of the signal number corresponding to the signal that caused the invocation of the handler). Behavior is also undefined if the signal handler refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type `volatile sig_atomic_t`. Furthermore, if such a call to the `signal()` function returns `SIG_ERR`, the value of `errno` is indeterminate.



At program startup, the equivalent of `signal(sig, SIG_IGN);` may be executed for some selected signals. The equivalent of `signal(sig, SIG_DFL);` is executed for all other signals.

The action taken when the interrupt signal is received depends on the value of *func*.

Value	Meaning
-------	---------

SIG_DFL	Default handling for the signal will occur.
---------	---

SIG_IGN	The signal is to be ignored.
---------	------------------------------

As of Language Environment Release 3, the defaults for SIGUSR1, SIGUSR2, SIGINT, and SIGTERM are changed from the signal being ignored to abnormal termination. To compensate for this change, you would explicitly register that the signal is to be ignored, via a call sequence such as:

```
signal(SIGUSR1, SIG_IGN);
signal(SIGUSR2, SIG_IGN);
signal(SIGINT, SIG_IGN);
signal(SIGTERM, SIG_IGN);
```

These calls may be made either in the source or they can be made from the HLL user exit CEEBINT, which will require a re-link.

### Special Behavior for POSIX

For an OS/390 UNIX C application running POSIX(ON), the interrupt signal can also come from `kill()` or from another process. A program can use `sigaction()` to establish a signal handler; `sigaction()` blocks the signal while the signal handler has control. If you use `signal()` to establish a signal handler, the signal reverts back to the default action. If you want the signal handler to get control for the next signal of this type, you must reissue `signal()`.

See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information about using POSIX support.

`signal(sig, func)` is equivalent to `sigaction(sig, &act, NULL)`, where *act* points to a `sigaction` structure containing an `sa_action` of *func*, an `sa_mask` by `sigemptyset()`, and an `sa_flags` containing `_SA_OLD_STYLE`.

For a list of considerations for coding signal-catching functions that will support asynchronous signals, refer to “`sigaction()` — Examine or Change a Signal Action” on page 1295.

The *sig* argument must be one of the macros defined in the `signal.h` header file.

### Special Behavior for C++

- The behavior when mixing signal-handling with C++ exception handling is undefined. Also, the use of signal-handling with constructors and destructors is undefined.
- C++ and C language linkage conventions are incompatible, and therefore `signal()` cannot receive C++ function pointers. If you attempt to pass a C++ function pointer to `signal()`, the compiler will flag it as an error. Therefore, to use

the `signal()` function in the C++ language, you must ensure that signal handler routines established have C linkage, by declaring them as `extern "C"`.

The signals supported are listed below.

Table 33. Signals Supported by C or C++ — POSIX(OFF)

Value	Default Action	Meaning
SIGABND	1	Abend
SIGABRT	1	Abnormal termination (sent by <code>abort()</code> )
SIGFPE	1	Arithmetic exceptions that are not masked, for example, overflow, division by zero, and incorrect operation
SIGILL	1	Detection of an incorrect function image
SIGINT	1	Interactive attention
SIGSEGV	1	Incorrect access to memory
SIGTERM	1	Termination request sent to the program
SIGUSR1	1	Intended for use by user applications
SIGUSR2	1	Intended for use by user applications
SIGIOERR	2	A serious I/O error was detected. (When running on MVS 4.3, this signal is not supported by the kernel. It will be mapped to SIGIO. An application that uses SIGIO and SIGIOERR may have undesirable results. This limitation is removed when running on MVS 5.1, as SIGIO and SIGIOERR are both supported.)

In Table 33, the **Default Actions** are:

- 1 Normal termination of the process.
- 2 Ignore the signal.

If a signal catcher for a SIGABND, SIGFPE, SIGILL or SIGSEGV signal runs as a result of a program check or an ABEND, and the signal catcher executes a RETURN statement, the process will be terminated.

### Returned Value

If successful, the call to `signal()` returns the most recent value of *func*. Otherwise, if there is an error in the call, `signal()` returns a value of SIG\_ERR and a positive value in `errno`.

There are no documented `errno`s for this function. If an error occurs, issue `perror()` using the `errno` value.

## Example

### CBC3BS20

```

/* CBC3BS20
   This example shows you how to establish a signal handler.
*/
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#define ONE_K 1024

#define OUT_OF_STORAGE      (SIGUSR1)

/* The SIGNAL macro does a signal() checking the return code */
#define SIGNAL(handler, StrCln) { \
    if (signal((handler), (StrCln)) == SIG_ERR) { \
        perror("Could not signal user signal"); \
        abort(); \
    } \
}

#ifdef __cplusplus          /* the __cplusplus macro      */
extern "C" void StrCln(int); /* is automatically defined */
#else                      /* by the C++/MVS compiler  */
void StrCln(int);
#endif

void DoWork(char **, int);

int main(int argc, char *argv[]) {
    int size;
    char *buffer;

    SIGNAL(OUT_OF_STORAGE, StrCln);

    if (argc != 2) {
        printf("Syntax: %s size \n", argv[0]);
        return(-1);
    }

    size = atoi(argv[1]);

    DoWork(&buffer, size);
    return(0);
}

void StrCln(int SIG_TYPE) {
    printf("Failed trying to malloc storage\n");
    SIGNAL(SIG_TYPE, SIG_DFL);
    exit(0);
}

void DoWork(char **buffer, int size) {
    int rc;

    while (*buffer != NULL)
        *buffer = (char *)malloc(size*ONE_K);
    if (*buffer == NULL) {
        if (raise(OUT_OF_STORAGE)) {
            perror("Could not raise user signal");
            abort();
        }
    }
}

```

```
    return;  
}
```

### Related Information

- Signal Handling in the *OS/390 C/C++ Programming Guide*
- “signal.h” on page 41
- “abort() — Stop a Program” on page 71
- “atexit() — Register Program Termination Function” on page 116
- “bsd\_signal() — BSD Version of signal()” on page 135
- “exit() — End Program” on page 330
- “kill() — Send a Signal to a Process” on page 728
- “pthread\_kill() — Send a Signal to a Thread” on page 984
- “raise() — Raise Signal” on page 1074
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “sigignore() — Set Disposition to Ignore a Signal” on page 1323
- “sigset() — Change a Signal Action and/or a Thread” on page 1343
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351
- “waitid() — Wait for Child Process to Change State” on page 1690
- “wait3() — Wait for Child Process to Change State” on page 1696

## \_\_signgam() — Return signgam Reference

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#define _XOPEN_SOURCE
#include <math.h>

int *__signgam(void);
#define signgam (*__signgam())
```

### General Description

The `__signgam()` function returns the address of the calling thread's storage for the *signgam* external variable used by the `gamma()` and `lgamma()` functions. This extended mechanism is necessary for multithreaded processes which use either of these two functions, since each thread has its own instance of *signgam*. The `<math.h>` header defines *signgam* to an invocation of `__signgam()`, so generally, all references to *signgam* will be mapped to calls to `__signgam()`. If the user eliminates this definition, either by not including the header, or by using `#undef`, then references to *signgam* will refer to the actual *signgam* external variable, which contains the *signgam* value for the IPT only. In the absence of the definition of *signgam* to a call to `__signgam()`, *signgam* values in threads other than the IPT are inaccessible.

### Returned Value

`__signgam()` is always successful.

### Related Information

- “math.h” on page 35
- “gamma() — Calculate Gamma Function” on page 497
- “lgamma() — Log Gamma Function” on page 744

## sigpause() — Unblock a Signal and Wait for a Signal

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON) MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>
```

```
int sigpause(int sig);
```

### General Description

The sigpause() function provides a simplified method for removing a signal, specified by the argument 'sig', from the calling's threads signal mask and suspending this thread until a signal is received whose action is either to execute a signal catcher function or to terminate the process.

### Returned Value

If successful, sigpause() returns -1, and sets errno to EINTR.

If unsuccessful, sigpause() returns -1, and returns the error value in errno. The following are the possible values of errno:

EINVAL      The value of the argument 'sig' is not a valid signal type or it is SIGKILL.

### Related Information

- “signal.h” on page 41
- “pause() — Suspend a Process Pending a Signal” on page 899
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351

## sigpending() — Examine Pending Signals

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <signal.h>
```

```
int sigpending(sigset_t *set);
```

### General Description

Returns the union of the set of signals that are blocked from delivery and pending for the calling thread and the set that are pending for the process. If there is only one thread, it does the same for the calling process. This information is represented as a signal set stored in *set*. For more information on examining the signal set pointed to by *set*, see “sigismember() — Test If a Signal Is in a Signal Mask” on page 1325.

### Returned Value

If successful, sigpending() returns zero. If unsuccessful, it returns the value -1. There are no documented errno values for this function.

### Example

#### CBC3BS22

```
/* CBC3BS22
   This example returns blocked or pending signals.
*/
#define _POSIX_SOURCE
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void catcher(int signum) {
    puts("inside catcher!");
}

void check_pending(int signum, char *signame) {
    sigset_t sigset;

    if (sigpending(&sigset) != 0)
        perror("sigpending() error");
    else if (sigismember(&sigset, signum))
        printf("a %s signal is pending\n", signame);
    else
        printf("no %s signals are pending\n", signame);
}

main() {
    struct sigaction sigact;
    sigset_t sigset;

    sigemptyset(&sigact.sa_mask);
```

```
sigact.sa_flags = 0;
sigact.sa_handler = catcher;
if (sigaction(SIGUSR1, &sigact, NULL) != 0)
    perror("sigaction() error");
else {
    sigemptyset(&sigset);
    sigaddset(&sigset, SIGUSR1);
    if (sigprocmask(SIG_SETMASK, &sigset, NULL) != 0)
        perror("sigprocmask() error");
    else {
        puts("SIGUSR1 signals are now blocked");
        kill(getpid(), SIGUSR1);
        printf("after kill: ");
        check_pending(SIGUSR1, "SIGUSR1");
        sigemptyset(&sigset);
        sigprocmask(SIG_SETMASK, &sigset, NULL);
        puts("SIGUSR1 signals are no longer blocked");
        check_pending(SIGUSR1, "SIGUSR1");
    }
}
```

### Output

```
SIGUSR1 signals are now blocked
after kill: a SIGUSR1 signal is pending
inside catcher!
SIGUSR1 signals are no longer blocked
no SIGUSR1 signals are pending
```

### Related Information

- “signal.h” on page 41
- “sigismember() — Test If a Signal Is in a Signal Mask” on page 1325
- “sigprocmask() — Examine or Change a Thread” on page 1339



## sigprocmask() — Examine or Change a Thread

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <signal.h>
```

```
int sigprocmask(int option, const sigset_t *new_set, sigset_t *old_set);
```

### General Description

Examines, changes, or examines and changes the signal mask of the calling thread. If there is only one thread, it does the same for the calling process.

Typically, `sigprocmask(SIG_BLOCK, ..., ...)` is used to block signals during a critical section of code. At the end of the critical section of code, `sigprocmask(SIG_SETMASK, ..., ...)` is used to restore the mask to the previous value returned by `sigprocmask(SIG_BLOCK, ..., ...)`.

*option* Indicates the way in which the existing set of blocked signals should be changed. The following are the possible values for *option*, defined in the `signal.h` header file:

#### SIG\_BLOCK

Indicates that the set of signals given by *new\_set* should be blocked, in addition to the set currently being blocked.

#### SIG\_UNBLOCK

Indicates that the set of signals given by *new\_set* should not be blocked. These signals are removed from the current set of signals being blocked.

#### SIG\_SETMASK

Indicates that the set of signals given by *new\_set* should replace the old set of signals being blocked.

*new\_set* Points to a signal set giving the new signals that should be blocked or unblocked (depending on the value of *option*) or it points to the new signal mask if the option was *sig\_setmask*. Signal sets are described in “`sigemptyset()` — Initialize a Signal Mask to Exclude All Signals” on page 1318. If *new\_set* is a NULL pointer, the set of blocked signals is not changed. `sigprocmask()` determines the current set and returns this information in *\*old\_set*. If *new\_set* is NULL, the value of *option* is not significant.

The signal set manipulation functions: `sigemptyset()`, `sigfillset()`, `sigaddset()`, and `sigdelset()` must be used to establish the new signal set pointed to by *new\_set*.

*old\_set* Points to a memory location where `sigprocmask()` can store a signal set. If *new\_set* is NULL, *old\_set* returns the current set of signals being blocked. When *new\_set* is not NULL, the set of signals pointed to by *old\_set* is the previous set.

If there are any pending unblocked signals, either at the process level or at the current thread's level after `sigprocmask()` has changed the signal mask, then at least one of those signals is delivered to the thread before `sigprocmask()` returns.

The signals SIGKILL or SIGSTOP cannot be blocked. If you attempt to use `sigprocmask()` to block these signals, the attempt is simply ignored. `sigprocmask()` does not return an error status.

SIGFPE, SIGILL, and SIGSEGV signals that are not artificially generated by `kill()`, `killpg()`, `raise()`, or `pthread_kill()` (that is, were generated by the system as a result of a hardware or software exception) will not be blocked.

If an artificially raised SIGFPE, SIGILL, or SIGSEGV signal is pending and blocked when an exception causes another SIGFPE, SIGILL, or SIGSEGV signal, both the artificial and exception-caused signals may be delivered to the application.

If `sigprocmask()` fails, the signal mask of the thread is not changed.

## Returned Value

If successful, `sigprocmask()` returns zero. If unsuccessful, it returns the value `-1` and sets `errno` to `EINVAL`, which indicates that *option* does not have one of the recognized values.

## Example CBC3BS23

```
/* CBC3BS23
   This example changes the signal mask.
*/
#define _POSIX_SOURCE
#include <signal.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>

void catcher(int signum) {
    puts("inside catcher");
}

main() {
    time_t start, finish;
    struct sigaction sact;
    sigset_t new_set, old_set;
    double diff;

    sigemptyset(&sact.sa_mask);
    sact.sa_flags = 0;
    sact.sa_handler = catcher;
    if (sigaction(SIGALRM, &sact, NULL) != 0)
        perror("sigaction() error");
    else {
        sigemptyset(&new_set);
        sigaddset(&new_set, SIGALRM);
        if (sigprocmask(SIG_BLOCK, &new_set, &old_set) != 0)
```

```

    perror("1st sigprocmask() error");
else {
    time(&start);
    printf("SIGALRM signals blocked at %s", ctime(&start));
    alarm(1);

    do {
        time(&finish);
        diff = difftime(finish, start);
    } while (diff < 10);
    if (sigprocmask(SIG_SETMASK, &old_set, NULL) != 0)
        perror("2nd sigprocmask() error");
    else
        printf("SIGALRM signals unblocked at %s", ctime(&finish));
    }
}
}

```

### Output

```

SIGALRM signals blocked at Fri Jun 16 12:24:19 1995
inside catcher
SIGALRM signals unblocked at Fri Jun 16 12:24:29 1995

```

### Related Information

- “signal.h” on page 41
- “raise() — Raise Signal” on page 1074
- “kill() — Send a Signal to a Process” on page 728
- “killpg() — Send a Signal to a Process Group” on page 731
- “pthread\_kill() — Send a Signal to a Thread” on page 984
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sigaddset() — Add a Signal to the Signal Mask” on page 1312
- “sigdelset() — Delete a Signal from the Signal Mask” on page 1316
- “sigemptyset() — Initialize a Signal Mask to Exclude All Signals” on page 1318
- “sigfillset() — Initialize a Signal Mask to Include All Signals” on page 1320
- “sighold() — Add a Signal to a Thread” on page 1322
- “sigismember() — Test If a Signal Is in a Signal Mask” on page 1325
- “sigpending() — Examine Pending Signals” on page 1337
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351
- “signal() — Handle Interrupts” on page 1330
- “sigpause() — Unblock a Signal and Wait for a Signal” on page 1336
- “sigpending() — Examine Pending Signals” on page 1337
- “sigrelse() — Remove a Signal from a Thread” on page 1342
- “sigset() — Change a Signal Action and/or a Thread” on page 1343
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351

## sigrelse() — Remove a Signal from a Thread

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>
```

```
int sigrelse(int sig);
```

### General Description

The sigrelse() function provides a simplified method for removing the signal specified by the argument, *sig*, from the calling thread's signal mask.

### Returned Value

If successful, sigrelse() return a zero (0)

If unsuccessful, sigrelse() returns -1, and returns the error value in errno. The following are the possible values of errno:

**EINVAL**      The value of the argument, *sig*, is not a valid signal type or it is SIGKILL or SIGSTOP.

### Related Information

- “signal.h” on page 41
- “sighold() — Add a Signal to a Thread” on page 1322
- “sigpause() — Unblock a Signal and Wait for a Signal” on page 1336
- “sigprocmask() — Examine or Change a Thread” on page 1339

## sigset() — Change a Signal Action and/or a Thread

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON) MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>
```

```
void (*sigset(int sig, void (*disp)(int)))(int);
```

### General Description

The `sigset()` function provides a simplified method for changing the action associated with a specific signal and unblock the signal, or to block this signal.

*sig* The number of a recognized signal. `sigset()` sets the action associated with this signal and unblock this signal, or adds this signal to the calling thread's signal mask (thus blocking this signal). Refer to Table 32 on page 1295 for a list of the supported values of *sig*.

The value of *sig* can be any valid signal type except SIGKILL or SIGSTOP.

*disp* There are four possible value that *disp* can have. Three are actions that can be associated with the signal, *sig*: SIG\_DFL, SIG\_IGN, or a pointer to a function. The fourth value is not a signal action, but a flag to `sigset()` that affects whether the signal action is changed.

The values that *disp* is permitted to have are:

SIG\_DFL Set the signal action to the signal-specific default.

- The default actions for each signal is shown in Table 32 on page 1295.
- If *disp* is set to SIG\_DFL, `sigset()` will change the signal action associated with *sig* and remove this signal from the calling thread's signal mask (thus unblocking this signal).
- If the default action is to stop the process, the execution of that process is temporarily suspended. When a process stops, a SIGCHLD signal will be generated for its parent process, unless the parent process has set the **SA\_NOCLDSTOP** flag. While a process is stopped, any additional signals that are sent to the process will not be delivered until the process is continued, except SIGKILL which always terminates the receiving process. A process that is a member of an orphaned process group will not be allowed to stop in response to the SIGTSTP, SIGTTIN, or SIGTTOU signals. In cases where delivery of one of these signals would stop such a process, the signal will be discarded.

- Setting a signal action to SIG\_DFL for a signal that is pending, and whose default action is to ignore the signal (for example SIGCHLD), will cause the pending signal to be discarded.

SIG\_IGN Set the signal action to ignore the signal.

- Delivery of the signal will have no effect on the process.
- If *disp* is set to SIG\_IGN, sigset() will change the signal action associated with *sig* and remove this signal from the calling thread's signal mask (thus unblocking this signal).
- Setting a signal action to SIG\_IGN for a signal that is pending will cause the pending signal to be discarded. This provides the ability to discard signals that are found to be blocked and pending by sigpending().
- If *sig* is SIGCHLD, child processes of the calling process will not be transformed into 'zombie' processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited from children that were transformed into 'zombie' processes, it will block until all of its children terminate. The wait(), waitid(), or waitpid() function will fail and set errno to **ECHILD**.

SIG\_HOLD Set the calling thread's signal mask to block signal, *sig*.

- The signal action associated with *sig* is not changed.

Pointer to function

Set the signal action to catch the signal.

- sigset() will change the signal action associated with *sig* and remove this signal from the calling thread's signal mask (thus unblocking this signal).
- On delivery of the signal, the receiving process is to execute the signal-catching function at the specified address. After returning from the signal-catching function, the receiving process will resume execution at the point at which it was interrupted.
- The signal-catching function specified by *disp* is invoked as:

```
void function(int signo);
```

Where *function* is the specified signal-catching function and *signo* is the signal number of the signal being delivered.

After an action has been specified for a particular signal, using sigset(), it remains installed until it is explicitly changed with another call to sigset(), sigaction(), signal(), one of the exec functions, bsd\_signal(), or sigignore().

### Special Behavior for C++

- The behavior when mixing signal-handling with C++ exception handling is undefined. Also, the use of signal-handling with constructors and destructors is undefined.
- C++ and C language linkage conventions are incompatible, and therefore sigaction() cannot receive C++ function pointers. If you attempt to pass a C++ function pointer to sigaction(), the compiler will flag it as an error. Therefore, to

use the `sigaction()` function in the C++ language, you must ensure that signal handler routines established have C linkage, by declaring them as `extern "C"`.

### Returned Value

If successful, `sigset()` returns `SIG_HOLD` if the signal had been blocked and the signal's previous action if it had not been blocked.

If unsuccessful, `sigset()` returns `SIG_ERR` and returns the error value in `errno`. The following are the possible values of `errno`:

`EINVAL`      The value of the argument *sig* was not a valid signal type, or it was `SIGKILL` ignore a signal that cannot be ignored.

### Related Information

- “`signal.h`” on page 41
- “`bsd_signal()` — BSD Version of `signal()`” on page 135
- “`sigaction()` — Examine or Change a Signal Action” on page 1295
- “`sighold()` — Add a Signal to a Thread” on page 1322
- “`signal()` — Handle Interrupts” on page 1330
- “`sigprocmask()` — Examine or Change a Thread” on page 1339

## sigsetjmp() — Save Stack Environment and Signal Mask

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <setjmp.h>
```

```
int sigsetjmp(sigjmp_buf env, int savemask);
```

### General Description

Saves the current stack environment including, optionally, the current signal mask. The stack environment and signal mask saved by `sigsetjmp()` can subsequently be restored by `siglongjmp()`.

*env* is an address for a `sigjmp_buf` structure. *savemask* is a flag used to determine if the signal mask is to be saved. If it has a value of 0, the current signal mask is not to be saved or restored as part of the environment. Any other value means the current signal mask is saved and restored.

`sigsetjmp()` is similar to `setjmp()` and `_setjmp()`, except for the optional capability of saving the signal mask. Like `setjmp()` and `longjmp()`, the `sigsetjmp()` and `siglongjmp()` functions provide a way to perform a nonlocal `goto`.

The `sigsetjmp()`—`siglongjmp()` pair, the `setjmp()`—`longjmp()` pair, the `_setjmp()`—`_longjmp()` pair and the `getcontext()`—`setcontext()` pair cannot be inter-mixed. A stack environment and signal mask saved by `sigsetjmp()` can be restored only by `siglongjmp()`.

A call to `sigsetjmp()` causes it to save the current stack environment in *env*. If the value of the *savemask* parameter is nonzero, it will also save the current signal mask in *env*. A subsequent call to `siglongjmp()` restores the saved environment and signal mask (if saved by `sigsetjmp()`), and returns control to a point corresponding to the `sigsetjmp()` call. The values of all variables (except register variables) accessible to the function receiving control contain the values they had when `siglongjmp()` was called. The values of register variables are unpredictable. Nonvolatile auto variables that are changed between calls to `sigsetjmp()` and `siglongjmp()` are also unpredictable.

**Note:** Ensure that the function that calls `sigsetjmp()` does not return before you call the corresponding `siglongjmp()` function. Calling `siglongjmp()` after the function calling `sigsetjmp()` returns causes unpredictable program behavior.

### Special Behavior for C++

If `sigsetjmp()` and `siglongjmp()` are used to transfer control in a OS/390 C++ program, the behavior in terms of the destruction of automatic objects is undefined. This applies to both OS/390 C++ and C/C++ OS/390 ILC modules. The use of



sigsetjmp() and siglongjmp() in conjunction with try(), catch(), and throw() is also undefined.

## Returned Value

Returns a value of zero when it is invoked to save the stack environment and signal mask. sigsetjmp() returns the value *val*, specified on siglongjmp() (or 1 if the value of *val* is zero), when siglongjmp() causes control to be transferred to the place in the user's program where sigsetjmp() was issued. There are no documented errors for this function.

## Example

The following saves the stack environment and signal mask at the statement:

```
if(sigsetjmp(mark,1) != 0) ...
```

When the system first performs the if statement, it saves the environment and signal mask in *mark* and sets the condition to false because sigsetjmp() returns zero when it saves the environment. The program prints the message:

```
sigsetjmp() has been called
```

The subsequent call to function *p()* tests for a local error condition, which can cause it to perform siglongjmp(). Then control returns to the original sigsetjmp() function using the environment saved in *mark* and the restored signal mask. This time the condition is true because -1 is the return value from siglongjmp(). The program then performs the statements in the block and prints:

```
siglongjmp() has been called
```

Then the program performs the sample *recover()* function and exits.

```
#define _POSIX_SOURCE
#include <stdio.h>
#include <setjmp.h>

sigjmp_buf mark;

void p(void);
void recover(void);

int main(void)
{
    if (sigsetjmp(mark,1) != 0) {
        printf("siglongjmp() has been called\n");
        recover();
        exit(1);
    }
    printf("sigsetjmp() has been called\n");
    :
    p();
    :
}

void p(void)
{
    int error = 0;
    :
    error = 9;
    :
    if (error != 0)
        siglongjmp(mark, -1);
}
```

```
    :  
}  
  
void recover(void)  
{  
    :  
}
```

### **Related Information**

- “setjmp.h” on page 40
- “getcontext() — Get User Context” on page 505
- “longjmp() — Restore Stack Environment” on page 768
- “\_longjmp() — Non-Local Goto” on page 771
- “setcontext() — Restore User Context” on page 1210
- “setjmp() — Preserve Stack Environment” on page 1234
- “\_setjmp() — Set Jump Point for a Non-Local Goto” on page 1237
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “siglongjmp() — Restore the Stack Environment and Signal Mask” on page 1327
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351
- “swapcontext() — Save and Restore User Context” on page 1472

## sigstack() — Set and/or Get Signal Stack Context

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON)

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <signal.h>
```

```
int sigstack(struct sigstack *ss, struct sigstack *oss);
```

### General Description

The `sigstack()` function allows the calling thread to indicate, to the system, an area of its address space to be used for processing signals received by this thread.

**Note:** To explicitly declare that a signal catcher is to run on the alternate signal stack, the `SA_ONSTACK` flag must be set in the `sa_flags` when the signal action is set via `sigaction()`.

If the `ss` argument is not a null pointer, it must point to a `sigstack` structure. The length of the application-supplied stack must be at least `SIGSTKSZ` bytes. If the alternate signal stack overflows, the resulting behavior is undefined.

- The value of the `ss_onstack` member indicates whether the thread wants the system to use an alternate signal stack when delivering signals.
- The value of the `ss_sp` member indicates the desired location of the alternate signal stack area in the process' address space.
- If the `ss` argument is a null pointer, the current alternate signal stack context is not changed.

If the `oss` argument is not a null pointer, it must point to a `sigstack` structure into which the current alternate signal stack context is placed. The value stored in the `ss_onstack` member of this `sigstack` structure will be nonzero if the thread is currently executing on the alternate signal stack. If the `oss` argument is a null pointer, the current alternate signal stack context is not returned.

When a signal's action indicates its handler should execute on the alternate signal stack (specified by calling `sigaction()`), the implementation checks to see if the thread is currently executing on the alternate signal stack. If it is not, the system will switch to the alternate signal stack for the duration of the signal handler's execution.

After a successful call to one of the `exec` functions, there are no alternate signal stacks in the new process image.

### Returned Value

If successful, `sigstack()` returns 0.

If unsuccessful, `sigstack()` returns `-1`, and returns the error value in `errno`. The following are the possible values of `errno`:

**EPERM**      An attempt was made to modify an active stack.

**Related Information**

- “signal.h” on page 41
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sigaltstack() — Set and/or Get Signal Alternate Stack Context” on page 1314
- “sigsetjmp() — Save Stack Environment and Signal Mask” on page 1346

## sigsuspend() — Change Mask and Suspend the Thread

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	POSIX(ON) MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <signal.h>
```

```
int sigsuspend(const sigset_t *mask);
```

### General Description

Replaces the current signal mask of a thread with the signal set given by *mask* and then suspends execution of the calling thread. The thread does not resume running until a signal is delivered whose action is either to execute a signal-handling function or to end the process. (Signal sets are described in more detail in “sigemptyset() — Initialize a Signal Mask to Exclude All Signals” on page 1318.)

The signal mask indicates a set of signals that should be blocked. Such signals do not “wake up” the suspended function. The signals SIGSTOP and SIGKILL cannot be blocked or ignored; they are delivered to the thread no matter what the *mask* argument specifies.

If an incoming unblocked signal ends the thread, sigsuspend() never returns to the caller. If an incoming signal is handled by a signal-handling function, sigsuspend() returns after the signal-handling function returns. The signal mask of the thread is restored to whatever it was before sigsuspend() was called, unless the signal-handling functions explicitly changed the mask.

This function is supported only in a POSIX program. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information.

### Returned Value

If sigsuspend() returns, it always returns a value of –1. When unsuccessful, sigsuspend() sets errno to EINTR, which indicates that a signal was received and handled successfully.

### Example

#### CBC3BS25

```
/* CBC3BS25
   This example replaces the signal mask and then suspends execution.
*/
#define _POSIX_SOURCE
#include <stdio.h>
#include <signal.h>
#include <time.h>
#include <unistd.h>

void catcher(int signum) {
    switch (signum) {
        case SIGUSR1: puts("catcher caught SIGUSR1");
```

```

        break;
    case SIGUSR2: puts("catcher caught SIGUSR2");
        break;
    default:      printf("catcher caught unexpected signal %d\n",
                        signal);
}
}

main() {
    sigset_t sigset;
    struct sigaction sact;
    time_t    t;

    if (fork() == 0) {
        sleep(10);
        puts("child is sending SIGUSR2 signal - which should be blocked");
        kill(getppid(), SIGUSR2);
        sleep(5);
        puts("child is sending SIGUSR1 signal - which should be caught");
        kill(getppid(), SIGUSR1);
        exit(0);
    }

    sigemptyset(&sact.sa_mask);
    sact.sa_flags = 0;
    sact.sa_handler = catcher;
    if (sigaction(SIGUSR1, &sact, NULL) != 0)
        perror("1st sigaction() error");

    else if (sigaction(SIGUSR2, &sact, NULL) != 0)
        perror("2nd sigaction() error");

    else {
        sigfillset(&sigset);
        sigdelset(&sigset, SIGUSR1);
        time(&t);
        printf("parent is waiting for child to send SIGUSR1 at %s",
               ctime(&t));
        if (sigsuspend(&sigset) == -1)
            perror("sigsuspend() returned -1 as expected");
        time(&t);
        printf("sigsuspend is over at %s", ctime(&t));
    }
}

```

## Output

```

parent is waiting for child to send SIGUSR1 at Fri Jun 16 12:30:57 1995
child is sending SIGUSR2 signal - which should be blocked
child is sending SIGUSR1 signal - which should be caught
catcher caught SIGUSR2
catcher caught SIGUSR1
sigsuspend() returned -1 as expected: Interrupted function call
sigsuspend is over at Fri Jun 16 12:31:12 1995

```

## Related Information

- “signal.h” on page 41
- “bsd\_signal() — BSD Version of signal()” on page 135
- “raise() — Raise Signal” on page 1074
- “kill() — Send a Signal to a Process” on page 728
- “killpg() — Send a Signal to a Process Group” on page 731
- “pause() — Suspend a Process Pending a Signal” on page 899

- “pthread\_kill() — Send a Signal to a Thread” on page 984
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sigaddset() — Add a Signal to the Signal Mask” on page 1312
- “sigdelset() — Delete a Signal from the Signal Mask” on page 1316
- “sigemptyset() — Initialize a Signal Mask to Exclude All Signals” on page 1318
- “sigfillset() — Initialize a Signal Mask to Include All Signals” on page 1320
- “sigignore() — Set Disposition to Ignore a Signal” on page 1323
- “signal() — Handle Interrupts” on page 1330
- “sigpause() — Unblock a Signal and Wait for a Signal” on page 1336
- “sigpending() — Examine Pending Signals” on page 1337
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “sigset() — Change a Signal Action and/or a Thread” on page 1343

## sigtimedwait() — Wait for Queued Signals

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 UNIX

### Format

```
#define _XOPEN_SOURCE 500
#define _XOPEN_REALTIME
#include <signal.h>
```

```
int sigtimedwait(const sigset_t *set, siginfo_t *info
                 const struct timespec *timeout);
```

### General Description

The `sigtimedwait()` function selects a pending signal from the `sigset_t` object (signal set) pointed to by `set`, automatically clearing it from the system's set of pending signals, and returning that signal number. If there are multiple pending signals, the lowest numbered signal will be selected.

If no signal in the signal set is pending at the time of the call to `sigtimedwait()`, the thread is suspended until one or more of the signals specified in the signal set become pending or until it is interrupted by an unblocked, caught signal. The signals defined in the `sigset_t` object (signal set) pointed to by `set` may be unblocked during the call to this routine and will be blocked when the thread returns from the call unless some other thread is currently waiting for one of those signals.

If more than one thread is using `sigtimedwait()` to wait for the same signal, only one of these threads will return from this routine with the signal number, until a second signal of the same type is received.

The function `sigtimedwait()` behaves the same as the `sigwait()` function if the `info` argument is `NULL`. If the `info` argument is not `NULL` then in addition to behaving the same as `sigwait()`, `sigtimedwait()` places the selected signal number in the `si_signo` member, places the cause of the signal in the `si_code` member, and, if any value is queued to the selected signal, `sigtimedwait()` will place it in the `si_value` member of `info`. However, if there is no value queued for the selected signal then the content of `si_value` is undefined.

If the `sigtimedwait()` function finds that none of the signals specified by `set` are pending, it waits for the time interval specified in the `timespec` structure referenced by `timeout`. If the `timespec` structure pointed to by `timeout` is zero-valued and if none of the signals specified by `set` are pending, then `sigtimedwait()` returns immediately with an error. A `timespec` with the `tv_sec` field set with `INT_MAX`, as defined in `<limits.h>`, will cause the `sigtimedwait()` service to wait until a signal is received. If `timeout` is the `NULL` pointer, the behavior is not necessarily the same on all platforms but for this platform it will be treated the same as when `timespec` structure was supplied with the `tv_sec` field set with `INT_MAX`.



## Returned Value

If successful, sigtimedwait() returns the signal number.

If unsuccessful, sigtimedwait() returns -1 and returns the error value in errno. The following are the possible values of errno:

EAGAIN	No signal specified by set was generated within the specified time out period.
EINTR	The wait was interrupted by an unblocked, caught signal. No further waiting will occur for this call. sigtimedwait() can be re-issued to begin waiting again.
EINVAL	set points to a <b>sigset_t</b> that contains a signal number that is either not valid or not supported.

## Related Information

- “signal.h” on page 41
- “time.h” on page 51
- “pause() — Suspend a Process Pending a Signal” on page 899
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sigpending() — Examine Pending Signals” on page 1337
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351
- “sigwait() — Wait for an Asynchronous Signal” on page 1356

## sigwait() — Wait for an Asynchronous Signal

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1	both	MVS 4.3

### Format

```
#define _OPEN_THREADS
#include <signal.h>

int sigwait(sigset_t *set);
```

### General Description

Causes a thread to wait for an asynchronous signal by choosing a pending signal from *set*, automatically clearing it from the system's set of pending signals, and returning that signal number in *sig*.

If no signal in *set* is pending at the time of the call, the thread is suspended until one or more of the signals in *set* become pending. The signals defined by *set* may be unblocked during the call to this routine, and will be blocked when the thread returns from the call unless some other thread is currently waiting for one of those signals.

If more than one thread is using this routine to wait for the same signal, only one of these threads will return from this routine with the signal number.

### Returned Value

If successful, `sigwait()` returns the signal number. If unsuccessful, it returns the value `-1` and sets `errno` to `EINVAL`, which indicates that the *set* argument contains an invalid or unsupported signal number.

### Example

#### CBC3BS26

```
/* CBC3BS26 */
#define _OPEN_THREADS

#include <stdio.h>
#include <errno.h>
#include <signal.h>
#include <pthread.h>
#include <unistd.h>

void          *threadfunc(void *parm)
{
    int          threadnum;
    int          *tnum;
    sigset_t     set;

    tnum = parm;
    threadnum = *tnum;

    printf("Thread %d executing\n", threadnum);
    sigemptyset(&set);
    if(sigaddset(&set, SIGUSR1) == -1) {
        perror("Sigaddset error");
    }
```

```

    pthread_exit((void *)1);
}

if(sigwait(&set) != SIGUSR1) {
    perror("Sigwait error");
    pthread_exit((void *)2);
}

pthread_exit((void *)0);
}

main() {
    int          status;
    int          threadparm = 1;
    pthread_t     threadid;
    int          thread_stat;

    status = pthread_create( &threadid, NULL,
                            threadfunc,
                            (void *)&threadparm);

    if ( status < 0) {
        perror("pthread_create failed");
        exit(1);
    }

    sleep(5);

    status = pthread_kill( threadid, SIGUSR1);
    if ( status < 0)
        perror("pthread_kill failed");

    status = pthread_join( threadid, (void *)&thread_stat);
    if ( status < 0)
        perror("pthread_join failed");

    exit(0);
}

```

## Related Information

- “signal.h” on page 41
- “bsd\_signal() — BSD Version of signal()” on page 135
- “sigpause() — Unblock a Signal and Wait for a Signal” on page 1336
- “pause() — Suspend a Process Pending a Signal” on page 899
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sigpending() — Examine Pending Signals” on page 1337
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351
- “sigignore() — Set Disposition to Ignore a Signal” on page 1323
- “sigset() — Change a Signal Action and/or a Thread” on page 1343

## sigwaitinfo() — Wait for Queued Signals

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 UNIX

### Format

```
#define _XOPEN_SOURCE 500
#define _XOPEN_REALTIME
#include <signal.h>
```

```
int sigwaitinfo(const sigset_t *set, siginfo_t *info);
```

### General Description

The `sigwaitinfo()` function selects a pending signal from the `sigset_t` object (signal set) pointed to by `set`, automatically clearing it from the system's set of pending signals, and returning that signal number. If there are multiple pending signals, the lowest numbered signal will be selected.

If no signal in the signal set is pending at the time of the call to `sigwaitinfo()`, the thread is suspended until one or more of the signals specified in the signal set become pending or until it is interrupted by an unblocked, caught signal. The signals defined in the `sigset_t` object (signal set) pointed to by `set` may be unblocked during the call to this routine and will be blocked when the thread returns from the call unless some other thread is currently waiting for one of those signals.

If more than one thread is using `sigwaitinfo()` to wait for the same signal, only one of these threads will return from this routine with the signal number, until a second signal of the same type is received.

The function `sigwaitinfo()` behaves the same as the `sigwait()` function if the `info` argument is `NULL`. If the `info` argument is not `NULL` then in addition to behaving the same as `sigwait()`, `sigwaitinfo()` places the selected signal number in the `si_signo` member, places the cause of the signal in the `si_code` member, and, if any value is queued to the selected signal, `sigwaitinfo()` will place it in the `si_value` member of `info`. However, if there is no value queued for the selected signal then the content of `si_value` is undefined.

### Returned Value

If successful, `sigwaitinfo()` returns the signal number.

If unsuccessful, `sigwaitinfo()` returns `-1` and returns the error value in `errno`. The following are the possible values of `errno`:

- |        |   |
|--------|---|
| EINTR  | The wait was interrupted by an unblocked, caught signal. No further waiting will occur for this call. <code>sigwaitinfo()</code> can be re-issued to begin waiting again. |
| EINVAL | <code>set</code> points to a <code>sigset_t</code> that contains a signal number that is either not valid or not supported.   |

**Related Information**

- “signal.h” on page 41
- “time.h” on page 51
- “pause() — Suspend a Process Pending a Signal” on page 899
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sigpending() — Examine Pending Signals” on page 1337
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351
- “sigwait() — Wait for an Asynchronous Signal” on page 1356

## sin() — Calculate Sine

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double sin(double x);
```

### General Description

Calculates the sine of  $x$ , with  $x$  expressed in radians.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

If successful, the function returns the calculated value, expressed as a double float. Otherwise, if the result underflows, it returns zero and sets the `errno` to `ERANGE`.

### Example

#### CBC3BS27

```
/* CBC3BS27
   This example computes y = sin( pi/2 )
*/
#include <math.h>
#include <stdio.h>

int main(void)
{
    double pi, x, y;

    pi = 3.1415926535;
    x = pi/2;
    y = sin(x);

    printf("sin( %lf ) = %lf\n", x, y);
}
```

### Output

```
sin( 1.570796 ) = 1.000000
```

### Related Information

- “math.h” on page 35
- “acos() — Calculate Arccosine” on page 85
- “acosh() — Hyperbolic Arccosine” on page 87
- “asin() — Calculate Arcsine” on page 108
- “asinh() — Hyperbolic Arcsine” on page 110
- “atan() - atan2() — Calculate Arctangent” on page 113
- “atanh() — Hyperbolic Arctangent” on page 115

- “cos() — Calculate Cosine” on page 229
- “cosh() — Calculate Hyperbolic Cosine” on page 231
- “sinh() — Calculate Hyperbolic Sine” on page 1362
- “tan() — Calculate Tangent” on page 1500
- “tanh() — Calculate Hyperbolic Tangent” on page 1502

sinh() — Calculate Hyperbolic Sine

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

Format

#include <math.h>

double sinh(double x);

General Description

Calculates the hyperbolic sine of x, with x expressed in radians.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

Returned Value

If successful, the function returns the calculated value. Otherwise, if the result is too large, sinh() sets errno to ERANGE and returns the value HUGE\_VAL (positive or negative, depending on the value of x). If the value underflows, it returns zero and sets errno to ERANGE.

Example

CBC3BS28

```
/* CBC3BS28
   This example computes y = sinh ( pi / 2 )
*/
#include <math.h>
#include <stdio.h>

int main(void)
{
    double pi, x, y;

    pi = 3.1415926535;
    x = pi/2;
    y = sinh(x);

    printf("sinh( %lf ) = %lf\n", x, y);
}
```

Output

sinh( 1.570796 ) = 2.301299



**Related Information**

- “math.h” on page 35
- “acos() — Calculate Arccosine” on page 85
- “acosh() — Hyperbolic Arccosine” on page 87
- “asin() — Calculate Arcsine” on page 108
- “asinh() — Hyperbolic Arcsine” on page 110
- “atan() - atan2() — Calculate Arctangent” on page 113
- “atanh() — Hyperbolic Arctangent” on page 115
- “cos() — Calculate Cosine” on page 229
- “cosh() — Calculate Hyperbolic Cosine” on page 231
- “sin() — Calculate Sine” on page 1360
- “tan() — Calculate Tangent” on page 1500
- “tanh() — Calculate Hyperbolic Tangent” on page 1502

## sleep() — Suspend Execution of a Thread

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	POSIX(ON) MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
unsigned int sleep(unsigned int seconds);
```

### General Description

Suspends thread execution for a specified number of *seconds*. Because of processor delays, the thread can sleep slightly longer than this specified time. An unblocked signal received during this time (for which the action is to invoke a signal-handler function or to end the thread) “wakes up” the thread prematurely. When that function returns, sleep() returns immediately even if there is sleep time remaining.

This function is supported only in a POSIX program. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information.

### Returned Value

Returns zero if the thread slept for the full specified time. If the thread awoke prematurely because of a signal whose action is to invoke a signal-handling function or to end the thread, sleep() returns the number of seconds remaining in its sleep time (that is, the value of *seconds* minus the actual number of seconds that the thread was suspended).

sleep() always succeeds, so there is no failure return. An abend is generated when any failures are encountered that prevent this function from completing successfully.

There are no documented errno values for this function.

### Example CBC3BS29

```
/* CBC3BS29
   This example suspends execution for a specified time.
*/
#define _POSIX_SOURCE
#include <stdio.h>
#include <time.h>
#include <unistd.h>

main() {
    unsigned int ret;
    time_t t;
    time(&t);
    printf("starting sleep at %s", ctime(&t));
    ret = sleep(10);
```

```

time(&t);
printf("naptime over at %s", ctime(&t));
printf("sleep() returned %d\n", ret);
}

```

## Output

```

starting sleep at Fri Jun 16 07:44:47 1995
naptime over at Fri Jun 16 07:44:58 1995
sleep() returned 0

```

## Related Information

- “signal.h” on page 41
- “unistd.h” on page 53
- “longjmp() — Restore Stack Environment” on page 768
- “alarm() — Set an Alarm” on page 102
- “bsd\_signal() — BSD Version of signal()” on page 135
- “kill() — Send a Signal to a Process” on page 728
- “killpg() — Send a Signal to a Process Group” on page 731
- “\_longjmp() — Non-Local Goto” on page 771
- “pause() — Suspend a Process Pending a Signal” on page 899
- “pthread\_kill() — Send a Signal to a Thread” on page 984
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sigignore() — Set Disposition to Ignore a Signal” on page 1323
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “siglongjmp() — Restore the Stack Environment and Signal Mask” on page 1327
- “signal() — Handle Interrupts” on page 1330
- “sigset() — Change a Signal Action and/or a Thread” on page 1343
- “sigsuspend() — Change Mask and Suspend the Thread” on page 1351

## \_\_smf\_record() — Record an SMF Record

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
int __smf_record(int smf_record_type,  
                 int smf_record_subtype,  
                 int smf_record_length,  
                 char *smf_record);
```

### General Description

The \_\_smf\_record() function writes an SMF record pointed to by smf\_record of length smf\_record\_length for SMF record type smf\_record\_type and subtype smf\_record\_subtype to the SMF data set.

The service can also be used to determine if a particular type or subtype of SMF record is being recorded to avoid the overhead of data collection if the SMF record is not going to be recorded. See *OS/390 System Management Facilities (SMF)* for more information on SMF record types and layout.

The caller of this service must be permitted to the BPX.SMF facility class profile. For information on creating and using this profile and the restrictions on its use, refer to *OS/390 UNIX System Services Planning*.

### Returned Value

If successful, \_\_smf\_record() returns 0. Otherwise, \_\_smf\_record() returns -1 and sets errno to one of the following:

- EINVAL      The value specified on the length operand was incorrect.
- ENOMEM     Not enough storage.
- EPERM      The calling process is not permitted to the BPX.SMF facility class.
- EMVSERR    The SMF service returned a non-zero return code. Use \_errno2() to determine why the error occurred. The following reason codes can accompany the return code: JRSMFNotAccepting, JRSMFError, JRBadAddress, or JRInternalError.

### Related Information

None.

## sock\_debug() — Provide OE Syscall Tracing Facility

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R3

### Format

```
#define _OPEN_SYS_SOCKET_EXT
#include <sys/socket.h>
void sock_debug(int onoff);
```

### General Description

The `sock_debug()` call provides the UNIT tracing facility. The *onoff* parameter can have a value of 0 or nonzero. If *onoff* is equal to 0 (the default), no UNIT tracing is done. If *onoff* is a value other than zero, the system traces for UNIT calls and interrupts.

As an alternative to calling `sock_debug()` with *onoff* set to a nonzero value, you can include the statement `SOCKDEBUG` in the file `/etc/resolv.conf` or data set `tcip.TCPIP.DATA`. When the process is started, all the programs will begin the UNIT trace and report the progress to the `stderr` data set.

### Parameter Description

*onoff*            A parameter that can be set to zero or nonzero.

### Returned Value

There is no return value.

### Related Information

- “`sys/socket.h`” on page 48

## sock\_debug\_bulk\_perf0() — Produce a Report When a Socket Configured

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R3

### Format

```
#define _OPEN_SYS_SOCKET_EXT
#include <sys/socket.h>
void sock_debug_bulk_perf0(int onoff);
```

### General Description

When a socket is configured in bulk mode and `sock_debug_bulk_perf0()` is called, data is collected and used to produce a report. The report is written to `stderr` when the bulk mode socket is closed.

### Parameter Description

*onoff*            A boolean value either true to activate or false to deactivate the report.

The following is an example of the report written to `stderr` for socket descriptor 3 when the socket was closed, and the `sock_debug_bulk_perf0()` function was issued to start collecting data.

```
Bulkmode performance for socket 3:
Doing TESTSTOR (ie. testing addressability of buffers, etc.)
Received 14601460 bytes,
10001 datagrams, 846 UNIT's 11.8
datagrams/UNIT.
```

In this example, 3 is the socket descriptor for the socket running in bulk mode. Doing TESTSTOR indicates that the library was checking for addressing errors on socket calls, and 14 601 460 bytes of data were received in 10001 datagrams. 846 calls were done to read the datagrams for the socket with an average of 11.8 datagrams for each UNIT or OE Syscall.

As an alternative to calling `sock_debug_bulk_perf0()` with *onoff* set to a nonzero value, you can include the statement `SOCKDEBUGBULKPERF0` in the file `/etc/resolv.conf` or data set `tcpip.TCPIP.DATA`. When the process is started, all the programs using bulk mode sockets for this process will produce a report.

### Returned Value

There is no return value.

### Related Information

- “`sys/socket.h`” on page 48

## sock\_do\_bulkmode() — Use Bulk Mode for Messages Read by the Socket

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R3

### Format

```
#define _OPEN_SYS_SOCKET_EXT
#include <sys/socket.h>
void sock_do_bulkmode(int onoff);
```

### General Description

The `sock_do_bulkmode()` call uses bulk mode for messages read by the socket program.

#### Parameter Description

*onoff*            A parameter that can be set to zero or nonzero.

If *onoff* is set to a nonzero value when a `SOCK_DGRAM` socket is created, the socket is configured to use bulk mode for messages read by the socket program. Use of bulk mode can improve program performance. Performance improvement depends on the system load and the arrival pattern of the datagram messages at the socket. As system load increases, the reduction in CPU use because of bulk mode should also increase. When datagrams for the socket are processed, there should be an even greater reduction in CPU usage. With bulk mode set on, if a `setibmsockopt()` is not used to specify the receive and/or send queue size, a default of 32768 will be used for the receive queue. The default value for the send queue is 0. The `setibmsockopt()` function must be used to specify a value for the send queue, thus turning on bulk mode for sends.

If *onoff* is set to zero when a socket is created, the socket does not use bulk mode, unless the socket program is using `setibmsockopt()` to specify bulk mode for the individual socket.

As an alternative to calling `sock_do_bulkmode()` with *onoff* set to a nonzero value, you can include the statement `SOCKBULKMODE` in the file `/etc/resolv.conf` or data set `tcpip.TCPIP.DATA`. When the process is started, all the programs using datagram sockets will begin running in bulk mode for reads on the socket.

### Returned Value

There is no return value.

### Related Information

- “`sys/socket.h`” on page 48

## sock\_do\_teststor — Check for Attempt to Access Storage Outside

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	OS/390 R3

### Format

```
#define _OPEN_SYS_SOCKET_EXT
#include <sys/socket.h>
void sock_do_teststor(int onoff);
```

### General Description

The `sock_do_teststor` call is used to check for calls that attempt to access storage outside the caller's address space.

### Parameter Description

*onoff*            A parameter that can be set to zero or nonzero.

If *onoff* is set to a nonzero value, for either inbound or outbound sockets, both the address of the message buffer and the message buffer are checked for addressability for each bulk mode socket call. The EFAULT error condition is set if there is an addressing problem. If *onoff* is set to 0, address checking is not done by the socket library program. If an error occurs when *onoff* is 0, normal run-time error handling reports the exception condition.

To improve response time, you can disable this checking when your program has been tested.

As an alternative to calling `sock_do_teststor`, with *onoff* set to a nonzero value, you can include the statement `SOCKETTESTSTOR` in the file `/etc/resolv.conf` or data set `tcpip.TCPIP.DATA`. When the process is started, all the programs using bulk mode sockets for this process will validate the storage for the caller's parameters.

### Returned Value

There is no return value.

### Related Information

- “`sys/socket.h`” on page 48



socket() — Create a Socket

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

Format  
X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/socket.h>

int socket(int *domain, int type, int protocol);
```

General Description

The `socket()` call creates an endpoint for communication and returns a socket descriptor representing the endpoint. Different types of sockets provide different communication services.

Parameter	Description
<i>domain</i>	The address domain requested, either AF_INET, AF_UNIX, or AF_RAW.
<i>type</i>	The type of socket created, either SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW.
<i>protocol</i>	The protocol requested. Some possible values are 0, IPPROTO_UDP, or IPPROTO_TCP.

The *domain* parameter specifies a communication domain within which communication is to take place. This parameter selects the address family (format of addresses within a domain) that is used. The families supported are AF\_INET, which is the Internet domain, and AF\_UNIX, which is the local socket domain. These constants are defined in the **sys/socket.h** include file.

The *type* parameter specifies the type of socket created. The type is analogous with the semantics of the communication requested. These socket type constants are defined in the **sys/socket.h** include file. The types supported are:

Socket Type Description

**SOCK\_DGRAM**  
Provides datagrams, which are connectionless messages of a fixed maximum length whose reliability is not guaranteed. Datagrams can be corrupted, received out of order, lost, or delivered multiple times. This type is supported in both the AF\_INET and AF\_UNIX domains.

**SOCK\_STREAM**

Provides sequenced, two-way byte streams that are reliable and connection-oriented. They support a mechanism for out-of-band data. This type is supported in both the AF\_INET and AF\_UNIX domains.

**SOCK\_RAW**

Provides the interface to internal protocols (such as IP and ICMP). This type is supported in only the AF\_INET domain.

**Understanding the socket() Parameters**

The *protocol* parameter specifies a particular protocol to be used with the socket. In most cases, a single protocol exists to support a particular type of socket in a particular address family. If the *protocol* parameter is set to 0, the system selects the default protocol number for the domain and socket type requested. Protocol numbers are found in the *tcpip.ETC.PROTO* data set. Alternatively, the `getprotobyne()` call can be used to get the protocol number for a protocol with a known name.

**Note:** The *protocol* field *must* be set to 0, if the *domain* parameter is set to AF\_UNIX.

SOCK\_STREAM sockets model duplex-byte streams. They provide reliable, flow-controlled connections between peer application programs. Stream sockets are either active or passive. Active sockets are used by clients who start connection requests with `connect()`. By default, `socket()` creates active sockets. Passive sockets are used by servers to accept connection requests with the `connect()` call. You can transform an active socket into a passive socket by binding a name to the socket with the `bind()` call and by indicating a willingness to accept connections with the `listen()` call. After a socket is passive, it cannot be used to start connection requests.

In the AF\_INET domain, the `bind()` call applied to a stream socket lets the application program specify the networks from which it is willing to accept connection requests. The application program can fully specify the network interface by setting the *Internet address* field in the **address** structure to the Internet address of a network interface. Alternatively, the application program can use a *wildcard* to specify that it wants to receive connection requests from any network. This is done by setting the *Internet address* field in the **address** structure to the constant INADDR\_ANY, as defined in **netinet/in.h**.

After a connection has been established between stream sockets, any of the data transfer calls can be used: `read()`, `readv()`, `recv()`, `recvfrom()`, `recvmsg()`, `send()`, `sendmsg()`, `sendto()`, `write()`, and `writen()`. Usually, the `read()`-`write()` or `send()`-`recv()` pairs are used for sending data on stream sockets. If out-of-band data is to be exchanged, the `send()`-`recv()` pair is normally used.

SOCK\_DGRAM sockets model datagrams. They provide connectionless message exchange without guarantees of reliability. Messages sent have a maximum size. Datagram sockets are supported in the AF\_UNIX domain.

There is no active or passive analogy to stream sockets with datagram sockets. Servers must still call `bind()` to name a socket and to specify from which network interfaces it wishes to receive packets. Wildcard addressing, as described for stream sockets, applies for datagram sockets also. Because datagram sockets are

connectionless, the `listen()` call has no meaning for them and must not be used with them.

After an application program has received a datagram socket, it can exchange datagrams using the `sendto()` and `recvfrom()`, or `sendmsg()` and `recvmsg()`, calls. If the application program goes one step further by calling `connect()` and fully specifying the name of the peer with which all messages will be exchanged, then the other data transfer calls `read()`, `write()`, `readv()`, `writv()`, `send()`, and `recv()` can also be used. For more information on placing a socket into the connected state, see “`connect()` — Connect a Socket” on page 214.

Datagram sockets allow messages to be broadcast to multiple recipients. Setting the destination address to be a broadcast address is network-interface-dependent (it depends on the class of address and whether *subnets*—logical networks divided into smaller physical networks to simplify routing—are used). The constant `INADDR_BROADCAST`, defined in **netinet/in.h**, can be used to broadcast to the primary network if the primary network configured supports broadcast.

Outgoing packets have an IP header prefixed to them. IP options can be set and inspected using the `setsockopt()` and `getsockopt()` calls, respectively. Incoming packets are received with the IP header and options intact.

Sockets are deallocated with the `close()` call.

**Note:** For `AF_UNIX`, when closing sockets that were bound, you should also use `unlink()` to delete the file created at `bind()` time.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

A nonnegative socket descriptor indicates success. The value `-1` indicates an error. The value of the error code indicates the specific error.

#### Error Code Description

<code>EACCES</code>	Permission to create a socket of the specified type or protocol is denied.
<code>EAFNOSUPPORT</code>	The address family is not supported (it is not <code>AF_UNIX</code> or <code>AF_INET</code> ).
<code>EAGAIN</code>	Resource temporarily unavailable.
<code>EINVAL</code>	The request is invalid or not supported.
<code>EIO</code>	There has been a network or transport failure.
<code>ENOENT</code>	There was no <code>NETWORK</code> statement in the <code>parmlib</code> member to match the specified domain.
<code>ENOBUFS</code>	Insufficient system resources are available to complete the call.
<code>EPROTONOSUPPORT</code>	The protocol is not supported in this domain or this protocol is not supported for this socket type.

## EPROTOTYPE

The socket type is not supported by the protocol.

**Example**

The following are examples of the `socket()` call.

```
int s;
char *name;
int socket(int domain, int type, int protocol);
:
/* Get stream socket in Internet domain with default protocol */
s = socket(AF_INET, SOCK_STREAM, 0);
:
/* Get stream socket in local socket domain with default protocol */
s = socket(AF_UNIX, SOCK_STREAM, 0);
```

**Related Information**

- “`accept()` — Accept a New Connection on a Socket” on page 75
- “`bind()` — Bind a Name to a Socket” on page 128
- “`close()` — Close a File” on page 191
- “`connect()` — Connect a Socket” on page 214
- “`fcntl()` — Control Open File Descriptors” on page 350
- “`getprotobyname()` — Get a Protocol Entry by Name” on page 580
- “`getsockname()` — Get the Name of a Socket” on page 604
- “`getsockopt()` — Get the Options Associated with a Socket” on page 606
- “`ioctl()` — Control Device” on page 672
- “`read()` — Read From a File or Socket” on page 1080
- “`readv()` — Read Data on a File or Socket and Store in a Set of Buffers” on page 1093
- “`recv()` — Receive Data on a Socket” on page 1103
- “`recvfrom()` — Receive Messages on a Socket” on page 1106
- “`recvmsg()` — Receive Messages on a Socket and Store in an Array of Message Headers” on page 1110
- “`select()` — Monitor Activity on Files/Sockets and Message Queues” on page 1159
- “`selectex()` — Monitor Activity on Files/Sockets and Message Queues” on page 1166
- “`send()` — Send Data on a Socket” on page 1178
- “`sendmsg()` — Send Messages on a Socket” on page 1185
- “`shutdown()` — Shut Down All or Part of a Duplex Connection” on page 1293
- “`write()` — Write Data on a File or Socket” on page 1780
- “`writew()` — Write Data on a File or Socket Socket from an Array” on page 1785

## socketpair() — Create a Pair of Sockets

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/socket.h>

int socketpair(int *domain, int type, int protocol, int
socket_vector[2]);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/socket.h>

int socketpair(int *domain, int type, int protocol, int
sv[2]);
```

### General Description

The `socketpair()` call acquires a pair of sockets of the type specified that are unnamed and connected in the specified domain and using the specified protocol. For socket pairs in the `AF_UNIX` domain, the protocol *must* be 0.

#### Parameter Description

<i>domain</i>	The domain in which to open the socket. Although socket pairs can be obtained for <code>AF_INET</code> domain sockets, it is recommended that <code>AF_UNIX</code> domain sockets be used for socket pairs.
<i>type</i>	The type of socket created, either <code>SOCK_STREAM</code> , or <code>SOCK_DGRAM</code> .
<i>protocol</i>	The protocol requested <i>must</i> be 0.
<i>sv</i>	The descriptors used to refer to the obtained sockets.

#### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

A nonnegative socket descriptor indicates success. The value `-1` indicates an error. The value of the error code indicates the specific error.

#### Error Code Description

<code>EACCES</code>	Permission to create a socket of the specified type or protocol is denied.
<code>EFAULT</code>	<i>sv</i> is not in the writable part of the user's address space.
<code>EINVAL</code>	The request is invalid or not supported.

EMFILE	Too many files are open for this process.
ENFILE	Too many files are open in the system.
ENOBUFS	Insufficient system resources are available to complete the call.
EOPNOSUPPORT	The protocol does not allow for the creation of socket pairs.
EPROTONOSUPPORT	The protocol is not supported in this domain or this protocol is not supported for this socket type.
EPROTOTYPE	The socket type is not supported by the protocol.

### Example

The following are examples of the `socketpair()` call.

```
#include <types.h>
#include <sys/socket.h>

int sv[2];
:
/* Get stream socket in UNIX domain with default protocol */
if (socketpair(AF_UNIX, SOCK_STREAM, 0, sv) < 0)
printf ("Error occurred while trying to get a socket pair.\n");
else
:
```

### Related Information

- “`socket()` — Create a Socket” on page 1371

## spawn() - spawnp() — Spawn a New Process

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.4b OS/390 UNIX	both	MVS 5.2.2

### Format

```
#define _POSIX_SOURCE
#include <spawn.h>
pid_t spawn(const char *path,
            const int fd_count,
            const int fd_map[],
            const struct inheritance *inherit,
            const char *argv[],
            const char *envp[]);

pid_t spawnp(const char *file,
            const int fd_count,
            const int fd_map[],
            const struct inheritance *inherit,
            const char *argv[],
            const char *envp[]);
```

### General Description

The `spawn()` and `spawnp()` functions create a new process from the specified process image. `spawn()` and `spawnp()` create the new process image from a regular executable file called the new process image file.

To execute a C program as a result of this call, enter the function call as follows:

```
int main (int argc, char *argv[]);
```

Where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the following variable:

```
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings. The *argv* and *environ* arrays are each terminated by a NULL pointer. The NULL pointer terminating the *argv* array is not counted in *argc*.

Supported parameters are:

#### Parameter Description

<i>path</i>	Pathname used by <code>spawn()</code> that identifies the new process image file to execute.
<i>file</i>	Used by <code>spawnp()</code> to construct pathname that identifies the new process image file. If the file parameter contains a slash character, <code>spawnp()</code> uses the file parameter as a pathname for the new process image file. Otherwise, <code>spawnp()</code> obtains the path prefix for this file by a search of the directories passed as the environment variable <code>PATH</code> .

*fd\_count* Specifies the number of file descriptors the child process inherits. It may take values from zero to {OPEN\_MAX}. Except for those file descriptors designated by SPAWN\_FDCLOSED, each of the child's file descriptors, *x*, in the range zero to *fd\_count*-1 inherits descriptor *fd\_map[x]* from the parent process.

The files from *fd\_count* through {OPEN\_MAX} are closed in the child process, as are any elements of *fd\_map* designated as SPAWN\_FDCLOSED.

*fd\_map* If the *fd\_map* parameter is NULL, *fd\_count* and *fd\_map* are ignored. All file descriptors except those with the FD\_CLOEXEC or FD\_CLOFORK attribute are inherited without reordering. File descriptors with the FD\_CLOEXEC or FD\_CLOFORK attribute are closed under simple inheritance.

For those file descriptors that remain open, all other attributes of the associated file descriptor object and open file description remain unchanged by this operation.

Directory streams open in the calling process are closed in the new process image.

If an element of *fd\_map* refers to an invalid file descriptor, then the (EBADF) spawn() or spawnp() posts the error status.

The FD\_CLOEXEC and FD\_CLOFORK file descriptor attributes are never inherited.

The FD\_CLOEXEC and FD\_CLOFORK file descriptor attributes have no effect on inheritance when the *fd\_map* parameter is not NULL.

*inherit* The name of a data area that contains the inheritance structure.

The 'struct inheritance' is defined as follows:

```
struct inheritance {
    short    flags;           --Flags
    pid_t    pgroup;         --Process group
    sigset_t sigmask;        --Signal mask
    sigset_t sigdefault;     --Signals set to SIG_DFL
    int      ctltyfd;        --Cntl tty FD for tcsetpgrp()
}
```

The inherit.flags effect spawn() and spawnp() as follows:

#### SPAWN\_SETGROUP

If the SPAWN\_SETGROUP flag is set in inherit.flags, then the child's process group is as specified in inherit.pgroup.

If the SPAWN\_SETGROUP flag is set in inherit.flags and inherit.pgroup is set to SPAWN\_NEWPGROUP, then the child is in a new process group with a process group ID equal to its process ID.

If the SPAWN\_SETGROUP flag is not set in inherit.flags, the new child process inherits the parent's process group ID.

#### SPAWN\_SETSIGMASK

If the SPAWN\_SETSIGMASK flag is set in inherit.flags, the child process initially has the signal mask specified in inherit.sigmask.



## SPAWN\_SETSIGDEF

If the SPAWN\_SETSIGDEF flag is set in `inherit.flags`, the signals specified in `inherit.sigdefault` are set to their default actions in the child process. Signals set to the default action in the parent process, are set to the default action in the new process.

Signals set to be caught by the calling process are set to the default action in the child process.

Signals set to be ignored by the calling process are set to be ignored by the new process, unless otherwise specified by the SPAWN\_SETSIGDEF flag being set in `inherit.flags` and the signal being indicated in `inherit.sigdefault`.

## SPAWN\_SETTCPGRP

If the SPAWN\_SETTCPGRP flag is set in `inherit.flags`, the file descriptor specified in `inherit.clttyfd` is used to set the controlling terminal file descriptor (`tcsetpgrp()`) for the child's foreground process group. The child's foreground process group is inherited from the parent, unless the SPAWN\_SETGROUP flag in `inherit.flags` is set, indicating that the value specified in `inherit.pgroup` is to be used to determine the child's process group.

*argv* The value in the first element of *argv* should point to a filename that is associated with the process being started by the `spawn()` or `spawnp()` operation.

The number of bytes available for the new process' combined argument and environment lists is `{ARG_MAX}`.

*envp* The value *envp* contains the list of environmental variables that is to be passed to the specified program.

If the set-user-ID mode bit of the new process image file is set, the effective user ID of the new process image is set to the owner id of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image is set to the group id of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image are saved (as the saved set-user-ID and the saved set-group-ID) for use by the `setuid()` function.

The new process image inherits the following attributes from the calling process image:

- Process group ID (unless the **SPAWN\_SETGROUP** flag is set in `inherit.flags`)
- Session membership
- Real user ID
- Real group ID
- Supplementary group IDs
- Priority
- Current working directory
- Root directory

- File creation mask
- Signal mask (unless the **SPAWN\_SETSIGMASK** flag is set in `inherit.flags`)
- Signal actions specified as default (SIG\_DFL)
- Signal actions specified as ignore (SIG\_IGN) (except as modified by `inherit.sigdefault` and the **SPAWN\_SETSIGDEF** flag set in `inherit.flags`)

The following are differences between the parent process and the child process:

- Signals set to be caught by the calling process are set to the default action (SIG\_DFL).
- The process and system utilization times for the child are set to zero.
- Any file locks previously set by the parent are not inherited by the child.
- The child process has no alarms set.
- The child process has no interval timers set.
- The child has no pending signals.
- Memory mappings established by the parent are not inherited by the child.

If the process image was read from a writable file system, then upon successful completion, the `spawn()` or `spawnp()` function mark for update the `st_atime` field of the new process image file.

If the `spawn()` or `spawnp()` function is successful, the new child process image file is opened, with all the effects of the `open()` function.

### Special Behavior for OS/390 UNIX Services

Aspects of spawn processing are controlled by environment variables. The environment variables that affect spawn processing are the ones that are passed into the `spawn` syscall, and not the environment variables of the calling process. The environment variables of the calling process do not affect spawn processing, unless they are the same as those that are passed in `envp`.

For more information on the use of environment variables, see *OS/390 UNIX System Services Programming: Assembler Callable Services Reference*.

Security information from the parent's address space is propagated to the child's address space, unless the `_BPX_USERID` environment variable specifies otherwise. As a result, the child has a security environment equivalent to that of the parent.

The TASKLIB, STEPLIB, or JOBLIB DD data set allocations that are active for the current task are propagated to the child's address space, unless the STEPLIB environment variable specifies otherwise. This causes the child address space to have the same exact MVS program search order as the calling parent task.

The accounting information of the parent's address space is propagated to the child's address space. See *OS/390 UNIX System Services Planning*.

The jobname of the parent is propagated to the child and appended with a numeric value in the range of 1-9 if the jobname is 7 characters or less. If the jobname is 8 characters, then it is propagated as-is. When a jobname is appended with a numeric value, the count wraps back to 1 when it exceeds 9.

If the calling parent task is in a WLM enclave, the child is joined to the same WLM enclave. This allows WLM to manage the parent and child as one "business unit of work" entity for system accounting and management purposes.

To allow the caller to control whether the spawned child process runs in a separate address space from the parent address space or in the same address space, the spawn service allows for the specification of the `_BPX_SHAREAS` environment variable. The following are the accepted values for the `_BPX_SHAREAS` environment variable, and the actions taken for each value:

1. `_BPX_SHAREAS=YES` - Indicates that the child process that is to be created is to run in the same address space as the parent. In the following circumstances, the `_BPX_SHAREAS=YES` value cannot be honored, and the child process is created in its own address space:

- If the program to be run is a set-user-ID or set-group-ID program that would cause the effective user-ID or group-ID of the child process to be different from that of the parent process.
- If the program to be run is an APF-authorized HFS or MVS program and the caller is not running APF authorized.
- If the program to be run is an unauthorized HFS or MVS program and the caller is running APF authorized.
- If the specified filename represents an external link or a sticky bit file. If, however, the program that is to be run is a shell script and `_BPX_SPAWN_SCRIPT=YES` is set, the process runs in the same address space.
- If the parent's address space lacks the necessary resources to create another process within the address space.

Note that only one local spawned process per TSO address space is supported at a given time. This is done to reduce conflict among multiple shells running in the same address space.

2. `_BPX_SHAREAS=MUST` - Indicates that the child process that is to be created must run in the same address space as the parent, or the spawn request will fail. In the following circumstances, the `_BPX_SHAREAS=MUST` value cannot be honored, and the spawn invocation fails:

- If the program to be run is a set-user-ID or set-group-ID program that would cause the effective user ID or group ID of the child process to be different from that of the parent process.
- If the program to be run is an APF-authorized HFS or MVS program and the caller is not running APF authorized.
- If the program to be run is an unauthorized HFS or MVS program and the caller is running APF authorized.
- If the parent's address space lacks the necessary resources to create another process within the address space.

3. `_BPX_SHAREAS=REUSE` - Indicates that the child process to be created is to run in the same address space as the parent; also, that it will be created as a medium-weight process. Specifying `REUSE` allows the caller to indicate that it wants to reuse the existing process structure for locally spawned processes.

The same rules that apply to the creation of a local spawn process apply to the specification of a local spawn medium-weight process. In addition, in the following circumstances, the `_BPX_SHAREAS=REUSE` value cannot be honored, and the child process will be created as a non-medium weight local spawn process:

- If `PTRACE` is active for the process.
- If the program to execute is a REXX exec.

For performance reasons, the `STEPLIB` that is specified for each medium-weight process that is created for the address space should be the same.

4. `_BPX_SHAREAS=NO` - Indicates that the child process that is to be created is to run in a separate address space from the parent's address space. This is the default behavior for the spawn service if the `_BPX_SHAREAS` environment variable is not specified, or if it contains an unsupported value.

If you specify the `_BPX_USERID` environment variable, then `spawn()` creates the new address space and image with the specified `userid`'s identity. The invoker of `spawn()` must be authorized to change MVS identity. The resulting `spawn()` image will emerge as if a program had done a `fork()`, `setgid()`, `initgroups()`, `setuid()`, and `exec`.

The value of `_BPX_USERID` can be any 1 to 8 character XPG4 compliant username. If you specify both `_BPX_USERID` and `_BPX_SHAREAS`, then `spawn()` ignores `_BPX_SHAREAS`, and creates a new address space with the new identity.

If the caller of the `spawn()` function is an OS/390 UNIX shell application (i.e `/bin/sh`), then the setting of the `_BPX_SPAWN_SCRIPT=` environment variable to `YES` is recommended. The setting of this variable to `YES` provides a more efficient mechanism to invoke shell scripts.

To support the creation and propagation of a `STEPLIB` environment to the new process image, `spawn()` - `spawnp()` allows for the specification of a `STEPLIB` environment variable. The following are the accepted values for the `STEPLIB` environment variable and the actions taken for each:

- `STEPLIB=NONE`. No `Steplib` DD is to be created for the new process image.
- `STEPLIB=CURRENT`. The `TASKLIB`, `STEPLIB` or `JOBLIB` DD data set allocations that are active for the calling task at the time of the call to `spawn()` - `spawnp()` are propagated to the new process image, if found to be cataloged. Uncataloged data sets are not propagated to the new process image.
- `STEPLIB=Dsn1:Dsn2;...DsnN`. The specified data sets, `Dsn1:Dsn2:...DsnN`, are built into a `STEPLIB` DD in the new process image.

**Note:** The actual name of the DD is not `STEPLIB`, but is a system generated name that has the same effect as a `STEPLIB` DD.

The data sets are concatenated in the order specified. The specified data sets must follow standard MVS data set naming conventions. Data sets found to be in violation of this standard are ignored. If the data sets do follow the standard, but:

- The caller does not have the proper security access to a data set.
- A data set is uncataloged or is not in load library format.

then the data set is ignored. Because the data sets in error are ignored, the executable file may run without the proper STEPLIB environment. If a data set is in error due to improper security access a X'913' abend is generated. The dump for this abend can be suppressed by your installation.

If the STEPLIB environment variable is not specified, spawn() - spawnp() default behavior is the same as if STEPLIB=CURRENT were specified.

If the program to be invoked is a set-user-ID or set-group-ID file and the user-ID or group-ID of the file is different from that of the current process image, then the data sets to be built into the STEPLIB environment for the new process image must be found in the system sanction list for set-user-id and set-group-id programs. Only those data sets that are found in the sanction list are built into the STEPLIB environment for the new process image. For detailed information regarding the sanction list, and for information on STEPLIB performance considerations, see *OS/390 UNIX System Services Planning*.

#### Notes:

1. A prior loaded copy of an HFS program in the same address space is reused under the same circumstances that apply to the reuse of a prior loaded MVS unauthorized program from an unauthorized library by the MVS XCTL service with the following exceptions:
  - If the calling process is in Ptrace debug mode, a prior loaded copy is not reused.
  - If the calling process is not in Ptrace debug mode, but the only prior loaded usable copy found of the HFS program is in storage modifiable by the caller, the prior copy is not reused.
2. If the specified file name represents an external link or a sticky bit file, the program is loaded from the caller's MVS load library search order. For an external link, the external name is only used if the name is eight characters or less, otherwise the caller receives an error from the loadhfs service. For a sticky bit program, the file name is used if it is eight characters or less. Otherwise, the program is loaded from the HFS.
3. If the calling task is in a WLM enclave, the resulting task in the new process image is joined to the same WLM enclave. This allows WLM to manage the old and new process images as one "business unit of work" entity for system accounting and management purposes.

#### Returned Value

If successful, spawn() and spawnp() return the value of the process ID of the child process to the parent process.

If unsuccessful, spawn() and spawnp() return -1 no child process is created, and they return an error value in errno. The following are the possible values of errno:

EACCES	Search permission is denied for a directory in the path of the new process image file or the new process image file denies execution permission, or the new process image file is not a regular file and the implementation does not support execution of files of its type.
E2BIG	The number of bytes used by the argument and environment list of the new process image is greater than the system imposed limit of {ARG_MAX} bytes.

ENOENT	One or more components of the pathname of the new process image file do not exist or the <i>path</i> or <i>file</i> argument is empty.
ELOOP	A loop exists in symbolic links encountered during resolution <i>file</i> argument. This error is issued if more than 8 symbolic links are detected in the resolution of Filename.
ENOTDIR	A component of the path prefix of the new process image file is not a directory.
ENOEXEC	The new process image file has the appropriate access permission but is not in the proper format.

**Note:**

Reason codes further qualify the errno. For most of the reason codes, see *OS/390 UNIX System Services Messages and Codes*.

For ENOEXEC, the reason codes are:

Reason Code	Explanation
X'xxx0C27'	The target HFS file is not in the correct format to be an executable file.
X'xxx0C31'	The target HFS file is built at a level that is higher than that supported by the running system.

**ENAMETOOLONG**

The length of the *path* or *file* arguments, or an element of the environment variable PATH prefixed to *file* exceeds {PATH\_MAX}, or a pathname component is longer than {NAME\_MAX} and {\_POSIX\_NO\_TRUNC} is in effect for that file.

EAGAIN	The system lacked the necessary resources to create another process or the system-imposed limit on the total number of processes or UIDs under execution by a single user would be exceeded.
ENOMEM	The new process requires more memory than is permitted by the hardware or the operating system.
EFAULT	The system detected an invalid address in attempting to use a parameter of the call.

**EMVSSAF2ERR**

The executable file is a set-user-ID or set-group-ID file and the file owner's UID or GID is not defined to the Security Access Facility (SAF), or \_BPX\_USERID was specified and the specified username was not defined to SAF with an OS/390 UNIX segment.

EBADF	An entry in the fd_map array refers to an invalid file descriptor or the controlling terminal file descriptor specified in the inherit.ctlttyfd is not valid.
ENOTTY	The tcsetpgrp() failed for the specified controlling terminal file descriptor in inherit.ctlttyfd. The failure occurred because the calling process does not have a controlling terminal, or the specified file descriptor is not associated with the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.

EPERM	<p>The spawn failed for one of the following reasons:</p> <ul style="list-style-type: none"> <li>• The spawned process is not a process group leader.</li> <li>• The <code>_BPX_USERID</code> environment variable was specified, and the invoker does not have appropriate privileges to change the MVS identity.</li> <li>• The invoker does not have the appropriate privileges to change one or more of the attributes specified in the inheritance structure (BPXYINHE).</li> </ul> <p>The following reason codes can accompany the return code: JROK, JRNoChangeIdentity, JRInheUserid, JRInheRegion, JRInheCPUTime, JRInheUmask, and JRInheCWD.</p>
ESRCH	<p>The process group ID specified in <code>inherit.pgroup</code> is not that of a process group in the calling process's session.</p>
EINVAL	<p>One or more of the following conditions were detected:</p> <ul style="list-style-type: none"> <li>• The username that was specified on the <code>_BPX_USERID</code> environment variable has an incorrect length.</li> <li>• An attribute that was specified in the inheritance structure (BPXYINHE) is not valid or contains an unsupported value.</li> <li>• The version number that was specified for the inheritance structure (BPXYINHE) is not valid.</li> <li>• The inheritance structure length that was specified by the <code>Inherit_area_len</code> parameter or within the inheritance structure does not contain a length that is appropriate for the BPXYINHE version.</li> <li>• The process group ID that was specified in the inheritance structure is less than zero or has some other unsupported value.</li> </ul> <p>The following reason codes can accompany the return code: JROK, JRUserNameLenError, JRJsRacXtr, JRInheUserid, JRInheRegion, JRInheCPUTime, JRInheDynamber, JRInheAccountData, JRInheCWD, JRInheSetPgrp, JRInheVersion, and JRInheLength..</p>

## Example

The following is an example of a parent program that uses `spawn` to create a child process.

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
#include <spawn.h>
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>

/* This program uses spawn instead of fork/exec to create a child
 * process and uses unnamed pipes to allow the parent and child to
 * exchange communication.
 */

void main(int argc, char *argv[]) {
    pid_t child;
    int fd_count, fd_map[10];
    struct inheritance inherit;
    const char *c_argv[10], *c_envp[10];
    char buf[256];
```

```

int nbytes;

int c_stdin[2], c_stdout[2], c_stderr[2]; /* Pipes for child
    * communication */

/* Create pipes to communicate with child via stdin/stdout/stderr */
if(pipe(c_stdin) ||
    pipe(c_stdout) ||
    pipe(c_stderr) ) {
    perror("Bad pipe");
    exit(-1);
}

/* Set up file descriptor map for child process */
fd_map[0]=dup(c_stdin[0]); /* child stdin is read end of pipe */
fd_map[1]=dup(c_stdout[1]); /* child stdout is write end of pipe */
fd_map[2]=dup(c_stderr[1]); /* child stderr is write end of pipe */
fd_count=3;

/* Close unused end of pipes for the parent */
close(c_stdin[0]); close(c_stdout[1]); close(c_stderr[1]);

/* Build the argument structure for child arguments.
    * [0] is the program name */
c_argv[0]="spawn";
c_argv[1]="arg1"; c_argv[2]="arg2"; c_argv[3]=NULL;

/* Build the environment structure which defines the child's
    * environment variables */
c_envp[0]="TEST_ENV=YES"; c_envp[1]="BPX_SHAREAS=NO"; c_envp[2]=NULL;

/* Spawn the child process */
child=spawn("spawn", fd_count, fd_map, &inherit, c_argv, c_envp);
if(child==-1) {
    perror("Error on spawn");
    exit(-1);
}
else printf("Spawned %i\n", child);

/* Test interaction with the child process */
printf("parent: Asking child, \"what are you doing?\\n\\n\");
strcpy(buf, "child from parent: what are you doing?\\n");
if(write(c_stdin[1], buf, sizeof(buf))==-1) {
    perror("write stdout");
    exit(-1);
}

memset(buf, 0, 255); /* Just zeroing out the buffer */
printf("parent: reading from child now\\n");
if((nbytes=read(c_stdout[0], buf, 255))==-1) {
    perror("read error:");
    exit(-1);
}
printf("parent: child says, \"%s\\n\", buf);

/* Cleanup pipes before exiting */
close(c_stdin[1]); close(c_stdout[0]); close(c_stderr[0]);

exit(0);
}

```

## Example

The following is an example of a child program used by spawn.

```

#include <stdlib.h>
#include <stdio.h>

/* This is a sample child program used by spawn. This program will
    * work stand-alone as well as from spawn or fork/exec. */

extern char ** environ; /* External used to access the environment

```



```

                                directly instead of using getenv */

void main(int argc, char *argv[]) {

    char *e, **env=environ; /* Used to step through the environment
                             * to write out to file. */
    char buf[256]={0};
    FILE *fp=fopen("spawntest.out","w");
    int i;

    /* Print out the environment variables */
    i=0;
    fprintf(fp, "Environment:\n");
    while(e=env[i++]) fprintf(fp, "%s\n", e);
    fprintf(fp, "\n\n");

    /* Just to prove getenv works */
    fprintf(fp, "TEST_ENV envvar = %s", getenv("TEST_ENV"));

    /* Print out the command line arguments */
    i=0;
    fprintf(fp, "Args:\n");
    while(e=argv[i++]) fprintf(fp, "%s\n", e);
    fprintf(fp, "\n\n");

    /* Print out what was sent on stdin */
    fprintf(fp, "Child/parent\n");
    if(!gets(buf)) {
        perror(stdin);
        exit(-1);
    }
    fprintf(fp, "child from parent: %i bytes, [%s]\n", strlen(buf), buf);

    /* Send something to stdout */
    printf("nothing");

    fclose(fp);
    exit(0);
}

```

## Related Information

- “spawn.h” on page 42
- “sys/wait.h” on page 51
- “alarm() — Set an Alarm” on page 102
- “chmod() — Change the Mode of a File or Directory” on page 174
- “exit() — End Program” on page 330
- “\_exit() — End a Process and Bypass the Cleanup” on page 332
- “fcntl() — Control Open File Descriptors” on page 350
- “fork() — Create a New Process” on page 422
- “kill() — Send a Signal to a Process” on page 728
- “rexec() — Execute Commands One at a Time on a Remote Host” on page 1143
- “setuid() — Set the Effective User ID” on page 1278
- “\_\_spawn2() - spawnp2() — Spawn a New Process Using Enhanced Inheritance Structure” on page 1388
- “stat() — Get File Information” on page 1404
- “times() — Get Process and Child Process Times” on page 1572
- “wait() — Wait for a Child Process to End” on page 1687
- “waitpid() — Wait for a Specific Child Process to End” on page 1692

## \_\_spawn2() - spawnp2() — Spawn a New Process Using Enhanced Inheritance Structure

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	POSIX(ON)

### Format

```
#include <spawn.h>
pid_t __spawn2(const char *path,
               const int fd_count,
               const int fd_map[],
               const struct __inheritance * inherit,
               const char *argv[],
               const char *envp[]);

pid_t __spawnp2(const char *file,
                const int fd_count,
                const int fd_map[],
                const struct __inheritance * inherit,
                const char *argv[],
                const char *envp[]);
```

### General Description

The `__spawn2()` and `__spawnp2()` functions create a new process from the specified process image. The new process image is constructed from a regular executable file called the new process image file.

To execute a C program as a result of this call, enter the function call as follows:

```
int main (int argc, char *argv[]);
```

Where *argc* is the argument count and *argv* is an array of character pointers to the arguments themselves. In addition, the following variable:

```
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings. The *argv* and *environ* arrays are each terminated by a NULL pointer. The NULL pointer terminating the *argv* array is not counted in *argc*.

Supported parameters are:

#### Parameter Description

<i>path</i>	Pathname used by <code>__spawn2()</code> that identifies the new process image file to execute.
<i>file</i>	Used by <code>__spawnp2()</code> to construct a pathname that identifies the new process image file. If the file parameter contains a slash character, the file parameter shall be used as a pathname for the new process image file. Otherwise, the path prefix for this file shall be obtained by a search of the directories passed as the environment variable <code>PATH</code> .

*fd-count* Specifies the number of file descriptors the child process shall inherit. It may take values from zero to {OPEN\_MAX}. Except those file descriptors designated by SPAWN\_FDCLOSED, each of the child's file descriptors, *x*, in the range zero to *fd-count*-1 shall inherit descriptor *fd\_map(x)* from the parent process.

The files from *fd-count* through {OPEN\_MAX} are closed in the child process, as are any elements of *fd\_map* designated as SPAWN\_FDCLOSED.

*fd-map* If the *fd\_map* parameter is NULL, *fd-count* and *fd\_map* are ignored. All file descriptors except those with the FD\_CLOEXEC or FD\_CLOFORK attribute are inherited without reordering. File descriptors with the FD\_CLOEXEC or FD\_CLOFORK attribute are closed under simple inheritance.

For those file descriptors that remain open, all other attributes of the associated file descriptor object and open file description shall remain unchanged by this operation.

Directory streams open in the calling process image shall be closed in the new process image, with the effect of the closedir() operation.

If an element of *fd\_map* refers to an invalid file descriptor, then the (EBADF) error status shall be posted by \_\_spawn2() or \_\_spawnp2().

The FD\_CLOEXEC and FD\_CLOFORK file descriptor attributes are never inherited.

The FD\_CLOEXEC and FD\_CLOFORK file descriptor attributes have no effect on inheritance when the *fd\_map* parameter is not NULL.

*inherit* The name of a data area that contains the inheritance structure.

The 'struct \_\_inheritance' is defined as follows:

```
struct __inheritance {
    short      flags;           -- Flags
    pid_t      pgroup;         -- Process group
    sigset_t    sigmask;       -- Signal mask
    sigset_t    sigdefault;    -- Signals set to SIG_DFL
    int        ctltyfd;        -- Cntl tty FD for tcsetpgrp()
    char        *cwdptr;       -- Pointer to the users CWD
    int        cwdlen;         -- Length of the users CWD
    int        acctdatalen;    -- Length of account data area
    char        *acctdataptr;  -- Ptr to account data area
    int        umask;          -- Users UMASK
    char        userid[9];     -- New A.S. user identity
    char        jobname[9];    -- New A.S. jobname
    int        regionsize;     -- New A.S. region size
    int        timelimit;      -- New A.S. time limit
}
```

The inherit.flags effect spawn() and spawnp() as follows:

SPAWN\_SETGROUP

If the SPAWN\_SETGROUP flag is set in *inherit.flags*, then the child's process group shall be as specified in *inherit.pgroup*.

If the SPAWN\_SETGROUP flag is set in *inherit.flags* and *inherit.pgroup* is set to SPAWN\_NEWPGROUP, then the

child shall be in a new process group with a process group ID equal to its process ID.

If the SPAWN\_SETGROUP flag is not set in *inherit.flags*, the newchild shall inherit the parent's process group ID.

#### SPAWN\_SETSIGMASK

If the SPAWN\_SETSIGMASK flag is set in *inherit.flags*, the child process shall initially have the signal mask specified in *inherit.sigmask*.

#### SPAWN\_SETSIGDEF

If the SPAWN\_SETSIGDEF flag is set in *inherit.flags*, the signals specified in *inherit.sigdefault* shall be set to their default actions in the child process. Signals set the default action in the parent process shall be set to the default action in the new process.

Signals set to be caught by the calling process shall be set to the default action in the child process.

Signals set to be ignored by the calling process shall be set to be ignored by the new process, unless otherwise specified by the SPAWN\_SETSIGDEF flag being set in *inherit.flags* and the signal being indicated *inherit.sigdefault*.

#### SPAWN\_SETTCPGRP

If the SPAWN\_SETTCPGRP flag is set in *inherit.flag*, the file descriptor specified in *inherit.clttyfd* is used to set the controlling terminal file descriptor (tcsetpgrp()) for the child's foreground process group. The child's foreground process group is inherited from the parent, unless the SPAWN\_SETGROUP flag in *inherit.flags* is set, indicating that the value specified in *inherit.pgroup* is to be used to determine the child's process group.

*argv* The value in the first element of *argv* should point to a filename that is associated with the process being started by the spawn2() or spawnp2() operation.

The number of bytes available for the new process' combined argument and environment lists is {ARG\_MAX}.

*envp* The value *envp* contains the list of environmental variables that is to be passed to the specified program.

If the set-user-ID mode bit of the new process image file is set, the effective user ID of the new process image shall be set to the owner ID of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image shall be set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image shall remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image shall be saved (as the saved set-user-ID and set-group-ID) for use by the setuid() function.

The new process image shall inherit the following attributes of the calling process image:

- Process group ID (unless the SPAWN\_SETGROUP flag is set in *inherit.flags*).
- Session membership.
- Real user ID.
- Real group ID.
- Supplementary group IDs.
- Priority.
- Current working directory.
- File creation mask.
- Signal mask (unless the SPAWN\_SETSIGMASK flag is set in *inherit.flags*).
- Signal actions specified as default (SIG\_DFL).
- Signal actions specified as ignore (SIG\_IGN) (except as modified by *inherit.sigdefault* and the SPAWN\_SETSIGDEF flag set in *inherit.flags*).

The following are differences between the parent process and child:

- Signals set to be caught by the calling process shall be set to the default action (SIG\_DFL).
- The process and system utilization times for the child are set to zero.
- Any file locks previously set by the parent are not inherited by the child.
- The child process has no alarms set and has no interval timers set.
- The child has no pending signals.
- Memory mappings established by the parent are not inherited by the child.

If the process image was read from a writable file system, then upon successful completion, the \_\_spawn2() and \_\_spawnp2() functions will mark for update the *st\_time* field of the new process image file.

If the \_\_spawn2() or \_\_spawnp2() function is successful, the new child process image file shall be opened with all the effects of the open() function.

All the following inherit flags are used by \_\_spawn2() and \_\_spawnp2():

#### SPAWN\_SETCWD

Specifies the Current Working Directory that the child process will run when first created. This will override the CWD that would normally be set up or propagated by the child process.

#### SPAWN\_SETUMASK

Specifies the UMASK that the child process will run when first created. This will override the UMASK that would normally be set up in the child process. The invoker must have superuser privileges to specify UMASK.

#### SPAWN\_SETUSERID

When this flag is set, this attribute will be the equivalent of the \_BPX\_USERID environment variable. If specified, the invoking userid will be checked for daemon authority. If the invoker is authorized and the userid is valid, the child process will be created with RACF identity and POSIX permissions associated with the input USERID. If not authorized or the userid is invalid, the \_\_spawn2() or \_\_spawnp2() function will fail. If the USERID value is specified, any value in \_BPX\_USERID will be ignored.

#### SPAWN\_SETREGIONSZ

Specifies the number of megabytes the child process will have available for private storage. The authority/ranges required will be as per RLIMIT\_AS rules. Unless the invoker has superuser privileges, the

region size range will be checked and if it exceeds the hard limit, the `__spawn2()` or `__spawnp2()` function will fail. This value will override `RLIMIT_AS` or the normal spawn propagation rules.

**SPAWN\_SETTIMELIMIT**

Specifies the number of seconds of CPU time that is allowed by the child process before receiving a `SIGXCPU` signal. Unless the invoker has superuser privileges, the time limit range will be checked and if it exceeds the hard limit, the `__spawn2()` or `__spawnp2()` function will fail. This value will override the `RLIMIT_CPU` or the normal spawn propagation rules.

**SPAWN\_SETACCTDATA**

Specifies account data of the child process. The format and length will be as per the `_BPX_ACCT_DATA` environmental variable. No special authority is needed to change account data. This will override the target userid's default account data and any value specified on the `_BPX_ACCT_DATA` will be ignored.

**SPAWN\_SETJOBNAME**

When this flag is set, it is the equivalent of the `_BPX_JOBNAME` environment variable. If specified, the invoking userid will be checked for superuser authority. If the invoker is authorized and the jobname is valid, the child process will be created with the specified jobname. If not authorized or the jobname is invalid, `__spawn2()` or `__spawnp2()` will ignore the `JOBNAME` attribute and continue. If the `JOBNAME` value is specified, any value in `_BPX_JOBNAME` will be ignored.

For more information on the use of inheritance structure flags, see *OS/390 UNIX System Services Programming: Assembler Callable Services Reference*.

**Returned Value**

Upon successful completion, `__spawn2()` or `__spawnp2()` returns the process ID of the child process to the parent. If the function is unsuccessful, a value of -1 is returned to the parent process, no child is created, and *errno* is set to indicate the error. The following are the possible values of *errno*:

- |                |  |
|----------------|--|
| <b>EACCES</b>  | Search permission is denied for a directory in the path of the new process image file or the new process image file denies execution permission, or the the new process image file is not a regular file and the implementation does not support execution of files of its type. |
| <b>E2BIG</b>   | The number of bytes used by the argument and environment list of the new process image is greater than the system imposed limit of <code>{ARG_MAX}</code> bytes.   |
| <b>ENOENT</b>  | One or more components of the pathname of the new process image file do not exist or the path or file paramter is empty.   |
| <b>ELOOP</b>   | A loop exists in symbolic links encountered during resolution of the filename argument. This error is issued if more than 8 symbolic links are detected..  |
| <b>ENOTDIR</b> | A component of the path prefix of the new process image file is not a directory.   |

**ENOEXEC** The new process image file has the appropriate access permission, but is not in the proper format.

**Note:**

Reason codes further qualify the errno. For most of the reason codes, see *OS/390 UNIX System Services Messages and Codes*.

For ENOEXEC, the reason codes are:

Reason Code	Explanation
X'xxx0C27'	The target HFS file is not in the correct format to be an executable file.
X'xxx0C31'	The target HFS file is built at a level that is higher than that supported by the running system.

**ENAMETOOLONG**

The length of the path or file parameter, or an element of the environmental variable PATH prefixed to a file, exceeds {PATH\_MAX}, or a pathname component is longer than {NAME\_MAX} and {\_POSIX\_N\_TRUNC\_} is in effect for that file.

**EAGAIN** The system lacked the necessary resources to create another process, or the system-imposed limit on the total number of processes under execution by a single user would be exceeded. The resources required to let another process be created are not available, or you have already reached the maximum number of processes or UIDs you are allowed to create. This error will also be generated if \_BPX\_USERID or INHEUSERID was specified and the username was not defined to SAF with an OE segment.

**ENOMEM** The new process requires more memory than is permitted by the hardware or operating system.

**EFAULT** The system detected an invalid address while attempting to use a parameter of the call.

**EMVSSAF2ERR**

The executable file is a set-user-ID or set-group-ID file and the owner's UID or GID is not defined to the Security Access Facility (SAF).

**EBADF** An entry in the *fd\_map* array refers to an invalid file descriptor or the controlling terminal file descriptor specified in the \_\_inheritance structure.

**ENOTTY** tcsetpgrp() failed for the specified controlling terminal file descriptor in \_\_inheritance structure. The failure occurred because the calling process does not have a controlling terminal, or the specified file descriptor is not associated with the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.

**ESRCH** The process group ID specified in the \_\_inheritance structure is not that of the process group in the calling process' session.

EPERM	<p>The spawn failed for one of the following reasons:</p> <ul style="list-style-type: none"><li>• The spawned process is not a process group leader.</li><li>• The <code>_BPX_USERID</code> environment variable was specified, and the invoker does not have appropriate privileges to change the MVS identity.</li><li>• The invoker does not have the appropriate privileges to change one or more of the attributes specified in the inheritance structure (BPXYINHE).</li></ul> <p>The following reason codes can accompany the return code: JROK, JRNoChangeIdentity, JRInheUserid, JRInheRegion, JRInheCPUTime, JRInheUmask, and JRInheCWD.</p>
EINVAL	<p>One or more of the following conditions were detected:</p> <ul style="list-style-type: none"><li>• The username that was specified on the <code>_BPX_USERID</code> environment variable has an incorrect length.</li><li>• An attribute that was specified in the inheritance structure (BPXYINHE) is not valid or contains an unsupported value.</li><li>• The version number that was specified for the inheritance structure (BPXYINHE) is not valid.</li><li>• The inheritance structure length that was specified by the <code>Inherit_area_len</code> parameter or within the inheritance structure does not contain a length that is appropriate for the BPXYINHE version.</li><li>• The process group ID that was specified in the inheritance structure is less than zero or has some other unsupported value.</li></ul> <p>The following reason codes can accompany the return code: JROK, JRUserNameLenError, JRJsRacXtr, JRInheUserid, JRInheRegion, JRInheCPUTime, JRInheDynamber, JRInheAccountData, JRInheCWD, JRInheSetPgrp, JRInheVersion, and JRInheLength..</p>

## Related Information

- “spawn.h” on page 42
- “sys/wait.h” on page 51
- “alarm() — Set an Alarm” on page 102
- “chmod() — Change the Mode of a File or Directory” on page 174
- “exit() — End Program” on page 330
- “\_exit() — End a Process and Bypass the Cleanup” on page 332
- “fcntl() — Control Open File Descriptors” on page 350
- “fork() — Create a New Process” on page 422
- “kill() — Send a Signal to a Process” on page 728
- “rexec() — Execute Commands One at a Time on a Remote Host” on page 1143
- “setuid() — Set the Effective User ID” on page 1278
- “spawn() - spawnp() — Spawn a New Process” on page 1377
- “stat() — Get File Information” on page 1404
- “times() — Get Process and Child Process Times” on page 1572
- “wait() — Wait for a Child Process to End” on page 1687
- “waitpid() — Wait for a Specific Child Process to End” on page 1692



**sprintf() — Format and Write Data to Buffer**

The information for this function is included in “fprintf() - printf() - sprintf() — Format and Write Data” on page 436.

## sqrt() — Calculate Square Root

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double sqrt(double x);
```

### General Description

Calculates the square root of the positive value, *x*.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

Returns the square root result. If *x* is negative, the function sets *errno* to *EDOM*, and returns the value 0. If the correct value would cause underflow, zero is returned and the value of the macro *ERANGE* is stored in *errno*.

### Example CBC3BS30

```
/* CBC3BS30
   This example computes the square root of the quantity passed as the
   first argument to main.
   It prints an error message if you pass a negative value.
*/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char ** argv)
{
    char * rest;
    double value;

    if ( argc != 2 )
        printf( "Usage: %s value\n", argv[0] );
    else
    {
        value = strtod( argv[1], &rest );
        if ( value < 0.0 )
            printf( "sqrt of a negative number\n" );
        else
            printf("sqrt( %f ) = %f\n", value, sqrt( value ));
    }
}
```

### Output

If the input is 45, then the output should be:

```
sqrt( 45.000000 ) = 6.708204
```

**Related Information**

- “math.h” on page 35
- “exp() — Calculate Exponential Function” on page 334
- “hypot() — Calculate Hypotenuse” on page 650
- “log() — Calculate Natural Logarithm” on page 762
- “log10() — Calculate Base 10 Logarithm” on page 767
- “pow() — Raise to Power” on page 916

## srand() — Set Seed for rand() Function

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
void srand(unsigned int seed);
```

### General Description

srand() uses its argument *seed* as a seed for a new sequence of pseudorandom numbers to be returned by subsequent calls to rand(). If srand() is not called, the rand() seed is set as if srand(1) was called at program start. Any other value for *seed* sets the generator to a different starting point. The rand() function generates pseudorandom numbers.

Some people find it convenient to use the return value of the time() function as the argument to srand(), as a way to ensure random sequences of random numbers.

### Returned Value

Returns no value.

### Example

#### CBC3BS31

```
/* CBC3BS31
   This example first calls srand() with a value other than 1 to initiate
   the random value sequence. Then the program computes 5 random values for
   the array of integers called ranvals. If you repeat this code exactly,
   then the same sequence of random values will be generated.
*/
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int i, ranvals[5];

    srand(17);
    for (i = 0; i < 5; i++) {
        ranvals[i] = rand();
        printf("Iteration %d ranvals [%d] = %d\n", i+1, i, ranvals[i]);
    }
}
```

### Output

```
Iteration 1 ranvals [0] = 24107
Iteration 2 ranvals [1] = 16552
Iteration 3 ranvals [2] = 12125
Iteration 4 ranvals [3] = 9427
Iteration 5 ranvals [4] = 13152
```

**Related Information**

- “stdlib.h” on page 45
- “rand() — Generate Random Number” on page 1077

## srandom() — Use Seed to Initialize Generator for random()

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>
```

```
void srandom(unsigned seed);
```

### General Description

The `srandom()` function initializes the calling thread's current state array for the `random()` function using the value of *seed*.

### Returned Value

The `srandom()` function returns no value.

### Related Information

- “`stdlib.h`” on page 45
- “`random()` — A Better Random-Number Generator” on page 1079
- “`initstate()` — Initialize Generator for `Random()`” on page 670
- “`setstate()` — Change Generator for `random()`” on page 1275
- “`drand48()` — Pseudo-random Number Generator” on page 286
- “`rand()` — Generate Random Number” on page 1077

srand48() — Pseudo-random Number Initializer

Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>

void srand48(long int seedval);
```

General Description

The drand48(), erand48(), jrand48(), lrand48(), mrand48() and nrand48() functions generate uniformly distributed pseudorandom numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The lcong48(), seed48(), and srand48() functions are initialization functions, one of which should be invoked before either the drand48(), lrand48() or mrand48() function is called.

The drand48(), lrand48() and mrand48() functions generate a sequence of 48-bit integer values, X(i), according to the linear congruential formula:

$$X(n+1) = (aX(n) + c) \bmod (2^{**}48) \qquad n \geq 0$$

The initial values of X, a, and c are:

```
X(0) = 1
a    = 5deece66d (base 16)
c    = b          (base 16)
```

C/370 provides storage to save the most recent 48-bit integer value of the sequence, X(i). This storage is shared by the drand48(), lrand48() and mrand48() functions. The srand48() function is used to reinitialize the most recent 48-bit value in this storage. The srand48() function replaces the high order (leftmost) 32 bits of this storage with *seedval* argument value. The srand48() function replaces the low order 16 bits of this storage with the value **330E** (base 16).

The values a and c, may be changed by calling the lcong48() function. The srand48()function restores the initial values of a and c.

Special Behavior for OS/390 UNIX Services

You can make the srand48() function and other functions in the drand48 family thread specific by setting the environment variable `_RAND48` to the value `THREAD` before calling any function in the drand48 family.

If you do not request thread specific behavior for the drand48 family, C/370 serializes access to the storage for X(n), a and c by functions in the drand48 family when they are called by a multithreaded application.

If thread specific behavior is requested, calls to the `drand48()`, `lrand48()` and `mrnd48()` functions from thread `t` generate a sequence of 48-bit integer values,  $X(t,i)$ , according to the linear congruential formula:

$$X(t,n+1) = (a(t)X(t,n) + c(t)) \bmod (2^{**}48) \quad n \geq 0$$

C/370 provides thread specific storage to save the most recent 48-bit integer value of the sequence,  $X(t,i)$ . When the `srand48()` function is called from thread `t`, it reinitializes the most recent 48-bit value in this storage. The `srand48()` function replaces the high order (leftmost) 32 bits of this storage with *seedval* argument value. The `srand48()` function replaces the low order 16 bits of this storage with the value **330E** (base 16).

The values of  $a(t)$  and  $c(t)$  may be changed by calling the `lcong48()` function from thread `t`. When the `srand48()` function is called from this thread it restores the initial values of  $a(t)$  and  $c(t)$  for the thread which are:

$$\begin{aligned} a(t) &= 5deece66d \text{ (base 16)} \\ c(t) &= b \text{ (base 16)} \end{aligned}$$

### Returned Value

After the `srand48()` function has used the value of the argument *seedval*, to reinitialized storage for the most recent 48-bit integer value in the sequence,  $X(i)$ , and has restored the initial values of  $a$  and  $c$ , it returns.

### Special Behavior for OS/390 UNIX Services

If thread specific behavior is requested for the `drand48` family and the `srand48()` function is called on thread `t`, it uses the value of the argument, *seedval*, to reinitialize storage for the most recent 48-bit integer value in the sequence,  $X(t,i)$ , for the thread. It also restores the initial values of  $a(t)$  and  $c(t)$  for the thread. Then it returns.

### Related Information

- “`stdlib.h`” on page 45
- “`drand48()` — Pseudo-random Number Generator” on page 286
- “`erand48()` — Pseudo-random Number Generator” on page 314
- “`jrand48()` — Pseudo-random Number Generator” on page 724
- “`lcong48()` — Pseudo-random Number Initializer” on page 736
- “`lrand48()` — Pseudo-random Number Generator” on page 773
- “`mrnd48()` — Pseudo-random Number Generator” on page 843
- “`nrnd48()` — Pseudo-random Number Generator” on page 868
- “`seed48()` — Pseudo-random Number Initializer” on page 1156



**sscanf() — Read and Format Data from Buffer**

The information for this function is included in “fscanf() – scanf() – sscanf() — Read and Format Data” on page 464.

stat() — Get File Information

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

Format

```
#define _POSIX_SOURCE
#include <sys/stat.h>
```

```
int stat(const char *pathname, struct stat *info);
```

General Description

Gets status information about a specified file and places it in the area of memory pointed to by the *info* argument. The process does not need permissions on the file itself, but must have search permission on all directory components of the *pathname*. If the named file is a symbolic link, stat() resolves the symbolic link. It also returns information about the resulting file.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

The information is returned as shown in the following stat structure table, as defined in the sys/stat.h header file.

Table 34. Values Returned in the stat Structure	
Value	Description
mode_t st_mode	A bit string indicating the permissions and privileges of the file. Symbols are defined in the sys/stat.h header file to refer to bits in a mode_t value; these symbols are listed in “chmod() — Change the Mode of a File or Directory” on page 174.
ino_t st_ino	The serial number of the file.
dev_t st_dev	The numeric ID of the device containing the file.
nlink_t st_nlink	The number of links to the file.
uid_t st_uid	The numeric user ID (UID) of the file's owner.
gid_t st_gid	The numeric group ID (GID) of the file's group.
off_t st_size	For regular files, the file's size in bytes. For other kinds of files, the value of this field is unspecified.
time_t st_atime	The most recent time the file was accessed.
time_t st_ctime	The most recent time the status of the file was changed.
time_t st_mtime	The most recent time the contents of the file were changed.

Values for time\_t\_ are given in terms of seconds since epoch.

stat() updates the time-related fields before putting information in the stat structure.

You can examine properties of a `mode_t` value from the `st_mode` field using a collection of macros defined in the `sys/modes.h` header file. If *mode* is a `mode_t` value, and *genvalue* is an unsigned `int` value from the `stat` structure, then:

- `S_ISBLK(mode)`  
Is nonzero for block special files.
- `S_ISCHR(mode)`  
Is nonzero for character special files.
- `S_ISDIR(mode)`  
Is nonzero for directories.
- `S_ISEXTL(mode,genvalue)`  
Is nonzero for external links. (External links are introduced in MVS 5.1; they are not supported in MVS 4.3 or previous releases, where they return 0).
- `S_ISFIFO(mode)`  
Is nonzero for pipes and FIFO special files.
- `S_ISLNK(mode)`  
Is nonzero for symbolic links.
- `S_ISREG(mode)`  
Is nonzero for regular files.
- `S_ISSOCK(mode)`  
Is nonzero for sockets. (Sockets are introduced in MVS 5.1; they are not supported in MVS 4.3 or previous releases, where they return 0).

If `stat()` successfully determines this information, it stores it in the area indicated by the *info* argument. The size of the buffer determines how much information is stored; data that exceeds the size of the buffer is truncated.

## Returned Value

If successful, `stat()` returns zero. If unsuccessful, it returns the value `-1` and sets `errno` to one of the following:

- EACCES** The process does not have search permission on some component of the *pathname* prefix.
- EINVAL** *info* is a null pointer.
- ELOOP** A loop exists in symbolic links encountered during resolution of the *pathname* argument. This error is returned if more than `POSIX_SYMLINK` (defined in the `limits.h` header file) symbolic links are encountered during resolution of the *pathname* argument.
- ENAMETOOLONG** *pathname* is longer than `PATH_MAX` characters, or some component of *pathname* is longer than `NAME_MAX` characters while `_POSIX_NO_TRUNC` is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds `PATH_MAX`. The `PATH_MAX` and `NAME_MAX` values can be determined with `pathconf()`.
- ENOENT** There is no file named *pathname*, or *pathname* is an empty string.
- ENOTDIR** A component of the *pathname* prefix is not a directory.

**For XPG4.2 added the following:**

**EIO** An error occurred while reading from the file system.

**EOVERFLOW**

The file size exceeded the storage reserved for `st_size` in the **stat** structure.

**Example  
CBC3BS33**

```
/* CBC3BS33
   This example gets status information about a file.
*/
#define _POSIX_SOURCE
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <time.h>

main() {
    struct stat info;

    if (stat("/", &info) != 0)
        perror("stat() error");
    else {
        puts("stat() returned the following information about root f/s:");
        printf(" inode:  %d\n",    (int) info.st_ino);
        printf(" dev id:  %d\n",    (int) info.st_dev);
        printf(" mode:    %08x\n",   info.st_mode);
        printf(" links:   %d\n",    info.st_nlink);
        printf(" uid:     %d\n",    (int) info.st_uid);
        printf(" gid:     %d\n",    (int) info.st_gid);
        printf("created:  %s",      ctime(&info.st_createtime));
    }
}
```

**Output**

```
stat() returned the following information about root f/s:
inode:  0
dev id:  1
mode:    010001ed
links:   11
uid:     0
gid:     500
created:  Fri Jun 16 10:07:55 1995
```

**Related Information**

- “sys/stat.h” on page 48
- “sys/types.h” on page 49
- “remove() — Delete File” on page 1133
- “chmod() — Change the Mode of a File or Directory” on page 174
- “chown() — Change the Owner or Group of a File or Directory” on page 177
- “creat() — Create a New File or Rewrite an Existing One” on page 233
- “dup() — Duplicate an Open File Descriptor” on page 288
- “rexec() — Execute Commands One at a Time on a Remote Host” on page 1143
- “fcntl() — Control Open File Descriptors” on page 350
- “fstat() — Get Status Information about a File” on page 479

- “link() — Create a Link to a File” on page 749
- “lstat() — Get Status of File or Symbolic Link” on page 778
- “mkdir() — Make a Directory” on page 817
- “mkfifo() — Make a FIFO Special File” on page 820
- “open() — Open a File” on page 872
- “pipe() — Create an Unnamed Pipe” on page 907
- “read() — Read From a File or Socket” on page 1080
- “readlink() — Read the Value of a Symbolic Link” on page 1091
- “symlink() — Create a Symbolic Link to a Path Name” on page 1479
- “unlink() — Remove a Directory Entry” on page 1660
- “utime() — Set File Access and Modification Times” on page 1664
- “write() — Write Data on a File or Socket” on page 1780

statvfs() — Get File System Information

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/statvfs.h>
int statvfs(const char *pathname, struct statvfs *fsinfo);
```

General Description

The statvfs() function obtains information about the file system containing the file named by *pathname* and stores it in the area of memory pointed to by the *fsinfo* argument. The process does not need permissions on the file itself, but must have search permission on all directory components of the *pathname*.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

The information is returned in the following statvfs structure, as defined in the sys/statvfs.h header file.

Table 35 (Page 1 of 2). Values Returned in the statvfs Structure	
Value	Description
char f_OEcbid[4]	The structure acronym (eye catcher).
int f_OEcblen	The length of the structure.
unsigned long f_bsize	The file system block size.
unsigned long f_blocks	The total number of blocks on the file system in units of f_frsize.
unsigned long f_OEusedspace	The allocated space in block size units.
unsigned long f_bavail	The number of free blocks available to non-privileged process.
unsigned long f_fsid	The file system ID.
unsigned long f_flag	A bit string indicating file system status.
int f_OEmaxfilesizehw	The high word of maximum file size.
unsigned long f_OEmaxfilesizelw	The low word of maximum file size.
unsigned long f_frsize	The fundamental file system block size.
unsigned long f_bfree	The total number of free blocks.
unsigned long f_files	The total number of file serial numbers.
unsigned long f_ffree	The total number of free file serial numbers.
unsigned long f_favail	The number of file serial numbers available to non-privileged process.
unsigned long f_namemax	The maximum filename length.

Table 35 (Page 2 of 2). Values Returned in the statvfs Structure

Value	Description
unsigned long f_OEinvarsec	The number of seconds the file system will remain unchanged.

The following flags can be returned in the `f_flag` member:

ST_RDONLY	read-only file system
ST_NOSUID	setuid/setgid bits ignored by exec
ST_OEEXPORTED	file system is exported

If `statvfs()` successfully determines this information, it stores in the area indicated by the `fsinfo` argument. The size of the buffer determines how much information is stored; data that exceeds the size of the buffer is truncated.

### Returned Value

If successful, `statvfs()` returns zero. If unsuccessful, `statvfs()` returns `-1`, and returns the error value in `errno`. The following are the possible values of `errno`:

EACCES	The process does not have search permission on some component of the <i>pathname</i> prefix.
EIO	An I/O error has occurred while reading the file system.
EINTR	A signal was caught during the execution of the function.
ELOOP	A loop exists in symbolic links encountered during resolution of the <i>pathname</i> argument. This error is issued if more than the system-defined limit of symbolic links, 8, are detected in the resolution of <i>pathname</i> .
ENAMETOOLONG	The length of the <i>pathname</i> exceeds <code>PATH_MAX</code> or a component of <i>pathname</i> is longer than <code>NAME_MAX</code> .
ENOENT	There is no file named <i>pathname</i> , or <i>pathname</i> is an empty string.
ENOTDIR	A component of the <i>pathname</i> prefix is not a directory.

### Example

```
#include <sys/statvfs.h>
#include <stdio.h>

main() {
    int fd;
    struct statvfs buf;

    if (statvfs(".", &buf) == -1)
        perror("statvfs() error");
    else {
        printf("each block is %d bytes big\n", fs,
            buf.f_bsize);
        printf("there are %d blocks available out of a total of %d\n",
            buf.f_bavail, buf.f_blocks);
    }
}
```

```

        printf("in bytes, that's %.0f bytes free out of a total of %.0f\n",
               ((double)buf.f_bavail * buf.f_bsize),
               ((double)buf.f_blocks * buf.f_bsize));
    }
}

```

### Output

each block is 4096 bytes big  
 there are 2089 blocks available out of a total of 2400  
 in bytes, that's 8556544 bytes free out of a total of 9830400

### Related Information

- “sys/statvfs.h” on page 49
- “chmod() — Change the Mode of a File or Directory” on page 174
- “chown() — Change the Owner or Group of a File or Directory” on page 177
- “creat() — Create a New File or Rewrite an Existing One” on page 233
- “dup() — Duplicate an Open File Descriptor” on page 288
- “exec Functions” on page 322
- “fcntl() — Control Open File Descriptors” on page 350
- “link() — Create a Link to a File” on page 749
- “mknod() — Make a Directory or File” on page 823
- “open() — Open a File” on page 872
- “pipe() — Create an Unnamed Pipe” on page 907
- “read() — Read From a File or Socket” on page 1080
- “time() — Determine Current Time” on page 1570
- “unlink() — Remove a Directory Entry” on page 1660
- “utime() — Set File Access and Modification Times” on page 1664
- “write() — Write Data on a File or Socket” on page 1780



## step() — Pattern Match with Regular Expression

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <regex.h>
```

```
int step(const char *string, const char *expbuf);
```

```
extern char *loc1, *loc2;
```

### General Description

The `step()` function attempts to match an input string of characters with the compiled regular expression which was obtained by an earlier call to `compile()`.

The first parameter *string* is a pointer to a string of characters to be checked for a match.

*expbuf* is the pointer to the regular expression which was previously obtained by a call to `compile()`.

### Notes:

1. The external variables *cirf*, *sed*, and *nbra* are reserved.
2. The application must provide the proper serialization for the `compile()`, `step()`, and `advance()` functions if they are run under a multi-threaded environment.
3. The `compile()`, `step()` and `advance()` functions are provided for historical reasons. New applications should use the new functions `fnmatch()`, `glob()`, `regcomp()`, and `regexexec()`, which provide full internationalized regular expression functionality compatible with ISO POSIX.2 standard.

### Returned Value

The `step()` function returns nonzero if some substring of *string* matches the regular expression in *expbuf*. It returns 0 if there is no match.

If there is a match, the `step()` function sets two external pointers, as follows:

- The variable *loc1* points to the first character that matched the regular expression.
- The variable *loc2* points to the character after the last character that matched the regular expression.

For example, if the regular expression matches the entire input *loc1* will point to the first character of *string* and *loc2* will point to the null at the end of *string*.

## **Related Information**

- “`regex.h`” on page 40
- “`advance()` — Pattern Match Given a Compiled Regular Expression” on page 88
- “`compile()` — Compile Regular Expression” on page 208
- “`fnmatch()` — Match Filename or Pathname” on page 415
- “`glob()` — Generate Pathnames Matching a Pattern” on page 636
- “`regcomp()` — Compile Regular Expression” on page 1120
- “`regexec()` — Execute Compiled Regular Expression” on page 1126

## strcasecmp() — Case-insensitive String Comparison

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <strings.h>
```

```
int strcasecmp(const char *string1, const char *string2);
```

### General Description

The `strcasecmp()` function compares, while ignoring differences in case, the string pointed to by *string1* to the string pointed to by *string2*.

The string arguments to the function must contain a null character (`\0`) marking the end of the string.

The `strcasecmp()` function is locale-sensitive.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

The `strcasecmp()` function returns a value indicating the relationship between the strings, while ignoring case, as follows:

Value	Meaning
< 0	String pointed to by <i>string1</i> is less than string pointed to by <i>string2</i> .
= 0	String pointed to by <i>string1</i> is equal to string pointed to by <i>string2</i> .
> 0	String pointed to by <i>string1</i> is greater than string pointed to by <i>string2</i> .

There are no `errno` values defined for `strcasecmp()`.

### Related Information

- “strings.h” on page 46
- “strncasecmp() — Case-insensitive String Comparison” on page 1437
- “strcspn() — Compare Strings” on page 1424

## strcat() — Concatenate Strings

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <string.h>
```

```
char *strcat(char *string1, const char *string2);
```

### General Description

The `strcat()` built-in function concatenates *string2* with *string1* and ends the resulting string with the null character. In other words, `strcat()` appends a copy of the string pointed to by *string2*—including the terminating null byte—to the end of a string pointed to by *string1*, with its last byte (that is, the terminating null byte of *string1*) overwritten by the first byte of the appended string.

Do not use a literal string for a *string1* value, although *string2* may be a literal string.

If the storage of *string1* overlaps the storage of *string2*, the behavior is undefined.

### Returned Value

Returns the value of *string1*, the concatenated string.

### Example

#### CBC3BS34

```
/* CBC3BS34
   This example creates the string "computer program" using strcat().
*/
#include <stdio.h>
#include <string.h>

#define SIZE 40

int main(void)
{
    char buffer1[SIZE] = "computer";
    char * ptr;

    ptr = strcat( buffer1, " program" );
    printf( "buffer1 = %s\n", buffer1 );
}
```

### Output

```
buffer1 = computer program
```

**Related Information**

- “string.h” on page 46
- “strchr() — Search for Character” on page 1416
- “strcmp() — Compare Strings” on page 1418
- “strcpy() — Copy String” on page 1422
- “strcspn() — Compare Strings” on page 1424
- “strncat() — Concatenate Strings” on page 1438

## strchr() — Search for Character

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <string.h>
```

```
char *strchr(const char *string, int c);
```

### General Description

The `strchr()` built-in function finds the first occurrence of `c` converted to `char`, in the string `*string`. The character `c` can be the null character (`\0`); the ending null character of `string` is included in the search.

The `strchr()` function operates on null-terminated strings. The string argument to the function *must* contain a null character (`\0`) marking the end of the string.

### Returned Value

Returns a pointer to the first occurrence of `c` (converted to a character) in `string`. The function returns a null pointer if the character is not found.

### Example CBC3BS35

```
/* CBC3BS35
   This example finds the first occurrence of the character p in "computer
   program".
*/
#include <stdio.h>
#include <string.h>

#define SIZE 40

int main(void)
{
    char buffer1[SIZE] = "computer program";
    char * ptr;
    int   ch = 'p';

    ptr = strchr( buffer1, ch );
    printf( "The first occurrence of %c in '%s' is '%s'\n",
           ch, buffer1, ptr );
}
```

### Output

The first occurrence of p in 'computer program' is 'puter program'

**Related Information**

- “string.h” on page 46
- “memchr() — Search Buffer” on page 809
- “strcat() — Concatenate Strings” on page 1414
- “strcmp() — Compare Strings” on page 1418
- “strcpy() — Copy String” on page 1422
- “strcspn() — Compare Strings” on page 1424
- “strncmp() — Compare Strings” on page 1440
- “strpbrk() — Find Characters in String” on page 1444
- “strrchr() — Find Last Occurrence of Character in String” on page 1449
- “strspn() — Search String” on page 1451

## strcmp() — Compare Strings

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <string.h>
```

```
int strcmp(const char *string1, const char *string2);
```

### General Description

The `strcmp()` built-in function compares the string pointed to by *string1* to the string pointed to by *string2*. The string arguments to the function must contain a null character (`\0`) marking the end of the string.

The relation between the strings is determined by subtracting:  $string1[i] - string2[i]$ , as  $i$  increases from 0 to *strlen* of the smaller string. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type `unsigned char`) that differ in the strings being compared. This function is *not* locale-sensitive.

### Returned Value

Returns a value indicating the relationship between the strings, as listed below.

Value	Meaning
Less than 0	String pointed to by <i>string1</i> less than string pointed to by <i>string2</i>
0	String pointed to by <i>string1</i> equivalent to string pointed to by <i>string2</i>
Greater than 0	String pointed to by <i>string1</i> greater than string pointed to by <i>string2</i>

### Example

#### CBC3BS36

```
/* CBC3BS36
   This example compares the two strings passed to main using strcmp().
*/
#include <stdio.h>
#include <string.h>

int main(int argc, char ** argv)
{
    int result;

    if ( argc != 3 ) {
        printf( "Usage: %s string1 string2\n", argv[0] );
    }
    else {
        result = strcmp( argv[1], argv[2] );

        if ( result == 0 )
```



```

        printf( "\"%s\" is identical to \"%s\"\n", argv[1], argv[2]
);
    else if ( result < 0 )
        printf( "\"%s\" is less than \"%s\"\n", argv[1], argv[2] );
    else
        printf( "\"%s\" is greater than \"%s\"\"
    }
}

```

### Output

If the input is the strings "is this first?" and "is this before that one?" then the expected output is:

"is this first?" is greater than "is this before that one?"

### Related Information

- "string.h" on page 46
- "memcmp() — Compare Bytes" on page 811
- "strcspn() — Compare Strings" on page 1424
- "strncmp() — Compare Strings" on page 1440
- "strpbrk() — Find Characters in String" on page 1444
- "strrchr() — Find Last Occurrence of Character in String" on page 1449
- "strspn() — Search String" on page 1451

## strcoll() — Compare Strings

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <string.h>
```

```
int strcoll(const char *string1, const char *string2);
```

### General Description

Compares the string pointed to by *string1* against the string pointed to by *string2*, both interpreted according to the information in the LC\_COLLATE category of the current locale.

### Returned Value

Returns a value indicating the relationship between the strings, as listed below.

Value	Meaning
Less than 0	string pointed to by <i>string1</i> less than string pointed to by <i>string2</i>
0	string pointed to by <i>string1</i> equivalent to string pointed to by <i>string2</i>
Greater than 0	string pointed to by <i>string1</i> greater than string pointed to by <i>string2</i>

### Notes:

- The strcoll() function may need to allocate additional memory to perform the comparison algorithm specified in the LC\_COLLATE. If the memory request cannot be satisfied (by malloc()), strcoll() fails.
- If the locale supports double-byte characters (MB\_CUR\_MAX specified as 4), the strcoll() function validates the multibyte characters, whereas previously the strcoll() function did not validate the string. The strcoll() function will fail if the string contains invalid multibyte characters.
- If MB\_CUR\_MAX is specified as 4, but the charmap file does not specify the DBCS characters, the DBCS characters will collate after the single-byte characters.

### Example

#### CBC3BS37

```
/* CBC3BS37
   This example compares the two strings passed to main.
*/
#include <stdio.h>
#include <string.h>

int main(int argc, char ** argv)
{
    int result;
```

```

if ( argc != 3 ) {
    printf( "Usage: %s string1 string2\n", argv[0] );
}
else {

    result = strcoll( argv[1], argv[2] );

    if ( result == 0 )
        printf( "\"%s\" is identical to \"%s\"\n", argv[1],; argv[2]
);
    else if ( result < 0 )
        printf( "\"%s\" is less than \"%s\"\n", argv[1], argv[2] );
    else
        printf( "\"%s\" is greater than \"%s\"\n", argv[1],
argv[2] );
}
}

```

### Output

If the input is the strings “firststring” and “secondstring”, then the expected output is:  
“firststring” is less than “secondstring”

### Related Information

- “string.h” on page 46
- “setlocale() — Set Locale” on page 1241
- “strcmp() — Compare Strings” on page 1418
- “strncmp() — Compare Strings” on page 1440

## strcpy() — Copy String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <string.h>
```

```
char *strcpy(char *string1, const char *string2);
```

### General Description

The `strcpy()` built-in function copies *string2*, including the ending null character, to the location specified by *string1*. The *string2* argument to `strcpy()` must contain a null character (`\0`) marking the end of the string. You cannot use a literal string for a *string1* value, although *string2* may be a literal string. If the two objects overlap, the behavior is undefined.

### Returned Value

Returns the value of *string1*.

### Example CBC3BS38

```
/* CBC3BS38
   This example copies the contents of source to destination.
*/
#include <stdio.h>
#include <string.h>

#define SIZE    40

int main(void)
{
    char source[ SIZE ] = "This is the source string";
    char destination[ SIZE ] = "And this is the destination string";
    char * return_string;

    printf( "destination is originally = \"%s\\n\"", destination );
    return_string = strcpy( destination, source );
    printf( "After strcpy, destination becomes \"%s\\n\"", destination );
}
```

### Output

```
destination is originally = "And this is the destination string"
After strcpy, destination becomes "This is the source string"
```

### Related Information

- “string.h” on page 46
- “memcpy() — Copy Buffer” on page 813
- “strcat() — Concatenate Strings” on page 1414
- “strchr() — Search for Character” on page 1416
- “strcmp() — Compare Strings” on page 1418

- “strcspn() — Compare Strings” on page 1424
- “strncpy() — Copy String” on page 1442
- “strpbrk() — Find Characters in String” on page 1444
- “strrchr() — Find Last Occurrence of Character in String” on page 1449
- “strspn() — Search String” on page 1451

## strcspn() — Compare Strings

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <string.h>
```

```
size_t strcspn(const char *string1, const char *string2);
```

### General Description

Computes the length of the initial portion of the string pointed to by *string1* that contains no characters from the string pointed to by *string2*.

### Returned Value

Returns the calculated length of the initial portion found.

### Example

#### CBC3BS39

```
/* CBC3BS39
   This example uses strcspn() to find the first occurrence of any of the
   characters a, x, l or e in string.
*/
#include <stdio.h>
#include <string.h>

#define SIZE    40

int main(void)
{
    char string[ SIZE ] = "This is the source string";
    char * substring = "axle";

    printf( "The first %i characters in the string \"%s\"\\
are not in the " "string \"%s\" \\n",
           strcspn(string, substring), string, substring);
}
```

### Output

The first 10 characters in the string "This is the source string" are not in the string "axle"

### Related Information

- “string.h” on page 46
- “strcat() — Concatenate Strings” on page 1414
- “strchr() — Search for Character” on page 1416
- “strcmp() — Compare Strings” on page 1418
- “strcpy() — Copy String” on page 1422
- “strncmp() — Compare Strings” on page 1440
- “strpbrk() — Find Characters in String” on page 1444
- “strrchr() — Find Last Occurrence of Character in String” on page 1449

- “strspn() — Search String” on page 1451

## strdup() — Duplicate a String

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <string.h>
```

```
char *strdup(const char *string);
```

### General Description

The `strdup()` function creates a duplicate of the string pointed to by *string*.

### Returned Value

If successful, the `strdup()` function returns a pointer to a new string which is a duplicate of *string* and returns a null pointer otherwise.

The caller of `strdup()` should free the storage obtained for the string.

ENOMEM    Insufficient storage space is available.

### Related Information

- “string.h” on page 46
- “malloc() — Reserve Storage Block” on page 786
- “free() — Free a Block of Storage” on page 458



## strerror() — Get Pointer to Runtime Error Message

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <string.h>
```

```
char *strerror(int errnum);
```

### General Description

Maps the error number in *errnum* to an error message string. The *errnum* must be a valid *errno* value.

There is an environment variable `_EDC_ADD_ERRNO2`, which when set to 1, will append the current *errno2* value to the end of the `strerror()` string as shown.

EDC5121I Invalid argument. (*errno2*=0x0C0F8402)

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

Returns a pointer to the string, which may be overwritten by a subsequent call to `strerror()`. Do not allow the content of this string to be modified by the program.

### Example

```
/* This example opens a file and prints a runtime error message if an
   error occurs.
   */
#include <stdio.h>
#include <string.h>
#include <errno.h>

int main(void)
{
    FILE *stream;
    :
    if ((stream = fopen("myfile.dat", "r")) == NULL)
        printf(" %s \n", strerror(errno));
}
```

### Related Information

- “string.h” on page 46
- “clearerr() — Reset Error and End-of-File” on page 187
- “ferror() — Test for Read/Write Errors” on page 367
- “perror() — Print Error Message” on page 903

## strfmon() — Convert Monetary Value to String

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#include <monetary.h>
```

```
int strfmon(char *s, size_t maxsize, const char *format, ...);
```

### General Description

strfmon() produces a formatted monetary output string from a double argument. It has been extended to determine floating-point argument format (hexadecimal floating-point or IEEE floating-point) using the `__isBFP()` function.

**Note:** In IEEE floating-point mode, denormal, infinity and NaN argument values are out of range.

Places characters into the array pointed to by *s* as controlled by the string pointed to by *format*. No more than *maxsize* characters are placed into the array.

The character string *format* contains two types of objects: plain characters, which are copied to the output array, and directives, each of which results in the fetching of zero or more arguments that are converted and formatted. The results are undefined if there are insufficient arguments for the *format*. If the *format* is exhausted while arguments remain, the excess arguments are simply ignored. If objects pointed to by *s* and *format* overlap, the behavior is undefined.

The directive (conversion specification) consists of the following sequence:

1. A % character
2. Optional flags: =f, ^, !, then +, C, or (
3. Optional field width (may be preceded by w
4. Optional left precision: #n
5. Optional right precision: .p
6. Required conversion character to indicate what conversion should be performed: i or n

Each directive is replaced by the appropriate characters, as described in the following list:

- |    |   |
|----|---|
| %i | The double argument is formatted according to the locale's international currency format (for example, in USA: USD 1,234.56). An @euro codeset modifier can be used to request "EUR" instead of a national 4-character monetary string. |
| %n | The double argument is formatted according to the locale's national currency format (for example, in USA: \$1,234.56). An @euro codeset modifier can be used to get <euro-sign> instead of <currency>.                                  |

%% is replaced by %. No argument is converted.

The following optional conversion specifications may immediately follow the initial % of a directive:

<code>=f</code>	A flag, used in conjunction with the maximum digits specification <code>#n</code> (see below), specifies that the character <i>f</i> should be used as the numeric fill character. The default numeric fill character is the space character. This option does not affect the other fill operations that always use space as the fill character.
<code>^</code>	A flag. Do not format the currency amount with thousands grouping characters. The default is to insert the grouping characters if defined for the current locale.  <b>Note:</b> The code point for the <code>^</code> character will be determined according to the current LC_SYNTAX category.
<code>+   C   (</code>	A flag, specifies the style of representing positive and negative currency amounts. Only one of <code>+</code> , <code>C</code> , or <code>(</code> may be specified. If <code>+</code> is specified, the locale's equivalent of <code>+</code> and <code>-</code> are used (for example, in USA: the empty (null) string if positive and <code>-</code> if negative). If <code>C</code> is specified, the locale's equivalent of DB for negative and CR for positive are used. If <code>(</code> is specified, the locale's equivalent of enclosing negative amounts within parentheses is used. If this option is not included, a default specified by the current locale is used.
<code>[-]w</code>	The field width. The decimal digit string <i>w</i> specifies a minimum field width in which the result of the conversion is right-justified (or left-justified if the optional flag <code>"-"</code> is specified).
<code>#n</code>	The left precision. The decimal digit string <i>n</i> specifies the maximum number of digits expected to be formatted to the left of the radix character. This option can be used to keep the formatted output from multiple calls to the <code>strfmon()</code> aligned in the same columns. It can also be used to fill unused positions with a special character as in <code>\$***123.45</code> . This option causes an amount to be formatted as if it has the number of digits specified by <i>n</i> . If more digit positions are required than the number specified, conversion specification is ignored. Digit positions in excess of those actually required are filled with the numeric fill character. (See the <code>=f</code> specification above.)

If the thousands grouping is enabled, the behavior is:

1. Format the number as if it is an *n* digit number.
2. Insert fill characters to the left of the leftmost digit (for example, `$0001234.56` or `$***1234.56`)
3. Insert the separator character (for example, `$0,001,234.56` or `$*,**1,234.56`)
4. If the fill character is not the digit zero, the separators are replaced by the fill character (for example, `$****1,234.56`).

To ensure alignment, any characters appearing before or after the number in the formatted output such as currency or sign symbols are padded as necessary with space characters to make their positive and negative formats an equal length.

**Note:** The code point for the `#` character (in `#n`) will be determined according to the current LC\_SYNTAX category.

**.p** The right precision. The decimal digit string *p* specifies the number of digits after the radix character. If the value of the precision *p* is zero, no radix character appears. If this option is not included, a default specified by the current locale is used. The amount being formatted is rounded to the specified number of digits prior to formatting.

**!** A flag used to suppress the currency symbol from the output conversion.

**Note:** The code point for the **!** character is determined according to the current LC\_SYNTAX category.

The LC\_MONETARY category of the program's locale affects the behavior of this function including the monetary radix character (which is different from the numeric radix character affected by the LC\_NUMERIC category), the thousands (or alternative grouping) separator, the currency symbols and formats. The international currency symbol must be in accordance with those specified in ISO 4217 Codes for the representation of currencies and funds.

### Returned Value

If the total number of resulting bytes including the terminating null character is not more than *maxsize*, the `strfmon()` function returns the number of bytes placed into the array pointed to by *\*s*, not including the terminating null character. Otherwise, -1 is returned, the contents of the array are indeterminate and `errno` is set to indicate the error.

E2BIG Conversion stopped due to lack of space in the buffer

### Example CBC3BS41

```
/* CBC3BS41 */
#include <localdef.h>
#include <monetary.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char    string[100];    /* hold the string returned from strfmon() */
    double money = 1234.56;

    if (setlocale(LC_ALL, "En_US") == NULL) {
        printf("Unable to setlocale().\n");
        exit(1);
    }

    strfmon(string, 100, "%i", money);
    printf("%s\n", string);
    strfmon(string, 100, "%n", money);
    printf("%s\n", string);
}
```

### Example

The following example shows euro currency support:

```

/* EUROSAMP
   This example sets the locale to Fr_BE.IBM-1148
   and Fr_BE.IBM-1148@euro and prints a value with
   the locales national and international currency
   format.
*/

#include <locale.h>
#include <locale.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    char string[100];
    double money = 1234.56;

    if (setlocale(LC_ALL,"Fr_BE.IBM-1148") == NULL) {
        printf("Unable to setlocale().\n");
        exit(1);
    }

    strfmon(string,100,"%i",money);
    printf("%s\n",string);
    strfmon(string,100,"%n",money);
    printf("%s\n",string);

    if (setlocale(LC_ALL,"Fr_BE.IBM-1148@euro") == NULL) {
        printf("Unable to setlocale().\n");
        exit(1);
    }

    strfmon(string,100,"%i",money);
    printf("%s\n",string);
    strfmon(string,100,"%n",money);
    printf("%s\n",string);
}

```

### Output

```

1.234,56 BEF
1.234,56 BF
1.234,56 EUR
1.234,56 <euro-sign>

```

### Related Information

- “monetary.h” on page 36
- “\_\_isBFP() — Determine Application Floating-Point Format” on page 705

## strptime() — Convert to Formatted Time

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <time.h>
```

```
size_t strptime(char *dest, size_t maxsize,  
                const char *format, const struct tm *timeptr);
```

### General Description

Places characters into the array pointed to by *dest* according to the string pointed to by *format*. The format string is a multibyte character string which contains:

- Conversion specification characters
- Ordinary multibyte characters, which are copied into an array unchanged.

The characters that are converted are determined by the LC\_CTYPE category of the current locale and by the values in the time structure pointed to by *timeptr*. The time structure pointed to by *timeptr* is usually obtained by calling the `gmtime()` or `localtime()` function.

Table 36 (Page 1 of 3). Conversion Specifiers Used by `strptime()`

Specifier	Meaning
%a	Replace with abbreviated weekday name of locale.
%A	Replace with full weekday name of locale.
%b	Replace with abbreviated month name of locale.
%B	Replace with full month name of locale.
%c	Replace with date and time of locale.
%C	Replace with locale's century number (year divided by 100 and truncated).
%d	Replace with day of the month (01-31).
%D	Insert date in mm/dd/yy form, regardless of locale.
%e	Insert month of the year as a decimal number (01–12). Under C POSIX only, it's a 2-character, right-justified, blank-filled field.
%E[cCxY]	If the alternative date/time format is not available, the %E descriptors are mapped to their unextended counterparts. For example, %EC is mapped to %C
%Ec	Replace with the locale's alternative date and time representation.
%EC	Replace with the name of the base year (period) in the locale's alternative representation.
%Ex	Replace with the locale's alternative date representation.
%EX	Replace with the locale's alternative time representation.
%Ey	Replace with the offset from %EC (year only) in the locale's alternative representation.
%EY	Replace with the full alternative year representation.

Table 36 (Page 2 of 3). Conversion Specifiers Used by `strptime()`

Specifier	Meaning
%h	Replace with locale's abbreviated month name. This is the same as %b. %b.
%H	Replace with hour (24-hour clock) as a decimal number (00-23).
%I	Replace with hour (12-hour clock) as a decimal number (01-12).
%j	Replace with day of the year (001-366).
%m	Replace with month (01-12).
%M	Replace with minute (00-59).
%n	Replace with a new line.
%O[deHImMSUwWy]	If the alternative date/time format is not available, the %E descriptors are mapped to their unextended counterparts. For example, %Od is mapped to %d.
%Od	Replace with the day of month, using the locale's alternative numeric symbols, filled as needed with leading zeroes if there is any alternative symbol for zero, otherwise with leading spaces.
%Oe	Replace with the day of the month, using the locale's alternative symbols, filled as needed with leading spaces.
%OH	Replace with the hour (24-hour clock) using the locale's alternative symbols.
%OI	Replace with the hour (12-hour clock) using the locale's alternative symbols.
%Om	Replace with the month using the locale's alternative numeric symbols.
%OM	Replace with the minutes using the locale's alternative numeric symbols.
%OS	Replace with the seconds using the locale's alternative numeric symbols.
%OU	Replace with the week number of the year (Sunday as the first day of the week, rules corresponding to %U) using the locale's alternative numeric symbols.
%Ow	Replace with the weekday (Sunday=0) using the locale's alternative numeric symbols.
%OW	Replace with the week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.
%Oy	Replace with the year (offset from %C) in the locale's alternative representation and using the locale's alternative numeric symbols.
%p	Replace with the locale's equivalent of AM or PM.
%r	Replace with a string equivalent to %l:%M:%S %p; or use <code>t_fmt_ampm</code> from <code>LC_TIME</code> , if present
%R	Replace with time in 24 hour notation (%H:%M)
%S	Replace with second (00-61).
%t	Replace with a tab.
%T	Replace with a string equivalent to %H:%M:%S.
%u	Replace with the weekday as a decimal number (1 to 7), with 1 representing Monday.
%U	Replace with week number of the year (00-53) where Sunday is the first day of the week.
%V	Replace with week number of the year (01-53) where Monday is the first day of the week.
%w	Replace with weekday (0-6) where Sunday is 0.
%W	Replace with week number of the year (00-53) where Monday is the first day of the week.
%x	Replace with date representation of locale.

Table 36 (Page 3 of 3). Conversion Specifiers Used by `strptime()`

Specifier	Meaning
%X	Replace with time representation of locale.
%y	Replace with year without the century (00-99).
%Y	Replace with year with century.
%Z	Replace with name of time zone, or no characters if time zone is not available.
%%	Replace with %.

If data has the form of a directive, but is not one of the above, the characters following the % are copied to the output.

The behavior is undefined when objects being copied overlap. *maxsize* specifies the maximum number of characters that can be copied into the array.

If `strptime()` is called by a non-POSIX application, it obtains appropriate time-zone name information from LC\_TOD locale category. Time zone name defaults to STD for Standard time name, DST for Daylight Savings time name, or UTC for Coordinated Universal Time, name, as appropriate, if time-zone name information is unspecified in the current LC\_TOD locale category.

**Note** This function is sensitive to time zone information which is provided by:

- The TZ environmental variable when POSIX(ON) and TZ is correctly defined, or by the \_TZ environmental variable when POSIX(OFF) and \_TZ is correctly defined.
- The LC\_TOD category of the current locale if POSIX(OFF) or TZ is not defined.

The time zone external variables `tzname`, `timezone`, and `daylight` declarations remain feature test protected in `time.h`.

#### Note

The `strptime()` function requires time-zone name information to convert the %Z conversion specifier. It is obtained as follows.

If `strptime()` is called by a OS/390 C application running POSIX(ON), it calls the `tzset()` function to obtain the time-zone name from the current LC\_TOD locale category, by parsing the TZ environment variable. If the `tm` structure input to `strptime()` was produced by calling `localtime()`, `strptime()` converts %Z to the Standard or Daylight Savings name characters specified by the TZ environment variable or LC\_TOD category (if TZ cannot be found or parsed).

The `tm_isdst` flag in the time structure input to `strptime()` determines whether %Z is replaced by the Standard or Daylight Savings name characters. If Standard or Daylight Savings name characters are not available in the current LC\_TOD locale category or from parsing TZ, `strptime()` uses the characters STD for Standard or DST for Daylight Savings time name.

If the `tm` structure input to `strptime()` was produced by the `gmtime()` function, `strptime()` replaces %Z by UCTNAME characters specified in the current LC\_TOD locale category or by UTC if UCTNAME is not specified.



## Returned Value

Returns the number of characters (bytes) placed into the array, not including the terminating null character. On error, the value 0 is returned, and the content of the string is indeterminate.

## Example CBC3BS42

```
/* CBC3BS42
   This example places characters into the array dest and prints the
   resulting string.
*/
#include <stdio.h>
#include <time.h>

int main(void)
{
    char dest[70];
    int ch;
    time_t temp;
    struct tm *timeptr;

    temp = time(NULL);
    timeptr = localtime(&temp);
    ch = strftime(dest, sizeof(dest)-1, "Today is %A, "
                  " %b %d. \n Time: %I:%M %p", timeptr);
    printf("%d characters placed in string to make: \n \n %s", ch, dest);
}
```

## Output

44 characters placed in string to make:

```
Today is Friday, Jun 16.
Time: 03:07 PM
```

## Related Information

- “time.h” on page 51
- “asctime() — Convert Time to Character String” on page 106
- “ctime() — Convert Time to a Character String” on page 250
- “gmtime() — Convert Time to Broken-Down UTC Time” on page 640
- “localdtconv() — Date/Time Formatting Convention Inquiry” on page 754
- “localtime() — Convert Time and Correct for Local Time” on page 758
- “mktime() — Convert Local Time” on page 828
- “time() — Determine Current Time” on page 1570
- “tzset() — Set the Time Zone” on page 1637

## strlen() — Determine String Length

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <string.h>
```

```
size_t strlen(const char *string);
```

### General Description

The `strlen()` built-in function determines the length of string pointed to by *string*, excluding the terminating null character.

### Returned Value

Returns the length of *string*.

### Example

#### CBC3BS43

```
/* CBC3BS43
   This example determines the length of the string that is passed to main.
*/
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
    if ( argc != 2 )
        printf( "Usage: %s string\n", argv[0] );
    else
        printf( "Input string has a length of %i\n", strlen( argv[1] ) );
}
```

### Output

If the input is the string: "How long is this string?", then the expected output is:

Input string has a length of 24

### Related Information

- “string.h” on page 46
- “mblen() — Calculate Length of Multibyte Character” on page 791
- “strncat() — Concatenate Strings” on page 1438
- “strncmp() — Compare Strings” on page 1440
- “strncpy() — Copy String” on page 1442
- “wcslen() — Calculate Length of Wide-Character String” on page 1715

## strncasecmp() — Case-insensitive String Comparison

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <strings.h>
```

```
int strncasecmp(const char *string1, const char *string2, size_t n);
```

### General Description

The `strncasecmp()` function compares, while ignoring differences in case, the string pointed to by *string1* to the string pointed to by *string2*. At most *n* characters will be compared.

The string arguments to the function should contain a null character (`\0`) marking the end of the string.

The `strncasecmp()` function is locale-sensitive.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

The `strncasecmp()` function returns a value indicating the relationship between the strings, while ignoring case, as follows:

Value	Meaning
< 0	String pointed to by <i>string1</i> is less than string pointed to by <i>string2</i> .
= 0	String pointed to by <i>string1</i> is equal to string pointed to by <i>string2</i> .
> 0	String pointed to by <i>string1</i> is greater than string pointed to by <i>string2</i> .

There are no `errno` values defined for `strncasecmp()`.

### Related Information

- “strings.h” on page 46
- “strcasecmp() — Case-insensitive String Comparison” on page 1413
- “strcspn() — Compare Strings” on page 1424
- “strncmp() — Compare Strings” on page 1440

## strncat() — Concatenate Strings

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <string.h>
```

```
char *strncat(char *string1, const char *string2 size_t count);
```

### General Description

The `strncat()` built-in function appends the first *count* characters of *string2* to *string1* and ends the resulting string with a null character (`\0`). If *count* is greater than the length of *string2*, `strncat()` appends only the maximum length of *string2* to *string1*. The first character of the appended string overwrites the terminating null character of the string pointed to by *string1*.

If copying takes place between overlapping objects, the behavior is undefined.

### Returned Value

Returns the value *string1*, the concatenated string.

### Example

#### CBC3BS44

```
/* CBC3BS44
   This example demonstrates the difference between strcat() and strncat().
   strcat() appends the entire second string to the first, whereas
   strncat() appends only the specified number of characters in the second
   string to the first.
*/
#include <stdio.h>
#include <string.h>

#define SIZE 40

int main(void)
{
    char buffer1[SIZE] = "computer";
    char * ptr;

    /* Call strcat with buffer1 and " program" */

    ptr = strcat( buffer1, " program" );
    printf( "strcat : buffer1 = \"%s\\n\"", buffer1 );

    /* Reset buffer1 to contain just the string "computer" again */

    memset( buffer1, '\\0', sizeof( buffer1 ));
    ptr = strcpy( buffer1, "computer" );

    /* Call strncat() with buffer1 and " program" */
    ptr = strncat( buffer1, " program", 3 );
```

```
    printf( "strncat: buffer1 = \"%s\\n", buffer1 );  
}
```

### Output

```
strcat : buffer1 = "computer program"  
strncat: buffer1 = "computer pr"
```

### Related Information

- “string.h” on page 46
- “strcat() — Concatenate Strings” on page 1414
- “strncmp() — Compare Strings” on page 1440
- “strncpy() — Copy String” on page 1442
- “strpbrk() — Find Characters in String” on page 1444
- “strrchr() — Find Last Occurrence of Character in String” on page 1449
- “strspn() — Search String” on page 1451

## strncmp() — Compare Strings

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <string.h>
```

```
int strncmp(const char *string1, const char *string2, size_t count);
```

### General Description

The `strncmp()` built-in function compares at most the first *count* characters of the string pointed to by *string1* to the string pointed to by *string2*.

The string arguments to the function should contain a null character (`\0`) marking the end of the string.

The relation between the strings is determined by the sign of the difference between the values of the leftmost first pair of characters that differ. The values depend on character encoding. This function is *not* locale sensitive.

### Returned Value

Returns a value indicating the relationship between the substrings, as follows:

Value	Meaning
Less than 0	String pointed to by <i>substring1</i> less than string pointed to by <i>substring2</i>
0	String pointed to by <i>substring1</i> equivalent to string pointed to by <i>substring2</i>
Greater than 0	String pointed to by <i>substring1</i> greater than string pointed to by <i>substring2</i>

### Example

#### CBC3BS45

```
/* CBC3BS45
   This example demonstrates the difference between strcmp() and strncmp().
*/
#include <stdio.h>
#include <string.h>

#define SIZE 10

int index = 3;

int main(void)
{
    int result;
```

```

char buffer1[SIZE] = "abcdefg";
char buffer2[SIZE] = "abcfg";
void print_result( int, char *, char * );

result = strcmp( buffer1, buffer2 );
printf( "  strcmp: compares each character\n");
print_result( result, buffer1, buffer2 );

result = strncmp( buffer1, buffer2, index);
printf( "\nstrncmp: compares only the first %i characters\n", index );
print_result( result, buffer1, buffer2 );
}

void print_result( int res, char * p_buffer1, char * p_buffer2 )
{
    if ( res == 0 )
        printf( "first %i characters of \"%s\" is identical to \"%s\"%bsl.n",
                index, p_buffer1, p_buffer2);
    else if ( res < 0 )
        printf( "\"%s\" is less than \"%s\"\\n", p_buffer1, p_buffer2 );
    else
        printf( "\"%s\" is greater than \"%s\"\\n", p_buffer1, p_buffer2 );
}

```

## Output

strcmp: compares each character  
 "abcdefg" is less than "abcfg"

strncmp: compares only the first 3 characters  
 first 3 characters of "abcdefg" is identical to "abcfg"

## Related Information

- “memcmp() — Compare Bytes” on page 811
- “strcmp() — Compare Strings” on page 1418
- “strcspn() — Compare Strings” on page 1424
- “strncat() — Concatenate Strings” on page 1438
- “strncpy() — Copy String” on page 1442
- “strpbrk() — Find Characters in String” on page 1444
- “strrchr() — Find Last Occurrence of Character in String” on page 1449
- “strspn() — Search String” on page 1451

## strncpy() — Copy String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <string.h>
```

```
char *strncpy(char *string1, const char *string2 size_t count);
```

### General Description

The `strncpy()` built-in function copies at most *count* characters of *string2* to *string1*. If *count* is less than or equal to the length of *string2*, a null character (`\0`) is *not* appended to the copied string. If *count* is greater than the length of *string2*, the *string1* result is padded with null characters (`\0`) up to length *count*.

If copying takes place between objects that overlap, the behavior is undefined.

### Returned Value

Returns *string1*.

### Example CBC3BS46

```
/* CBC3BS46
   This example demonstrates the difference between strcpy() and strncpy().
*/
#include <stdio.h>
#include <string.h>

#define SIZE    40

int main(void)
{
    char source[ SIZE ] = "123456789";
    char source1[ SIZE ] = "123456789";
    char destination[ SIZE ] = "abcdefg";
    char destination1[ SIZE ] = "abcdefg";
    char * return_string;
    int    index = 5;

    /* This is how strcpy works */
    printf( "destination is originally = '%s'\n", destination );
    return_string = strcpy( destination, source );
    printf( "After strcpy, destination becomes '%s'\n\n",

    /* This is how strncpy works */
    printf( "destination1 is originally = '%s'\n", destination );
    return_string = strncpy( destination1, source1, index );
    printf( "After strncpy, destination1 becomes '%s'\n", destination );
}
```

### Output



destination is originally = 'abcdefg'  
After strcpy, destination becomes '123456789'

destination1 is originally = 'abcdefg'  
After strncpy, destination1 becomes '12345fg'

### **Related Information**

- “string.h” on page 46
- “memcpy() — Copy Buffer” on page 813
- “strcpy() — Copy String” on page 1422
- “strncat() — Concatenate Strings” on page 1438
- “strncmp() — Compare Strings” on page 1440
- “strpbrk() — Find Characters in String” on page 1444
- “strrchr() — Find Last Occurrence of Character in String” on page 1449
- “strspn() — Search String” on page 1451

## strpbrk() — Find Characters in String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <string.h>
```

```
char *strpbrk(const char *string1, const char *string2);
```

### General Description

Locates the first occurrence in the string pointed to by *string1* of any character from the string pointed to by *string2*.

### Returned Value

Returns a pointer to the character. If *string1* and *string2* have no characters in common, a NULL pointer is returned.

### Example

#### CBC3BS47

```
/* CBC3BS47
   This example returns a pointer to the first occurrence in the array
   string of either a or b.
*/
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *result, *string = "A Blue Danube";
    char *chars = "ab";

    result = strpbrk(string, chars);
    printf("The first occurrence of any of the characters \"%s\" in "
           "\"%s\" is \"%s\"\\n", chars, string, result);
}
```

### Output

The first occurrence of any of the characters "ab" in "A Blue Danube" is "anube"

### Related Information

- “string.h” on page 46
- “strchr() — Search for Character” on page 1416
- “strcspn() — Compare Strings” on page 1424
- “strncmp() — Compare Strings” on page 1440
- “strrchr() — Find Last Occurrence of Character in String” on page 1449
- “strspn() — Search String” on page 1451

## strptime() — Date and Time Conversion

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <time.h>
```

```
char *strptime(const char *buf, const char *fmt, struct tm *tm);
```

### General Description

Converts the character string pointed to by *buf* to values that are stored in the *tm* structure pointed to by *tm*, using the format specified by *fmt*.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

The *\*fmt* is composed of zero or more directives. Each directive is composed of one of the following: one or more white-space characters (as specified by the `isalnum()` to `isxdigit()` function); an ordinary character (neither % nor a white-space character); or a conversion specification. Each conversion specification is composed of a % character followed by a conversion character that specifies the replacement required. There must be white-space or other non-alphanumeric characters between any two conversion specifications.

Table 37. Conversion Specifiers Used by strptime()

Specifier	Meaning
%a	Day of week, using locale's abbreviated or full weekday name.
%A	Day of week, using locale's abbreviated or full weekday name.
%b	Month, using locale's abbreviated or full month name.
%B	Month, using locale's abbreviated or full month name.
%c	Date and time, using locale's date and time.
%C	Century number (year divided by 100 and truncated to an integer)
%d	Day of the month (1-31; leading zeros permitted but not required).
%D	Date as %m/%d/%y.
%e	Day of the month (1-31; leading zeros permitted but not required).
%h	Month, using locale's abbreviated or full month name.
%H	Hour (0-23; leading zeros permitted but not required).
%I	Hour (0-12; leading zeros permitted but not required).
%j	Day number of the year (001-366).
%m	Month number (1-12; leading zeros are permitted but not required).
%M	Minute (0-59; leading zeros are permitted but not required).
%n	New-line character.
%p	Locale's equivalent of AM or PM.
%r	Time as %I:%M:%S a.m. or %I:%M:%S p.m.
%R	Time in 24 hour notation (%H%M).
%S	Seconds (0-61; leading zeros are permitted but not required).
%t	Tab character.
%T	Time as %H:%M:%S.
%U	Week number of the year (0-53; where Sunday is the first day of the week; leading zeros are permitted but not required).
%w	Weekday (0-6; where Sunday is 0; leading zeros are permitted but not required).
%W	Week number of the year (0-53; where Monday is the first day of the week; leading zeros are permitted but not required).
%x	Date, using locale's date format.
%X	Time, using locale's time format.
%y	Year within century. When a century is not otherwise specified, values in the range 69-99 refer to years in the twentieth century (1969 to 1999 inclusive); values in the range 00-68 refer to years in the twenty-first century (2000 to 2068 inclusive). Leading zeros are permitted but not required.
%Y	Year, including century.
%Z	Time zone name.
%%	Replace with %.

**Modified Directives**

Some directives can be modified by the E or O modifier characters to indicate that an alternative format or specification should be used rather than the one normally used by the unmodified directive. If the alternative format or specification does not exist in the current locale, the behavior will be as if the unmodified directive were used.

<i>Table 38. Modified Directives Used by strptime()</i>	
<b>Specifier</b>	<b>Meaning</b>
%Ec	Replace with the locale's alternative date and time representation.
%EC	Replace with the name of the base year (period) in the locale's representation.
%Ex	Replace with the locale's alternative date representation.
%EX	Replace with the locale's alternative time representation.
%Ey	Replace with the offset from %EC (year only) in the locale's representation
%EY	Replace with the full alternative year representation.
%Od	Replace with the day of month, using the locale's alternative numeric symbols, filled as needed with leading zeroes if there is any alternative symbol for zero, otherwise with leading spaces.
%Oe	Replace with the day of the month, using the locale's alternative numeric symbols, filled as needed with leading spaces.
%OH	Replace with the hour (24-hour clock) using the locale's alternative numeric symbols.
%OI	Replace with the hour (12-hour clock) using the locale's alternative numeric symbols.
%Om	Replace with the month using the locale's alternative numeric symbols.
%OM	Replace with the minutes using the locale's alternative numeric symbols.
%OS	Replace with the seconds using the locale's alternative numeric symbols.
%OU	Replace with the week number of the year (Sunday as the first rules corresponding to %U) using the locale's alternative numeric symbols.
%Ow	Replace with the weekday (Sunday=0) using the locale's alternative numeric symbols.
%OW	Replace with the week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.
%Oy	Replace with the year (offset from %C) in the locale's alternative representation and using the locale's alternative numeric symbols.

A directive composed of white-space characters is executed by scanning input up to the first character that is not white space (which remains unscanned) or until no more characters can be scanned.

A directive that is an ordinary character is executed by scanning the next character from the buffer. If the character scanned from the buffer differs from the one comprising the directive, the directive fails, and the differing and subsequent characters remain unscanned.

A series of directives composed or %n, %t, white-space characters or any combination is executed by scanning up to the first character that is not white space (which remains unscanned), or until no more characters can be scanned.

Any other conversion specification is executed by scanning characters until a character matching the next directive is scanned, or until no more characters can be scanned. These characters, excepting the one matching the next directive, are then

compared to the locale values associated with the conversion specifier. If a match is found, values for the appropriate `tm` structure members are set to values corresponding to the locale information. Case is ignored when matching items in *buf* such as month or weekday names. If no match is found, `strptime()` fails and no more characters are scanned.

## Returned Value

Upon successful completion `strptime()` returns a pointer to the character following the last character parsed. Otherwise, a null pointer is returned.

## Example CBC3BS48

```
/* CBC3BS48 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <localdef.h>

int main(void)
{
    struct tm xmas;

    if (strptime("12/25/93 13:30:00", "%D %T", &xmas) == NULL) {
        printf("strptime() failed.\n");
        exit(1);
    }

    printf("tm_sec   =%3d\n", xmas.tm_sec );
    printf("tm_min   =%3d\n", xmas.tm_min );
    printf("tm_hour  =%3d\n", xmas.tm_hour );
    printf("tm_mday  =%3d\n", xmas.tm_mday );
    printf("tm_mon   =%3d\n", xmas.tm_mon  );
    printf("tm_year  =%3d\n", xmas.tm_year );
    printf("tm_wday  =%3d\n", xmas.tm_wday );
    printf("tm_yday  =%3d\n", xmas.tm_yday );
}
```

## Output

```
tm_sec   = 0
tm_min   = 30
tm_hour  = 13
tm_mday  = 25
tm_mon   = 11
tm_year  = 93
tm_wday  = 0
tm_yday  =358
```

## Related Information

- “time.h” on page 51

strchr() — Find Last Occurrence of Character in String

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

Format

```
#include <string.h>

char *strchr(const char *string, int c);
```

General Description

The strchr() built-in function finds the last occurrence of *c* (converted to a char) in *string*. The ending null character is considered part of the *string*.

Returned Value

Returns a pointer to the last occurrence of *c* in *string*. If the given character is not found, a NULL pointer is returned.

Example  
CBC3BS49

```
/* CBC3BS49
   This example compares the use of strchr() and strrchr().
   It searches the string for the first and last occurrence of p in the string.
*/
#include <stdio.h>
#include <string.h>

#define SIZE 40

int main(void)
{
    char buf[SIZE] = "computer program";
    char * ptr;
    int    ch = 'p';

    ptr = strchr( buf, ch );    /* This illustrates strchr() */
    printf( "The first occurrence of %c in '%s' is '%s'\n", ch, buf, ptr );
    ptr = strrchr( buf, ch );  /* This illustrates strrchr() */
    printf( "The last occurrence of %c in '%s' is '%s'\n", ch, buf, ptr );
}
```

Output

The first occurrence of p in 'computer program' is 'puter program'  
The last occurrence of p in 'computer program' is 'program'

### **Related Information**

- “string.h” on page 46
- “memchr() — Search Buffer” on page 809
- “strchr() — Search for Character” on page 1416
- “strcspn() — Compare Strings” on page 1424
- “strncmp() — Compare Strings” on page 1440
- “strpbrk() — Find Characters in String” on page 1444
- “strspn() — Search String” on page 1451



## strspn() — Search String

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment ISO C XPG4 XPG4.2	both	

### Format

```
#include <string.h>
```

```
size_t strspn(const char *string1, const char *string2);
```

### General Description

Calculates the length of the maximum initial portion of the string pointed to by *string1* that consists entirely of the characters contained in the string pointed to by *string2*.

### Returned Value

Returns the length of the substring found.

### Example

#### CBC3BS50

```
/* CBC3BS50
   This example finds the first occurrence in the array string of a
   character that is neither an a, b, nor c. Because the string in this
   example is cabbage, strspn() returns 5, the length of the segment of
   cabbage before a character that is not an a, b or c.
*/
#include <stdio.h>
#include <string.h>

int main(void)
{
    char * string = "cabbage";
    char * source = "abc";
    int index;

    index = strspn( string, "abc" );
    printf( "The first %d characters of \"%s\" are found in \"%s\"\\n",
           index, string, source );
}
```

### Output

The first 5 characters of "cabbage" are found in "abc"

### Related Information

- “string.h” on page 46
- “strcat() — Concatenate Strings” on page 1414
- “strchr() — Search for Character” on page 1416
- “strcmp() — Compare Strings” on page 1418
- “strcpy() — Copy String” on page 1422
- “strcspn() — Compare Strings” on page 1424

- “strpbrk() — Find Characters in String” on page 1444
- “strrchr() — Find Last Occurrence of Character in String” on page 1449

## strstr() — Locate Substring

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <string.h>
```

```
char *strstr(const char *string1, const char *string2);
```

### General Description

Finds the first occurrence of the string pointed to by *string2* (excluding the null character) in the string pointed to by *string1*.

### Returned Value

Returns a pointer to the beginning of the first occurrence of *string2* in *string1*. If *string2* does not appear in *string1*, *strstr()* returns NULL. If *string2* points to a string with zero length, the function returns *string1*.

### Example

#### CBC3BS51

```
/* CBC3BS51
   This example locates the string haystack in the string "needle in a
   haystack".
*/
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *string1 = "needle in a haystack";
    char *string2 = "haystack";
    char *result;

    result = strstr(string1, string2);
    /* Result = a pointer to "haystack" */
    printf("%s\n", result);
}
```

### Output

```
haystack
```

### Related Information

- “string.h” on page 46
- “strchr() — Search for Character” on page 1416
- “strcmp() — Compare Strings” on page 1418
- “strcspn() — Compare Strings” on page 1424
- “strncmp() — Compare Strings” on page 1440
- “strpbrk() — Find Characters in String” on page 1444
- “strrchr() — Find Last Occurrence of Character in String” on page 1449
- “strspn() — Search String” on page 1451

## strtolcoll() — Return Collating Element for String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C	both	

### Format

```
#include <collate.h>
```

```
collat_t strtolcoll(char *s);
```

### General Description

Determines whether the string pointed to by *s* represents the valid element as defined in the LC\_COLLATE category of the current locale.

If a string pointed to by *s* contains only one character, the collating element representing this character always exists. Otherwise, a valid collating element exists if the LC\_COLLATE category contains the definition of a sequence of characters that collate as one for the purpose of culture-sensitive string comparison. This many-characters-to-one-collating element relation is also called *many-to-one* mapping.

### Returned Value

The type `collat_t` represents the collating elements. If many-to-one mapping is not defined in the LC\_COLLATE of the current locale, then `(collat_t)-1` is returned. Also, if the string is not a valid collating element or is of zero length, `(collat_t)-1` is returned.

### Example CBC3BS52

```
/* CBC3BS52
   This example uses the strtolcoll() function to get the collat_t value
   for the start and end collating-elements for the collrange() function.
*/
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <collate.h>
#include <wchar.h>
#include <wctype.h>

main(int argc, char *argv[]) {
    collat_t s, e, *rp;
    int i;

    setlocale(LC_ALL, "");
    if ((s = strtolcoll(argv[1])) == (collat_t)-1) {
        printf("%s collating element not defined\n", argv[1]);
        exit(1);
    }
    if ((e = strtolcoll(argv[2])) == (collat_t)-1) {
        printf("%s collating element not defined\n", argv[2]);
        exit(1);
    }
    if ((i = collrange(s, e, &rp)) == -1) {
        printf("Invalid range for %s to %s\n", argv[1], argv[2]);
        exit(1);
    }
}
```

```

    }
    for (; i-- > 0; rp++) {
        if (ismccollet(*rp))
            printf("%s ", colltostr(*rp));
        else if (iswprint(*rp))
            printf("%lc ", *rp);
        else
            printf("%x ", *rp);
    }
}

```

## Related Information

- “collate.h” on page 23
- “cclass() — Return Characters in a Character Class” on page 149
- “collequiv() — Return a List of Equivalent Collating Elements” on page 200
- “collorder() — Return List of Collating Elements” on page 202
- “collrange() — Calculate the Range List of Collating Elements” on page 204
- “colltostr() — Return a String for a Collating Element” on page 206
- “getmccoll() — Get Next Collating Element from String” on page 554
- “getwmccoll() — Get Next Collating Element from Wide String” on page 632
- “ismccollet() — Identify a Multi-Character Collating Element” on page 710
- “maxcoll() — Return Maximum Collating Element” on page 788

## strtod() — Convert Character String to Double

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
double strtod(const char *nptr, char **endptr)
```

### General Description

Converts a part of a character string, pointed to by *nptr*, to a double. The parameter *nptr* points to a sequence of characters that can be interpreted as a numerical value of the type *double*.

The double value is hexadecimal floating-point or IEEE floating-point format depending on the floating-point mode of the thread invoking the strtod() function. This function uses \_\_isBFP() to determine the floating-point mode of the invoking thread.

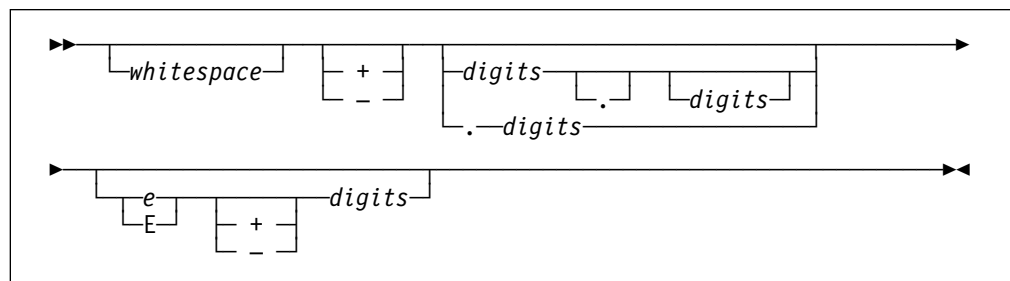
See the “*fscanf* Family of Formatted Input Functions” on page 467 for a description of special infinity and NaN sequences recognized by OS/390 formatted input functions, including atof() and strtod() in IEEE floating-point mode.

To provide an ASCII input/output format for applications using this function, define feature test macro \_\_LIBASCII as described on page 22.

The strtod() function breaks the string into three parts:

1. A sequence of white-space characters (as specified for the current locale, see isspace())
2. The sequence of characters of the floating-point constant format (the *subject string*)
3. A sequence of unrecognized characters (including a null character).

The function then attempts to convert the subject string into the floating-point number. The format of the expected string is as follows:



The subject string is the longest string that matches the expected form.

The pointer to the last string successfully converted is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer. If the subject string is empty or it does not have the expected form, then no conversion is performed. The value of *nptr* is stored in the object pointed to by *endptr*.

## Returned Value

Returns the value of the floating-point number, except when the representation causes an underflow or overflow. In an overflow, it returns `-HUGE_VAL` or `+HUGE_VAL`. In an underflow, it returns the value 0. If no conversion is performed, `strtod()` returns the value 0. In both cases, `errno` is set to `ERANGE`, depending on the base of the value.

## Example CBC3BS53

```
/* CBC3BS53
   This example converts a string to a double value. It prints out the
   converted value and the substring that stopped the conversion.
*/
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *string, *stopstring;
    double x;

    string = "3.1415926This stopped it";
    x = strtod(string, &stopstring);
    printf("string = %s\n", string);
    printf("    strtod = %f\n", x);
    printf("    Stopped scan at %s\n\n", stopstring)

    string = "100ergs";
    x = strtod(string, &stopstring);
    printf("string = \"%s\"\n", string);
    printf("    strtod = %f\n", x);
    printf("    Stopped scan at \"%s\"\n\n", stopstring);
}
```

## Output

```
string = 3.1415926This stopped it
    strtod = 3.141593
    Stopped scan at This stopped it

string = 100ergs
    strtod = 100.000000
    Stopped scan at ergs
```

## Related Information

- “`stdlib.h`” on page 45
- “`atof()` — Convert Character String to Double” on page 121
- “`atoi()` — Convert Character String to Integer” on page 122
- “`atol()` — Convert Character String to Long” on page 123
- “`fscanf()` — `scanf()` — `sscanf()` — Read and Format Data” on page 464
- “`__isBFP()` — Determine Application Floating-Point Format” on page 705
- “`strtol()` — Convert Character String to Long” on page 1460
- “`strtoul()` — Convert String to Unsigned Integer” on page 1462

## strtok() — Tokenize String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <string.h>
```

```
char *strtok(char *string1, const char *string2);
```

### General Description

Breaks a character string, pointed to by *string*, into a sequence of tokens. The tokens are separated from one another by the characters in the string pointed to by *string2*.

The token starts with the first character not in the string pointed to by *string2*. If such a character is not found, there are no tokens in the string. `strtok()` returns a null pointer. The token ends with the first character contained in the string pointed to by *string2*. If such a character is not found, the token ends at the terminating null character. Subsequent calls to `strtok()` will return the null pointer. If such a character *is* found, then it is overwritten by a null character, which terminates the token.

If the next call to `strtok()` specifies a null pointer for *string1*, the tokenization resumes at the first character following the found and overwritten character from the previous call. For example:

```
/* Here are two calls */
strtok(string, " ")
strtok(NULL, " ")
```

```
/* Here is the string they are processing */
      abc defg hij
first call finds  ↑
                  ↑ second call starts
```

### Returned Value

The first time `strtok()` is called, it returns a pointer to the first token in *string1*. In later calls with the same token string, `strtok()` returns a pointer to the next token in the string. A NULL pointer is returned when there are no more tokens. All tokens are null-terminated.

### Example CBC3BS54

```
/* CBC3BS54
Using a loop, this example gathers tokens, separated by blanks or
commas, from a string until no tokens are left.
After processing the tokens (not shown), the example returns the
pointers to the tokens a, string, of, and tokens.
The next call to strtok() returns NULL and the loop ends.
```



```

*/
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *token, *string =
        "a string, of,; ;;;,tokens\0,after null terminator";

    /* the string pointed to by string is broken up into the tokens
       "a string", " of", "; ;;;", and "tokens" ; the null terminator
       (\0) is encountered and execution stops after the token "tokens" */
    token = strtok(string, ",");
    do {
        printf("token: \"%s\"\n", token);
    }
    while (token = strtok(NULL, ",;"));
}

```

### Output

```

token: "a string"
token: " of"
token: " "
token: "tokens"

```

### Related Information

- “string.h” on page 46
- “strcat() — Concatenate Strings” on page 1414
- “strchr() — Search for Character” on page 1416
- “strcmp() — Compare Strings” on page 1418
- “strcpy() — Copy String” on page 1422
- “strcspn() — Compare Strings” on page 1424
- “strspn() — Search String” on page 1451

strtol() — Convert Character String to Long

Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

Format

```
#include <stdlib.h>

long int strtol(const char *nptr, char **endptr, int base);
```

General Description

Converts *string1*, a character string, to a long int value.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

The function decomposes the entire string into three parts:

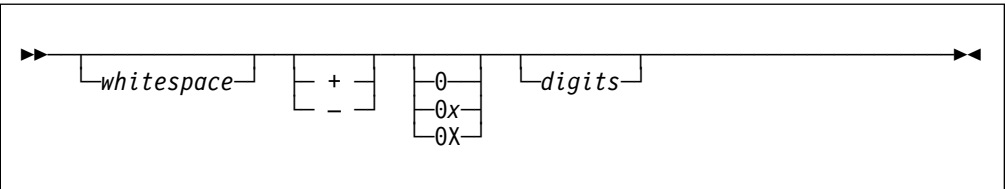
- 1. A sequence of characters, which in the current locale are defined as white-space characters. This part may be empty.
- 2. A sequence of characters interpreted as integer in some base notation. This is the *subject sequence*.
- 3. A sequence of unrecognized characters.

The base notation is determined by *base*, if *base* is greater than zero. If *base* is zero, the base notation is determined by the format of the sequence of characters that follow an optional plus—or optional minus—sign.

- 10            Sequence starts with nonzero decimal digit.
- 8             Sequence starts with 0, followed by a sequence of digits with values from 0 to 7
- 16            Sequence starts with either 0x or 0X, followed by digits, and letters A through F or a through f.

If the base is greater than zero, the subject sequence contains decimal digits and letters, possibly preceded by either a plus or a minus sign. The letters a (or A) through z (or Z) represent values from 10 through 36, but only those letters whose value is less than the value of the base are allowed.

When you use the `strtol()` function, *nptr* should point to a string with the following form:



The pointer to the converted characters, even if conversion was unsuccessful, is stored in the object pointed to by *endptr*, as long as *endptr* is not a null pointer.

## Returned Value

Returns the converted long int value, except when the representation causes an overflow. In the case of an overflow, it returns `LONG_MAX` or `LONG_MIN`, according to the sign of the value and `errno` is set to `ERANGE`. If *base* is not a valid number, or has the value 1, the `strtol()` function returns zero and sets `errno` to `EDOM`. If the string pointed to by *nptr* does not have the expected form, no conversion is performed, zero is returned, and the value of *nptr* is stored in the object pointed to by *endptr* provided that *endptr* is not a `NULL` pointer.

## Example CBC3BS55

```
/* CBC3BS55
   This example converts the strings to a long.
   It prints out the converted value and the substring that stopped the
   conversion.
*/
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    char *string, *stopstring;
    long l;
    int bs;

    string = "10110134932";
    printf("string = %s\n", string);
    for (bs = 2; bs <= 8; bs *= 2)
    {
        l = strtol(string, &stopstring, bs);
        printf("    strtol = %ld (base %d)\n", l, bs);
        printf("    Stopped scan at %s\n\n", stopstring);
    }
}
```

## Output

```
string = 10110134932
    strtol = 45 (base 2)
    Stopped scan at 34932

    strtol = 4423 (base 4)
    Stopped scan at 4932

    strtol = 2134108 (base 8)
    Stopped scan at 932
```

## Related Information

- “`stdlib.h`” on page 45
- “`atof()` — Convert Character String to Double” on page 121
- “`atoi()` — Convert Character String to Integer” on page 122
- “`atol()` — Convert Character String to Long” on page 123
- “`fscanf()` – `scanf()` – `sscanf()` — Read and Format Data” on page 464
- “`strtod()` — Convert Character String to Double” on page 1456
- “`strtoul()` — Convert String to Unsigned Integer” on page 1462

strtoul() — Convert String to Unsigned Integer

Standards

Standards / Extensions	C or C++	Dependencies
Language Environment ISO C XPG4 XPG4.2	both	

Format

```
#include <stdlib.h>

unsigned long int strtoul(const char *string1, char **string2, int base);
```

General Description

Converts *string1*, a character string, to an unsigned long int value.

| To provide an ASCII input/output format for applications using this function, define  
| feature test macro `__LIBASCII` as described on page 22.

The function decomposes the entire string into three parts:

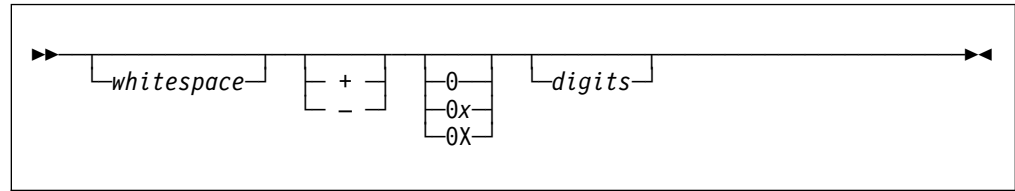
- 1. A sequence of characters, which in the current locale are defined as white-space characters. This part may be empty.
- 2. A sequence of characters interpreted as an unsigned integer in some base notation. This is the *subject sequence*.
- 3. A sequence of unrecognized characters.

The base notation is determined by *base*, if *base* is greater than zero. If *base* is zero, the base notation is determined by the format of the sequence of characters that follow an optional plus or optional minus sign.

- 10           Sequence starts with nonzero decimal digit.
- 8            Sequence starts with 0, followed by a sequence of digits with values from 0 to 7
- 16           Sequence starts with either 0x or 0X, followed by digits, and letters A through F or a through f.

If the base is greater than zero, the subject sequence contains decimal digits and letters, possibly preceded by either a plus or a minus sign. The letters a (or A) through z (or Z) represent values from 10 through 36, but only those letters whose value is less than the value of the base are allowed. The function stops reading the string at the first character that it cannot recognize as part of a number. This character can be the first numeric character greater than or equal to the *base*. The `strtoul()` function sets *string2* to point to the resulting output string if a conversion is performed and provided that *string2* is not a NULL pointer.

When you are using the `strtoul()` function, *string1* should point to a string with the following form:



If *base* is in the range of 2 through 36, it becomes the base of the number. If *base* is 0, the prefix determines the base (8, 16, or 10): the prefix 0 means base 8; the prefix 0x or 0X means base 16; using any other digit without a prefix means decimal.

The pointer to the converted characters, even if conversion was unsuccessful, is stored in the object pointed to by *string2*, as long as *string2* is not a null pointer.

### Returned Value

Returns the converted unsigned long int value, represented in the string, or returns the value 0 if no conversion could be performed. For an overflow, strtoul() returns ULONG\_MAX and sets errno to ERANGE. If *base* is not a valid number, or it has the value 1, the function returns zero and sets errno to EDOM. If the subject sequence starts with a minus sign, the converted value is negated.

### Example CBC3BS56

```
/* CBC3BS56
   This example converts the string to an unsigned long value.
   It prints out the converted value and the substring that stopped the
   conversion.
*/
#include <stdio.h>
#include <stdlib.h>

#define BASE 2

int main(void)
{
    char *string, *stopstring;
    unsigned long ul;

    string = "1000e13 e";
    printf("string = %s\n", string);
    ul = strtoul(string, &stopstring, BASE);
    printf("    strtoul = %ld (base %d)\n", ul, BASE);
    printf("    Stopped scan at %s\n\n", stopstring);
}
```

### Output

```
string = 1000e13 e
    strtoul = 8 (base 2)
    Stopped scan at e13 e
```

## **Related Information**

- “stdlib.h” on page 45
- “atof() — Convert Character String to Double” on page 121
- “atoi() — Convert Character String to Integer” on page 122
- “atol() — Convert Character String to Long” on page 123
- “fscanf() – scanf() – sscanf() — Read and Format Data” on page 464
- “strtod() — Convert Character String to Double” on page 1456
- “strtol() — Convert Character String to Long” on page 1460

## strxfrm() — Transform String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <string.h>
```

```
size_t strxfrm(char *s1, const char *s2, size_t n);
```

### General Description

Transforms the string pointed to by *s2* and places the resulting string into the array pointed to by *s1*. The transformation is determined by the program's locale. The transformed string is not necessarily readable, but can be used with the `strcmp()` or `strncmp()` functions.

The transformation is such that, if `strcmp()` or `strncmp()` were applied to the two transformed strings, the results would be the same as applying the `strcoll()` function to the two corresponding untransformed strings.

No more than *n* bytes are placed into the area pointed to by *s1*, including the terminating null byte. If *n* is zero, *s1* is allowed to be a null pointer.

### Returned Value

Returns the length of the transformed string (excluding the null byte). When *n* is zero and *s1* is a null pointer, the length returned is the number of bytes minus one required to contain the transformed string.

The `strxfrm()` function returns (size)-1 on error, and sets `errno` to indicate the error.

### Notes:

- The string returned by the `strxfrm()` function contains the weights for each order of the characters within the string. As a result, the string returned may be longer than the input string, and does not contain printable characters.
- The `strxfrm()` function issues a `malloc()` when the `LC_COLLATE` category specifies *backward* on the *order\_start* keyword, the *substitute* keyword is specified, or the locale has one-to-many mapping. The `strxfrm()` function will fail if the `malloc()` fails.
- If the locale supports double-byte characters (`MB_CUR_MAX` specified as 4), the `strxfrm()` function validates the multibyte characters, whereas previously the `strxfrm()` function did not validate the string. The `strxfrm()` function will fail if the string contains invalid multibyte characters.
- If `MB_CUR_MAX` is defined as 4, and no collation is defined for DBCS chars in the current locale, the DBCS characters will collate after the single-byte characters.

## Example

### CBC3BS57

```

/* CBC3BS57
   This example prompts the user to input a string of characters, then
   uses strxfrm() to transform the string and return its length.
*/
#include <collate.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(void)
{
    char *string1="string1", *string2="string2";
    char *newstring1, *newstring2;
    int length1, length2;

    length1=strxfrm(NULL, string1, 0);
    length2=strxfrm(NULL, string2, 0);
    if (((newstring1=(char *)calloc(length1+1, 1))==NULL) ||
        ((newstring2=(char *)calloc(length2+1, 1))==NULL))
    {
        printf("insufficient memory\n");
        exit(99);
    }
    if ((strxfrm(newstring1, string1, length1+1) != length1) ||
        (strxfrm(newstring2, string2, length2+1) != length2))
    {
        printf("error in string processing\n");
        exit(99);
    }
    if (strcoll(string1, string2) != strcmp(newstring1, newstring2))
    {
        printf("wrong results\n");
        exit(99);
    }
    printf("correct results\n");
    exit(0);
}

```

## Related Information

- “string.h” on page 46
- “localeconv() — Query Numeric Conventions” on page 756
- “setlocale() — Set Locale” on page 1241
- “strcmp() — Compare Strings” on page 1418
- “strcoll() — Compare Strings” on page 1420
- “strncmp() — Compare Strings” on page 1440



## svc99() — Access Supervisor Call

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

### Format

```
#include <stdio.h>
```

```
int svc99(__S99parms *string);
```

### General Description

Provides access to SVC99 on OS/390, which provides ability to:

- Dynamically allocate or deallocate a resource
- Dynamically concatenate or deconcatenate data sets
- Dynamically retrieve information on data sets

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

The \_\_S99parms structure is defined in `stdio.h`. It has been changed to include the address of the Request Block Extension. The Request Block Extension and the Error Message Parameter list can be used to process the messages returned by SVC99 when an error occurs. To use this feature, you must allocate and initialize these structures.

Table 39. Elements Contained by \_\_S99parms structure

Field	Type	Value Stored
__S99VERB	unsigned char	SVC99 verb code
__S99FLAG1	unsigned short	SVC99 Flags 1 field
__S99FLAG2	unsigned int	SVC99 Flags 2 field for APF authorized programs
__S99ERROR	unsigned short	SVC99 error code field
__S99INFO	unsigned short	SVC99 information code
__S99TXTPP	void *	SVC99 pointer to a list of text unit pointers
__S99RBLN	unsigned char *	SVC99 length of request block
__S99S99X	void *	SVC99 pointer to the Request Extension Block.

## Returned Value

If the input pointer is NULL, svc99() returns 0 if running under CICS and nonzero otherwise. The nonzero value indicates that svc99() is supported under the current operating system (that is, OS/390 non-CICS). If the input is not NULL, svc99() returns -1 if running under CICS (to indicate an error), otherwise it returns a code that results from svc99().

If the input is not NULL, and svc99() is not supported on the system, the function returns the value -1.

## Example CBC3BS58

```
/* CBC3BS58
   This example uses the svc99() function to dynamically allocate a data
   set called USERID.EXAMPLE.
*/
#include <stdio.h>
#include <string.h>

#define MASK 0x80000000

int main(void)
{
    int rc;
    struct __S99struc parmlist;
    char *s[10] = {                /* array of text pointers */
        /* text units follow */
        "\0\x02\0\x01\0\x0E" "USERID.EXAMPLE", /*
DSN=EXAMPLE */
        "\0\x05\0\x01\0\x01\x02",             /* DISP=(,CATLG) */
        "\0\x07\0\0",                          /* SPACE=(TRK,.. */
        "\0\x0A\0\x01\0\x03\0\0\0\x14",        /*
primary=20 */
        "\0\x0B\0\x01\0\x03\0\0\0\x01",        /* secondary=1
*/
        "\0\x15\0\x01\0\x05SYSDA",             /* UNIT=SYSDA */
        "\0\x30\0\x01\0\x02\0\0\x50",          /* BLKSIZE=80
*/
        "\0\x3C\0\x01\0\x02\0\x40\0",         /* DSORG=PS */
        "\0\x42\0\x01\0\x02\0\0\x50",         /* LRECL=80 */
        "\0\x49\0\x01\0\x01\0\x80"};          /* RECFM=F */

    memset(&parmlist, 0, sizeof(parmlist));
    parmlist.__S99RBLN = 20;
    parmlist.__S99VERB = 1;                    /* verb for dsname allocation */
    parmlist.__S99FLAG1 = 0x4000;              /* do not use existing allocation */
    parmlist.__S99XTTPP = s;                   /* pointer to pointer to text units */
    s[9] = (char *)((long unsigned) (s[9]) | MASK);

    rc = svc99(&parmlist);
    if (rc != 0)
        printf(" Error code = %d   Information code = %d\n",
            parmlist.__S99ERROR, parmlist.__S99INFO);
}
```

If your userid starts with one of the letters A-F, you must add two double quotation marks (") before the userid so that the first letter of the userid is interpreted as a character rather than as a hexadecimal digit.

The preceding example can be made more readable by using symbolic names and data structures as demonstrated in the example below. The members IEFZB4DB, IEFZB4D0 and IEFZB4D2 of SYS1.MACLIB contain symbolic names that will be familiar to most assembler language programmers.

This next example uses symbolic names taken from these members to define, in OS/390 C/C++ the text unit representing primary=20 or s[3]. Similar definitions can be made for the remaining text units but will not be given here.

```
#include <stdio.h>
#include <string.h>

#define MASK 0x80000000
#define CHAR_BIT 4
/* Defines one text unit with an integer of size 'bytes' */

#define __S99TUNIT_INT(bytes) struct {
    short unsigned __S99TUKEY;    /* KEY */
    short unsigned __S99TUNUM;    /* NO. OF LENGTH+PARM ENTRIES */
    struct {
        short unsigned __S99TULNG; /* LENGTH OF 1ST (ONLY) PARM */
        unsigned int __S99TUPAR : (bytes) * CHAR_BIT; /* PARAMETER */
    } __S99TUENT;
}

/* initialize by: __S99TUNUM = 1; */
/* __S99TUENT.__S99TULNG = <bytes>; */
/* __S99TUENT.__S99TUPAR = <value>; */

#define __DALPRIME 0x000A /*PRIMARY SPACE QUANTITY */

static const __S99TUNIT_INT(3) primary = {__DALPRIME, 1, 3, 20};

int main(void)
{
    int rc;
    struct __S99struc parmlist;
    memset(&parmlist, 0, sizeof(parmlist));
    void *s[10] = { /* array of text pointers */
        /* text units follow */
        . , /* DSN=EXAMPLE */
        . , /* DISP=(,CATLG)*/
        . , /* SPACE=(TRK,..*/
        &primary, /* primary=20 */
        . , /* secondary=1 */
        . , /* UNIT=SYSDA */
        . , /* BLKSIZE=80 */
        . , /* DSORG=PS */
        . , /* LRECL=80 */
        . }; /* RECFM=F */

    parmlist.__S99RBLN = 20;
    parmlist.__S99VERB = 01; /* verb for dsname allocation */
    parmlist.__S99FLAG1 = 0x4000; /* do not use existing allocation */
    parmlist.__S99XTTP = s; /* pointer to pointer to text units */
    s[9] = (char *)((long unsigned) (s[9]) | MASK);

    rc = svc99(&parmlist);
    if (rc != 0)
        printf(" Error code = %d Information code = %d\n",
            parmlist.__S99ERROR, parmlist.__S99INFO);
}
```

### **Related Information**

- “stdio.h” on page 43
- “dynalloc() — Allocate a Data Set” on page 292
- “dynfree() — Deallocate a Data Set” on page 299
- “dyninit() — Initialize \_\_dyn\_t Structure” on page 301

## swab() — Copy and Swap Bytes

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <unistd.h>
```

```
void swab(const void *src, void *dest, ssize_t nbytes);
```

### General Description

The `swab()` function copies *nbytes* bytes, which are pointed to by *src* to the object pointed to by *dest*, exchanging adjacent bytes. The *nbytes* argument should be even. If *nbytes* is odd, `swab()` copies and exchanges *nbytes*-1 bytes and the disposition of the last byte is left unchanged in the target area. If *nbytes* is zero or negative, no copying is performed.

### Returned Value

The `swab()` function does not have a return value.

### Related Information

None

## swapcontext() — Save and Restore User Context

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON)

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <ucontext.h>
```

```
int swapcontext(ucontext_t *oucp, const ucontext_t *ucp);
```

### General Description

The `swapcontext()` function saves the current user context in the context structure pointed to by *oucp* and restores the user context structure pointed to by *ucp*. `swapcontext()` is equivalent to `getcontext()` with the *oucp* argument followed by `setcontext()` with the *ucp* argument.

Control does not return from the initial invocation of `swapcontext()`. However, if the saved context is not modified using `makecontext()`, and a subsequent `setcontext()` or `swapcontext()` is issued using the saved context, `swapcontext()` returns with a 0 return value.

**Note:** If the *ucontext* pointed to by *ucp* that is input to `swapcontext()`, has not been modified by `makecontext()`, you must ensure that the function that calls `getcontext()` does not return before you call the corresponding `swapcontext()` function. Calling `swapcontext()` after the function calling `getcontext()` returns causes unpredictable program behavior.

This function is supported only in a POSIX program.

The `<ucontext.h>` header file defines the `ucontext_t` type as a structure that includes the following members:

<code>mcontext_t</code>	<code>uc_mcontext</code>	A machine-specific representation of the saved context.
<code>ucontext_t</code>	<code>*uc_link</code>	Pointer to the context that will be resumed when this context returns.
<code>sigset_t</code>	<code>uc_sigmask</code>	The set of signals that are blocked when this context is active.
<code>stack_t</code>	<code>uc_stack</code>	The stack used by this context.

### Special Behavior for C++

If `getcontext()` and `swapcontext()` are used to transfer control in a OS/390 C++ program, the behavior in terms of the destruction of automatic objects is undefined. This applies to both OS/390 C++ and OS/390 C++ ILC modules. The use of `getcontext()` and `swapcontext()` in conjunction with `try()`, `catch()`, and `throw()` is also undefined.

Do not issue `getcontext()` in a C++ constructor or destructor, since the saved context would not be usable in a subsequent `setcontext()` or `swapcontext()` after the constructor or destructor returns.

## Returned Value

If successful, `swapcontext()` does not return from the initial invocation. If the unmodified saved context is later restored, `swapcontext()` returns 0.

If unsuccessful, `swapcontext()` returns -1. There are no defined `errno` values.

## Example

This example uses two contexts. It creates the first, *fcontext*, in main with the *getcontext()* statement, and invokes the function *func*. It invokes the function with the *swapcontext()* statement, saving the context at that point in the second context, *mcontext*. The function returns to the point of the *swapcontext()* using the *setcontext()* statement and specifying *mcontext* as the context.

```
/* This example shows the usage of swapcontext(). */

#define _XOPEN_SOURCE_EXTENDED 1
#include <stdio.h>
#include <ucontext.h>

#define STACK_SIZE 16384

void func(int);

ucontext_t fcontext,mcontext;
int x = 0;

int main(void) {
    int value = 1;

    getcontext(&fcontext);
    if ((fcontext.uc_stack.ss_sp = (char *) malloc(STACK_SIZE)) != NULL) {
        fcontext.uc_stack.ss_size = STACK_SIZE;
        fcontext.uc_stack.ss_flags = 0;
        makecontext(&fcontext,func,1,value);
    }
    else {
        perror("not enough storage for stack");
        abort();
    }
    printf("context has been built\n");
    swapcontext(&mcontext,&fcontext);
    if (!x) {
        perror("incorrect return from swapcontext");
        abort();
    }
    else {
        printf("returned from function\n");
    }
}

void func(int arg) {

    printf("function called with value %d\n",arg);
    x++;
    printf("function returning to main\n");
    setcontext(&mcontext);
}
```

## Output

```
context has been built  
function called with value 1  
function returning to main  
returned from function
```

### Related Information

- “ucontext.h” on page 52
- “getcontext() — Get User Context” on page 505
- “longjmp() — Restore Stack Environment” on page 768
- “\_longjmp() — Non-Local Goto” on page 771
- “makecontext() — Modify User Context” on page 783
- “setcontext() — Restore User Context” on page 1210
- “setjmp() — Preserve Stack Environment” on page 1234
- “\_setjmp() — Set Jump Point for a Non-Local Goto” on page 1237
- “siglongjmp() — Restore the Stack Environment and Signal Mask” on page 1327
- “sigsetjmp() — Save Stack Environment and Signal Mask” on page 1346



## swprintf() — Write Wide Characters to a Wide-Character Array

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

### Format

#### Non-XP4

```
#include <wchar.h>

int swprintf(wchar_t *wcs, size_t n, const wchar_t *format, ...);
```

#### XP4

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

int swprintf(wchar_t *wcs, size_t n, const wchar_t *format, ...);
```

### General Description

The `swprintf()` function is equivalent to `sprintf()`, except for the following:

- The argument *wcs* specifies an array of type `wchar_t` into which the generated output is to be written, rather than an array of type `char`.
- The argument *format* specifies an array of type `wchar_t` that describes how subsequent arguments are converted for output, rather than an array of type `char`.
- `%c` without an `l` prefix means an `int` arg is to be converted to `wchar_t`, as if `mbtowl()` were called, and then written.
- `%c` with `l` prefix means a `wint_t` is converted to `wchar_t` and then written.
- `%s` without an `l` prefix means a character array containing a multibyte character sequence is to be converted to an array of `wchar_t` and then written. The conversion will take place as if `mbrtowl()` were called repeatedly.
- `%s` with `l` prefix means an array of `wchar_t` will be written. The array is written up to but not including the terminating null character, unless the precision specifies a shorter output.

A null wide character is written at the end of the wide characters written; the null wide character is not counted as part of the returned sum. If copying takes place between objects that overlap, the behavior is undefined.

### Special Behavior for XP4

If you define any feature test macro specifying XP4 behavior before the statement in your program source file to include the `wchar` header, then you must also define the `_MSE_PROTOS` feature test macro to make the declaration of the `swprintf()` function in the `wchar` header available when you compile your program. Please see Table 3 on page 18 for a list of XP4 and other feature test macros.

**Returned Value**

Returns the number of wide characters written in the array, not counting the terminating null wide character.

**Related Information**

- “wchar.h” on page 54
- `sprintf()`, in “`fprintf()` - `printf()` - `sprintf()` — Format and Write Data” on page 436

## swscanf() — Read a Wide-Character String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

### Format

#### Non-XP4

```
#include <wchar.h>
```

```
int swscanf(const wchar_t *wcs, const wchar_t *format, ...);
```

#### XP4

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>
```

```
int swscanf(const wchar_t *wcs, const wchar_t *format, ...);
```

### General Description

The `swscanf()` function is equivalent to `sscanf()`, except for the following:

- The argument `wcs` specifies an array of type `wchar_t` from which the input is to be obtained, rather than an array of type `char`.
- The argument `format` specifies an array of type `wchar_t` that describes the admissible input sequences and how they are to be converted for assignment, rather than an array of type `char`.
- `%c` with no `l` prefix means one or more (depending on precision) `wchar_t` is to be converted to multibyte characters, as if by repeated calls to `wcrtomb()`, and copied to the character array pointed to by the corresponding argument.
- `%c` with the `l` prefix means one or more (depending on precision) `wchar_t` is copied to the array of `wchar_t` pointed to by the corresponding argument.
- `%s` with no `l` prefix means a sequence of non-white `wchar_t` will be converted, as if by repeated calls to `wcrtomb()`, and copied, including the terminating null character, to the character array pointed to by the corresponding argument.
- `%s` with the `l` prefix means an array of `wchar_t` will be including the terminating null wide character, to the array of `wchar_t` pointed to by the corresponding argument.

Reaching the end of the wide character string is equivalent to reaching the end of the `char` string for the `sscanf()` function. If copying takes place between objects that overlap, the behavior is undefined.

### Special Behavior for XP4

If you define any feature test macro specifying XP4 behavior before the statement in your program source file to include the `wchar` header, then you must also define the `_MSE_PROTOS` feature test macro to make the declaration of the `swscanf()` function in the `wchar` header available when you compile your program. Please see Table 3 on page 18 for a list of XP4 and other feature test macros.

**Returned Value**

Returns EOF if an input failure occurs before any conversion. Otherwise, the `swscanf()` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

**Related Information**

- “`wchar.h`” on page 54
- “`fsync()` — Write Changes to Direct-Access Storage” on page 483

## symlink() — Create a Symbolic Link to a Path Name

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1a XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX1_SOURCE 2
#include <unistd.h>
```

```
int symlink(const char *pathname, const char *slink);
```

### General Description

Creates the symbolic link named by *slink* with the file specified by *pathname*. File access checking is not performed on the file *pathname*, and the file need not exist. In addition, a symbolic link can cross file system boundaries.

A symbolic link path name is resolved in this fashion:

- When a component of a path name refers to a symbolic link rather than to a directory, the path name contained in the symbolic link is resolved.
- If the path name in the symbolic link begins with / (slash), the symbolic link path name is resolved relative to the process root directory.  
If the path name in the symbolic link does not start with / (slash), the symbolic link path name is resolved relative to the directory that contains the symbolic link.
- If the symbolic link is the last component of a path name, it may or may not be resolved. Resolution depends on the function using the path name. For example, `rename()` does not resolve a symbolic link when it appears as the final component of either the new or old path name. However, `open` does resolve a symbolic link when it appears as the last component.
- If the symbolic link is not the last component of the original path name, remaining components of the original path name are resolved relative to the symbolic link.
- When a / (slash) is the last component of a path name and it is preceded by a symbolic link, the symbolic link is always resolved.

Because the mode of a symbolic link cannot be changed, its mode is ignored during the lookup process. Any files and directories to which a symbolic link refers are checked for access permission.

### Returned Value

If successful, `symlink()` returns zero. If unsuccessful, `symlink()` returns the value `-1`, and it does not affect any file it names. `symlink()` sets `errno` to one of the following:

**EACCES** A component of the *slink* path prefix denies search permission, or write permission is denied in the parent directory of the symbolic link to be created.

EINVAL	This may be returned for either of these reasons: <ul style="list-style-type: none"> <li>• There is a null character in <i>pathname</i>.</li> <li>• <i>slink</i> has a slash as its last component, which indicates that the preceding component will be a directory. A symbolic link cannot be a directory.</li> </ul>
ENAMETOOLONG	<i>pathname</i> is longer than PATH_MAX characters, or some component of <i>pathname</i> is longer than NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH_MAX. The PATH_MAX and NAME_MAX values can be determined with pathconf().
ENOTDIR	A component of the path prefix of <i>slink</i> is not a directory.
ELOOP	A loop exists in symbolic links. This error is issued if more than POSIX_SYMLINK symbolic links are encountered during resolution of the <i>slink</i> argument.
EEXIST	The file named by <i>slink</i> already exists.
ENOSPC	The new symbolic link cannot be created because there is no space left on the file system that will contain the symbolic link.
EROFS	The file <i>slink</i> cannot reside on a read-only system.

**For XPG4.2 added the following:**

EIO	An I/O error occurred while reading from the file system.
ENOENT	A component of <i>slink</i> does not name an existing file or <i>slink</i> is an empty string.

**Example**

```

/* This example works only under OS/390 C, not OS/390 C++ */
#define _POSIX1_SOURCE 2
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

main() {
    char fn[]="test.file";
    char sln[]="test.symlink";
    int fd;

    if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        close(fd);
        puts("before symlink()");
        system("ls -il test.*");
        if (symlink(fn, sln) != 0) {
            perror("symlink() error");
            unlink(fn);
        }
        else {
            puts("after symlink()");
            system("ls -il test.*");
        }
    }
}

```

```

        unlink(fn);
        puts("after first unlink()");
        system("ls -il test.*");
        unlink(sln);
    }
}
}

```

### Output

```

before symlink()
4030 --w----- 1 MVSUSR1 SYS1    0 Apr 20 13:57 test.file

after symlink()
4030 --w----- 1 MVSUSR1 SYS1    0 Apr 20 13:57 test.file
4031 l----- 1 MVSUSR1 SYS1    9 Apr 20 13:57 test.symlink -> test.file
after first unlink()
4031 l----- 1 MVSUSR1 SYS1    9 Apr 20 13:57 test.symlink -> test.file

```

### Related Information

- “unistd.h” on page 53
- “link() — Create a Link to a File” on page 749
- “readlink() — Read the Value of a Symbolic Link” on page 1091
- “unlink() — Remove a Directory Entry” on page 1660

## sync() — Schedule File System Updates

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
void sync(void);
```

### General Description

The sync() function causes all information in memory that updates file systems to be scheduled for writing out to all file systems.

The writing, although scheduled, is not necessarily complete upon return from sync().

### Returned Value

The sync() function returns no value.

No errors are defined.

### Related Information

- “fsync() — Write Changes to Direct-Access Storage” on page 483



## sysconf() — Determine System Configuration Options

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
long sysconf(int name);
```

### General Description

Determines the value of a configurable system option.

*int name* Specifies the system configuration option to be obtained. The value of *name* can be any one of the following set of symbols defined in the `unistd.h` header file, each corresponding to a system configuration option:

`_SC_ARG_MAX`

Represents `ARG_MAX`, as defined by the values returned by `sysconf()`, the maximum number of bytes of arguments and environment data that can be passed in an `exec` function.

`_SC_CHILD_MAX`

Represents `CHILD_MAX`, as defined by the values returned by `sysconf()`, the maximum number of processes that a real user ID (UID) may have running simultaneously.

`_SC_CLK_TCK`

Represents the `CLK_TCK` macro defined in the `time.h` header file: the number of clock ticks in a second.

`_SC_JOB_CONTROL`

Represents the `_POSIX_JOB_CONTROL` macro that can be defined in the `unistd.h` header file. This indicates that certain job control operations are implemented by this version of the operating system. If `_POSIX_JOB_CONTROL` is defined, various functions (for example, `setpgid()`) have more functionality than when the macro is not defined.

`_SC_NGROUPS_MAX`

Represents `NGROUPS_MAX`, as defined by the values returned by `sysconf()`, the maximum number of supplementary group IDs (GIDs) that can be associated with a process.

- \_SC\_OPEN\_MAX**  
Represents OPEN\_MAX, as defined by the values returned by sysconf(), the maximum number of files that a single process can have open at one time.
- \_SC\_PAGE\_SIZE**  
Returns the current page size in bytes.
- \_SC\_PAGESIZE**  
Returns the current page size in bytes.
- \_SC\_SAVED\_IDS**  
Represents the \_POSIX\_SAVED\_IDS macro, which may be defined in unistd.h header file, indicating that this POSIX implementation has a saved set UID and a saved set GID. This symbol affects the behavior of such functions as setuid() and setgid().
- \_SC\_STREAM\_MAX**  
Represents the \_STREAM\_MAX macro, which may be defined in the unistd.h header file, indicating the maximum number of streams that a process can have open at one time.
- \_SC\_THREADS\_MAX\_NP**  
Represents the THREAD\_MAX macro, as defined by the values returned by sysconf(), the maximum number of concurrent threads processed by pthread\_create(), including running, queued, and exited undetached threads in the caller's process.
- \_SC\_THREAD\_TASKS\_MAX\_NP**  
Represents the THREAD\_TASKS\_MAX macro, as defined by the values returned by sysconf(), the maximum number of MVS tasks simultaneously in use for threads processed by pthread\_create() in the caller's process.
- \_SC\_TZNAME\_MAX**  
Represents the \_TZNAME\_MAX macro, which may be defined in the unistd.h header file, indicating the maximum length of the name of a time zone.
- \_SC\_TTY\_GROUP**  
Retrieves the group number associated with the TTYGROUP() initialization parameter.
- \_SC\_VERSION**  
Represents the \_POSIX\_VERSION macro, which may be defined in the unistd.h header file, indicating the version of the POSIX.1 standard that the system conforms to.

## Returned Value

Returns the value associated with the specified option. If the variable corresponding to *name* exists but is not supported by the system, sysconf() returns the value -1 but does not change the value of errno. If sysconf() fails in some other way, it returns the value -1.

When unsuccessful, sysconf() sets errno to EINVAL, indicating that the value specified for the *name* argument is incorrect.

## Example CBC3BS61

```
/* CBC3BS61
   This example determines the value of ARG_MAX.
   */
#define _POSIX_SOURCE
#include <stdio.h>
#include <unistd.h>
#include <errno.h>

main() {
    long result;

    errno = 0;
    puts("examining ARG_MAX limit");
    if ((result = sysconf(_SC_ARG_MAX)) == -1)
        if (errno == 0)
            puts("ARG_MAX is not supported.");
        else perror("sysconf() error");
    else
        printf("ARG_MAX is %ld\n", result);
}
```

## Output

```
examining ARG_MAX limit
ARG_MAX is 1048576
```

## Related Information

- “unistd.h” on page 53
- “clock() — Determine Processor Time” on page 189
- “exec Functions” on page 322

## syslog() — Send a Message to the Control Log

### Standards

Standards / Extensions	C or C++	Dependencies
XP4.2	both	MVS 5.1

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <syslog.h>
```

```
void syslog(int priority, const char *message, . . . /* argument */);
```

### General Description

The `syslog()` function sends a message to an implementation-specific logging facility, which loads it in an appropriate system log, writes it to the system console, forwards it to a list of users, or forwards it to the logging facility on another host over the network. The logged message includes a message header and a message body. The message header consists of a facility indicator, a severity indicator, a timestamp, a tag string, and optionally the process ID.

The message body is generated from the *message* and following arguments in the same manner as if these were arguments to the `printf()` function, except that occurrences of `%m` in the format string pointed to by the *message* argument are replaced by the error message string associated with the current value of `errno`. A trailing newline character is added if needed.

**Note:** If the total length of the format string and the parameters is greater than 4096 bytes, then the results are undefined.

Values of the *priority* argument are formed by ORing together a severity level values and an option facility value. If no facility value is specified, the current default facility value is used. Possible values of severity level include:

LOG_EMERG	A Panic condition. This is normally broadcast to all processes.
LOG_ALERT	A condition that should be corrected immediately, such as a corrupted system database.
LOG_CRIT	Critical conditions, such as hard device errors.
LOG_ERR	Errors.
LOG_WARNING	Warning messages.
LOG_NOTICE	Conditions that are not error conditions, but that may require special handling.
LOG_INFO	Informational messages.
LOG_DEBUG	Messages that contain information normally of use only when debugging a program.

The facility indicates the application or system component generating the message. Possible facility values include:

LOG_USER	Message generated by random processes. This is the default facility identifier if none is specified.
LOG_LOCAL0	Reserved for local use.
LOG_LOCAL1	Reserved for local use.
LOG_LOCAL2	Reserved for local use.
LOG_LOCAL3	Reserved for local use.
LOG_LOCAL4	Reserved for local use.
LOG_LOCAL5	Reserved for local use.
LOG_LOCAL6	Reserved for local use.
LOG_LOCAL7	Reserved for local use.

### Returned Value

The syslog() function returns no value.

### Related Information

- “syslog.h” on page 47
- “closelog() — Close the Control Log” on page 196
- “openlog() — Open the System Control Log” on page 882
- “fprintf() - printf() - sprintf() — Format and Write Data” on page 436
- “setlogmask() — Set the Mask for the Control Log” on page 1250

system() — Execute a Command

Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

Format

```
#include <stdlib.h>

int system(const char *string);
```

General Description

Using the system() function, you can call commands, EXECs, CLISTs, or executable modules from a system call under MVS and TSO. It is not supported under CICS.

| To provide an ASCII input/output format for applications using this function, define  
| feature test macro \_\_LIBASCII as described on page 22.

In the following, *MVS* specifically refers to MVS batch (excluding batch TSO), whereas *TSO* includes both batch TSO (IKJEFT01, as the program is specified on the JCL EXEC statement) and interactive TSO (which is TSO at a terminal).

Not all targets can be called from all systems. For example, a non-TSO environment does not support calling TSO commands.

The *string* argument can take one of two formats:

Command Line    A string with TSO command line syntax: *command\_name parm1 parm2 ...* Here are two examples.

```
system("loadlib list user1 loadlib a (disk");
system("user_pgm1 1234 abcd xyz");
```

Named Program    A string with no embedded blanks except in the PARM. You can write portable code that calls OS/390 C or OS/390 C++ modules.

```
"PGM=program_name[,PARM='....']"
```

Here are two examples:

```
system("PGM=loadlib, PARM='list user1 loadlib a (disk'");
system("PGM=user_pgm1, PARM='1234 abcd xyz'");
```

If the argument is a named-program string, the system function calls *program\_name* with the parameters following "PARM=", if any.

The two formats are supported for maximum portability when invoking user modules on multiple systems, use the named-program format.

If system() receives a string in the form of a command, the system() function passes the given *string* to the command processor for execution.

If the string is of the form "PGM... ", the `system()` function calls the program *program\_name* to be executed with the parameters following PARM, if any.

In either case, if the specified module is a OS/390 C or OS/390 C++ module, full initialization and termination will be performed: automatic closing of files and releasing of fetched modules. To pass information across a `system()` call, use memory files. Memory files are not removed until the root program terminates or *clrmemf* is used.

### Special Behavior for POSIX C

`system()` passes the *string* to the sh shell command for execution. The environment is established by the runtime library by issuing a `fork()` and an `execl()` of the shell.

`system()` ignores the SIGINT and SIGQUIT signals, and blocks the SIGCHLD signal while it waits for the command specified by the *string* to end.

Under OS/390 UNIX services, `system()` is issued for TSO/E, MVS, or OS/390 UNIX services shell commands. For an open application, `system()` assumes that the command is a shell command unless the environment variable, `__POSIX_SYSTEM`, is set to no, No, or NO. If the variable is set to one of these values, or the application is not an open application, `system()` assumes that the command is a TSO/E or MVS command.

If the application is a forked child (such as running under the shell), `system()` cannot run TSO/E commands.

`system()` has these restrictions:

- A program running with POSIX(ON) can call another program that will also use POSIX(ON) only if the calling program invokes the called program out of the shell, without `__POSIX_SYSTEM` set to "NO".
- `system()` cannot run TSO/E commands, if there is an `exec` to the calling program. The TSO/E address space is no longer available after the `exec`.
- `system()` is not thread safe. It cannot be called simultaneously from more than one thread. A multi-threaded application must ensure that no more than one `system()` call is ever outstanding from the various threads. If this restriction is violated, unpredictable results may occur.

For traditional C application programs, `system()` is issued only for TSO/E and MVS commands. It cannot pass commands to the shell.

See "OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions" on page 9 for more information about using POSIX support.

### Mixed Environments across a `system()` Call

The mixing of OS/390 Language Environment, the Version 1 or Version 2 C/370 Library and system programming environments across a `system()` call is prohibited under C/OS/390. Whichever of these environments is active when the first `system()` call is issued is the only one that is tolerated in the `system()` call chain. Results are undefined if you violate this rule.

## Returned Value

If the argument is a null pointer, the `system()` function returns nonzero if a command processor exists and the value 0 if it does not exist.

Runtime options are passed from the caller to the `system()` call. If you have `TRAP(ON)` and a signal handler for a certain exception that occurs in the `system()` call, the signal handler may be invoked to handle the exception.

## OS/390 Considerations

Under OS/390, `system()` accepts either command-line or named-program strings. However, command-line strings are restricted to specifying only user modules (that is, TSO commands and clists *cannot* be specified). In the case of either a command-line or named-program string, `system` will search the usual MVS sources (steplib-joblib concatenation, Link Pack Area (LPA), Extended Link Pack Area (ELPA), and the link libraries) for the specified program name. The return value will be from the user module, if successfully called. If `system()` cannot call the specified module, the return value is -1 and `errno` is set appropriately.

Using an ATTACH will not allow you to share your memory files or standard streams between programs under MVS. When calling a program as a command (that is, not using the PGM format in the `system()` string), the TSO command processor uses an ATTACH to execute the program. Thus, your memory files and standard streams will not be shared. Likewise, if you make a `system()` call to a CLIST that then calls a OS/390 C or OS/390 C++ program, your memory files and standard streams will not be shared. See the *IBM Language Environment Programming Guide* for more information on using LINKs.

## Notes:

- If a module is placed in the STEPLIB or ISPLLIB (under ISPF), TSO will allow the module to be activated as a command and thus, if a CLIST had the same name, the module would be activated first. Activating a module as a command involves a different input interface and, if required, you can use the CALL command to activate a module that is not prepared to take the TSO Command input format. OS/390 C and OS/390 C++ modules can be called as TSO commands.
- A module that exists in the Link Pack Area (LPA) or Extended Link Pack Area (ELPA) and not on the STEPLIB concatenation (ISPLLIB on ISPF) will not be activated as a TSO command, and is treated as an executable module.

The returned value from `system()` will be nonzero if the command processor cannot execute the command. The macros `__abendcode()` and `__rsncode()` will contain the abend code and reason code from a failing TSO CLIST or command.

If the string is of the form "PGM... ", the `system()` function calls the program *program\_name* to be executed with the parameters following PARM, if any. Under MVS, `system()` will only look for an executable module and cannot be used to invoke a TSO command or CLIST. The returned value will be the returned value from the executed program if successfully called. If `system()` cannot call the specified program, the returned value is -1 and `errno` is set appropriately.

## TSO Considerations



Under TSO, `system()` accepts either command-line or the named-program strings.

Command-line strings are presented to the TSO command processor and can be used to execute user modules, TSO commands, or CLISTs. The return value from `system` will be nonzero if the specified command or program could not be executed. If there is any ambiguity as to what is to be run when a command-line string is supplied, the hierarchy is:

1. TSO command or user module
2. CLIST

If a command-line string is used and if a CLIST exists with the same name as a TSO or user command, the TSO EXEC command would have to be used to specifically invoke the CLIST.

Named-program strings are *not* presented to the TSO command processor and can only be used to execute user modules. The system function calls the program `program_name` with the parameters following `PARM=`, if any. The return value will be the return value from the executed program if it is successfully called. If the system cannot call the specified program, the return value is `-1` and `errno` is set appropriately.

You can use `system()` to pass a fully qualified data-set name to TSO. Use two single-quotation marks around the data set name:

```
system("PGM=TSOCALL,PARM='COMMAND 'dsn'")
```

As with MVS, when you are using ATTACH, you cannot share your memory files or standard streams between programs under TSO. When making a system call with a command-line string, the TSO command processor effectively uses ATTACH to execute the program. Thus your memory files and standard streams will not be shared. Likewise, if you make a `system()` call to a CLIST, which then calls a OS/390 C or OS/390 C++ program, your memory files and standard streams will not be shared. This is the reason that you should use named-program strings for maximum portability. Such system calls provide the same memory file and standard stream sharing under all systems. In addition, they provide the most compatible search order for user modules across systems.

### Returned Value for POSIX C

This information applies to OS/390 C programs that have issued a `POSIX(ON)` `system()` call. If *string* is a NULL pointer, `system()` returns nonzero. If *string* is not NULL, `system()` returns the termination status of the command language interpreter in the format specified by `waitpid()`.

If a child process cannot be created to invoke `sh`, or if the termination status for the command language interpreter cannot be obtained, `system()` returns the value `-1` and sets `errno` to one of the following:

**EAGAIN**     There are insufficient resources to create another process, or else the process has already reached the maximum number of processes you can run.

**ENOMEM**    The process requires more space than is available.

See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information about using POSIX support.

**Example**

/\* This example illustrates how to use system() to execute a command which returns the time. The example works only under TS0.

```
*/  
#include <stdlib.h>
```

```
int main(void)  
{  
    int rc;  
    rc = system("time");  
    exit(0);  
}
```

/\* This example may only be used in a POSIX program. \*/

```
#include <stdlib.h>  
  
int main(void)  
{  
    int result;  
  
    result = system("date | tee result.log");  
}
```

**Related Information**

- “stdlib.h” on page 45
- “clrmmemf() — Clear Memory Files” on page 197
- “signal() — Handle Interrupts” on page 1330

## t\_accept() — Accept a Connect Request

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_accept(int fd, int resfd, struct t_call *call);
```

### General Description

t\_accept() is issued by a transport user to accept a connect request. The parameter *fd* identifies the local transport endpoint where the connect indication arrived. *resfd* specifies the local transport endpoint where the connection is to be established, and *call* contains information required by the transport provider to complete the connection. The parameter *call* points to a *t\_call* structure which contains the following members:

```
struct netbuf  addr;
struct netbuf  opt;
struct netbuf  udata;
int            sequence;
```

In *call*, *addr* is the protocol address of the calling transport user. *opt* indicates any options associated with the connection. *udata* points to any user data to be returned to the caller, and *sequence* is the value returned by t\_listen() that uniquely associates the response with a previously received connect indication. The address of the caller, *addr* may be null (length zero). Where *addr* is not null then it may optionally be checked by XTI.

A transport user may accept a connection on either the same, or on a different, local transport endpoint than the one on which the connect indication arrived. Before the connection can be accepted on the same endpoint (*resfd==fd*), the user must have responded to any previous connect indications received on that transport endpoint (via t\_accept() or t\_snddis() ). Otherwise, t\_accept() fails and sets *t\_errno* to TINDOUT.

If a different transport endpoint is specified (*resfd!=fd*), then the user may or may not choose to bind the endpoint before the t\_accept() is issued. If the endpoint is not bound prior to the t\_accept() , then the transport provider will automatically bind it to the same protocol address *fd* is bound to. If the transport user chooses to bind the endpoint it must be bound to a protocol address with a *qlen* of zero and must be in the T\_IDLE state before the t\_accept() is issued.

The call to t\_accept() will fail with *t\_errno* set to TL00K if there are indications (for example, connect or disconnect) waiting to be received on the endpoint *fd*.

Return of user data over a connection accept is not supported under TCP, so the *udata* field is always meaningless.

When the user does not indicate any option (*call->opt.len == 0*) it is assumed that the connection is to be accepted unconditionally. The transport provider may

choose options other than the defaults to ensure that the connection is accepted successfully.

Due to implementation restrictions, behavior is undefined if a different process accepts a connection pending on an endpoint than obtained it (with `t_listen`).

### Valid States

*fd*: T\_INCON *resfd* (*fd!=resfd*): T\_IDLE

### Returned Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and `t_errno` is set to indicate an error. On failure, `t_errno` is set to one of the following:

TBADF	The file descriptor <i>fd</i> or <i>resfd</i> does not refer to a transport endpoint.
TOUTSTATE	The function was called in the wrong sequence on the transport endpoint referenced by <i>fd</i> , or the transport endpoint referred to by <i>resfd</i> is not in the appropriate state.
TACCES	The user does not have permission to accept a connection on the responding transport endpoint or to use the specified options.
TBADOPT	The specified options were in an incorrect format or contained illegal information.
TBADDATA	The amount of user data specified was not within the bounds allowed by the transport provider.
TBADADDR	The specified protocol address was in an incorrect format or contained illegal information.
TBADSEQ	An invalid sequence number was specified.
TLOOK	An asynchronous event has occurred on the transport endpoint referenced by <i>fd</i> and requires immediate attention.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TSYSERR	A system error has occurred during execution of this function.
TINDOUT	The function was called with <i>fd==resfd</i> but there are outstanding connection indications on the endpoint. Those other connection indications must be handled either by rejecting them via <code>t_snddis</code> (3) or accepting them on a different endpoint via <code>t_accept</code> (3).
TPROVMISMATCH	The file descriptors <i>fd</i> and <i>resfd</i> do not refer to the same transport provider.
TRESQLEN	The endpoint referenced by <i>resfd</i> (where <i>resfd != fd</i> ) was bound to a protocol address with a <i>qlen</i> that is greater than zero.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI ( <code>t_errno</code> ).
TRESADDR	This transport provider requires both <i>fd</i> and <i>resfd</i> to be bound to the same address. This error results if they are not.

**Related Information**

- “xti.h” on page 56
- “t\_connect() — Establish a Connection with Another Transport User” on page 1524
- “t\_getstate() — Get the Current State” on page 1569
- “t\_listen() — Listen for a Connect Indication” on page 1577
- “t\_open() — Establish a Transport Endpoint” on page 1594
- “t\_optmgmt() — Manage Options for a Transport Endpoint” on page 1596
- “t\_rcvconnect() — Receive the Confirmation from a Connect Request” on page 1607

## takesocket() — Acquire a Socket from Another Program

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/types.h>
#include <socket.h>
```

```
int takesocket(struct clientid *clientid, int sdesc);
```

### General Description

The `takesocket()` call acquires a socket from another program. Typically, the other program passes its client ID and socket descriptor, and/or process id (PID), to your program through your program's startup parameter list.

### Parameter Description

*clientid* A pointer to the *clientid* of the application from which you are taking a socket.

*sdesc* The descriptor of the socket to be taken.

If your program is using the PID to ensure integrity between `givesocket()` and `takesocket()`, before issuing the `takesocket()` call, your program should set the *c\_pid.pid* field of the *clientid* structure to the PID of the giving program (i.e. program that issued the `givesocket()` call). This identifies the process from which the socket is to be taken. If the *c\_reserved.type* field of the *clientid* structure was set to `SO_CLOSE` on the `givesocket()` call, *c\_close.SockToken* of *clientid* structure should be used as input to `takesocket()` instead of the normal socket descriptor. See “`givesocket()` — Make the Specified Socket Available” on page 633 for a description of the *clientid* structure.

### Returned Value

The value -1 indicates an error. The value of `errno` indicates the specific error. If not -1, the return value is the new socket descriptor.

### Errno Description

EACCES	The other application did not give the socket to your application.
EBADF	The <i>sdesc</i> parameter does not specify a valid socket descriptor owned by the other application, or the socket has already been taken.
EFAULT	Using the <i>clientid</i> parameter as specified would result in an attempt to access storage outside the caller's address space.
EINVAL	The <i>clientid</i> parameter does not specify a valid client identifier. Either the client process cannot be found, or the client exists, but has no outstanding givesockets.
EMFILE	The socket descriptor table is already full.

**Related Information**

- “sys/types.h” on page 49
- “sys/socket.h” on page 48
- “givesocket() — Make the Specified Socket Available” on page 633
- “getclientid() — Get the Identifier for the Calling Application” on page 501
- “\_\_getclientid() — Get the PID Identifier for the Calling Application” on page 503

## t\_alloc() — Allocate a Library Structure

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
char *t_alloc(int fd, int struct_type, int fields);
```

### General Description

Dynamically allocates memory for the various transport function argument structures as specified below. `t_alloc()` allocates memory for the specified structure, and memory for buffers referenced by the structure.

The structure to allocate is specified by *struct\_type* and must be one of the following:

T_BIND	struct t_bind
T_CALL	struct t_call
T_OPTMGMT	struct t_optmgmt
T_DIS	struct t_discon
T_UNITDATA	struct t_unitdata
T_UDERROR	struct t_uderr
T_INFO	struct t_info

where each of these structures may subsequently be used as an argument to one or more transport functions.

Each of the above structures, except T\_INFO, contains at least one field of type struct netbuf. For each field of this type, the user may specify that the buffer for that field should be allocated as well. The length of the buffer allocated will be equal to or greater than the appropriate size as returned in the *info* argument of `t_open()` or `t_getinfo()`. The relevant fields of the *info* argument are described in the following list. The *fields* argument specifies which buffers to allocate, where the argument is the bitwise-or of any of the following:

T_ADDR	The <i>addr</i> field of the t_bind, t_call, t_unitdata or t_uderr structures.
T_OPT	The <i>opt</i> field of the t_optmgmt, t_call, t_unitdata or t_uderr structures.
T_UDATA	The <i>udata</i> field of the t_call, t_discon or t_unitdata structures.
T_ALL	All relevant fields of the given structure. Fields which are not supported by the transport provider specified by <i>fd</i> will not be allocated.

For each relevant field specified in *fields*, `t_alloc()` allocates memory for the buffer associated with the field, and initializes the *len* field to zero and the *buf* pointer and *maxlen* field accordingly. Irrelevant or unknown values passed in fields are ignored. Since the length of the buffer allocated will be based on the same size information that is returned to the user on a call to `t_open()` and `t_getinfo()`, *fd* must refer to the transport endpoint through which the newly allocated structure will be passed. In this way the appropriate size information can be accessed. If the size value associated with any specified field is -1 or -2 (see `t_open()` or `t_getinfo()`), `t_alloc()` will be



unable to determine the size of the buffer to allocate and will fail, setting *t\_errno* to *TSYSERR* and *errno* to *EINVAL*. For any field not specified in *fields*, *buf* will be set to the null pointer and *len* and *maxlen* will be set to zero.

Use of *t\_alloc()* to allocate structures helps ensure the compatibility of user programs with future releases of the transport interface functions.

### Valid States

All - except for *T\_UNINIT*

### Returned Value

On successful completion, *t\_alloc()* returns a pointer to the newly allocated structure. On failure, a null pointer is returned and *t\_errno* is set to one of the following:

<i>TBADF</i>	The specified file descriptor does not refer to a transport endpoint.
<i>TSYSERR</i>	A system error has occurred during execution of this function.
<i>TNOSTRUCTYPE</i>	Unsupported <i>struct_type</i> requested. This can include a request for a structure type which is inconsistent with the transport provider type specified, that is, connection-oriented or connectionless.
<i>TPROTO</i>	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI ( <i>t_errno</i> ).

### Related Information

- “*xti.h*” on page 56
- “*t\_free()* — Free a Library Structure” on page 1564
- “*t\_getinfo()* — Get Protocol-specific Service Information” on page 1566
- “*t\_open()* — Establish a Transport Endpoint” on page 1594

## tan() — Calculate Tangent

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double tan(double x);
```

### General Description

Calculates the tangent of  $x$ , where  $x$  is expressed in radians. If  $x$  is large, a partial loss of significance in the result can occur.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

Returns the calculated tangent of  $x$ . If the correct value would cause underflow, zero is returned. If the result overflows, HUGE\_VAL or -HUGE\_VAL is returned. For both under- and overflow, the macro ERANGE is stored in errno.

### Example

#### CBC3BT01

```
/* CBC3BT01
   This example computes x as the tangent of PI/4.
*/
#include <math.h>
#include <stdio.h>

int main(void)
{
    double pi, x;

    pi = 3.1415926;
    x = tan(pi/4.0);

    printf("tan( %lf ) is %lf\n", pi/4, x);
}
```

### Output

```
tan( 0.785398 ) is 1.000000
```

### Related Information

- “math.h” on page 35
- “acos() — Calculate Arccosine” on page 85
- “acosh() — Hyperbolic Arccosine” on page 87
- “asin() — Calculate Arcsine” on page 108
- “asinh() — Hyperbolic Arcsine” on page 110

- “atan() - atan2() — Calculate Arctangent” on page 113
- “atanh() — Hyperbolic Arctangent” on page 115
- “cos() — Calculate Cosine” on page 229
- “cosh() — Calculate Hyperbolic Cosine” on page 231
- “sin() — Calculate Sine” on page 1360
- “sinh() — Calculate Hyperbolic Sine” on page 1362
- “tanh() — Calculate Hyperbolic Tangent” on page 1502

## tanh() — Calculate Hyperbolic Tangent

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <math.h>
```

```
double tanh(double x);
```

### General Description

Calculates the hyperbolic tangent of  $x$ , where  $x$  is expressed in radians. The result of the `tanh()` function cannot have a range error.

**Note:** This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE Floating Point” on page 65 for more information about IEEE floating-point.

### Returned Value

Returns the calculated value of the arctangent of  $x$ . If the result underflows, the function returns zero and sets the `errno` to `ERANGE`.

### Example

#### CBC3BT02

```
/* CBC3BT02
   This example computes x as the hyperbolic tangent of PI/4.
*/
#include <math.h>
#include <stdio.h>

int main(void)
{
    double pi, x;

    pi = 3.1415926;
    x = tanh(pi/4);

    printf("tanh( %lf ) = %lf\n", pi/4, x);
}
```

### Output

```
tanh( 0.785398 ) = 0.655794
```

### Related Information

- “`math.h`” on page 35
- “`acos()` — Calculate Arccosine” on page 85
- “`acosh()` — Hyperbolic Arccosine” on page 87
- “`asin()` — Calculate Arcsine” on page 108
- “`asinh()` — Hyperbolic Arcsine” on page 110
- “`atan()` - `atan2()` — Calculate Arctangent” on page 113

- “`atanh()` — Hyperbolic Arctangent” on page 115
- “`cos()` — Calculate Cosine” on page 229
- “`cosh()` — Calculate Hyperbolic Cosine” on page 231
- “`sin()` — Calculate Sine” on page 1360
- “`sinh()` — Calculate Hyperbolic Sine” on page 1362
- “`tan()` — Calculate Tangent” on page 1500

## t\_bind() — Bind an Address to a Transport Endpoint

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_bind(int fd, struct t_bind *req, struct t_bind *ret);
```

### General Description

Associates a protocol address with the transport endpoint specified by *fd* and activates that transport endpoint. In connection mode, the transport provider may begin enqueueing incoming connect indications, or servicing a connection request on the transport endpoint. In connectionless mode, the transport user may send or receive data units through the transport endpoint.

The *req* and *ret* arguments point to a *t\_bind* structure containing the following members:

```
struct netbuf  addr;
unsigned      qlen;
```

The *addr* field of the *t\_bind* structure specifies a protocol address, and the *qlen* field is used to indicate the maximum number of outstanding connect indications.

The parameter *req* is used to request that an address, represented by the *netbuf* structure, be bound to the given transport endpoint. The parameter *len* specifies the number of bytes in the address, and *buf* points to the address buffer. The parameter *maxlen* has no meaning for the *req* argument. On return, *ret* contains the address that the transport provider actually bound to the transport endpoint. This is the same as the address specified by the user in *req*. In *ret*, the user specifies *maxlen*, which is the maximum size of the address buffer, and *buf* which points to the buffer where the address is to be placed. On return, *len* specifies the number of bytes in the bound address, and *buf* points to the bound address. If *maxlen* is not large enough to hold the returned address, an error results.

If the requested address is not available, *t\_bind()* returns -1 with *t\_errno* set as appropriate. If no address is specified in *req* (the *len* field of *addr* in *req* is zero or *req* is NULL), the transport provider will assign an appropriate address to be bound, and will return that address in the *addr* field of *ret*. If the transport provider could not allocate an address, *t\_bind()* fails with *t\_errno* set to *TNOADDR*.

The parameter *req* may be a null pointer if the user does not wish to specify an address to be bound. Here, the value of *qlen* is assumed to be zero, and the transport provider assigns an address to the transport endpoint. Similarly, *ret* may be a null pointer if the user does not care what address was bound by the provider and is not interested in the negotiated value of *qlen*. It is valid to set *req* and *ret* to the null pointer for the same call, in which case the provider chooses the address to bind to the transport endpoint and does not return that information to the user.

The *qlen* field specifies the number of outstanding connect indications that the transport provider should support for the given transport endpoint. An outstanding connect indication is one that has been passed to the transport user by the transport provider, but which has not been accepted or rejected. A value of *qlen* greater than zero is only meaningful when issued by a passive transport user that expects other users to call it. The value of *qlen* will be negotiated by the transport provider and will always be negotiated to 1 (one) from any nonzero value. On return, the *qlen* field in *ret* will contain the negotiated value.

If *fd* refers to a connection-oriented service, then multiple endpoints may be bound to the same protocol address by way of connections accepted on an endpoint via *t\_accept*. The TCP transport provider will not permit the user to explicitly bind multiple endpoints to the same address. It is also not possible to bind an endpoint to more than one protocol address. If a user attempts to explicitly bind multiple endpoints to a protocol address, the second and subsequent binds will fail with *t\_errno* set to *TADDRBUSY*. When a user accepts a connection on the transport endpoint that is being used as the listening endpoint, the bound protocol address will be found to be busy for the duration of the connection, until a *t\_unbind()* or *t\_close()* call has been issued. No other transport endpoints may be bound for listening on that same protocol address while that initial listening endpoint is active (in the data transfer phase or in the *T\_IDLE* state). This prevents more than one transport endpoint bound to the same protocol address from accepting connect indications.

### Valid States

*T\_UNBND*

### Returned Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *t\_errno* is set to indicate an error. On failure, *t\_errno* is set to one of the following:

**TBADF** The specified file descriptor does not refer to a transport endpoint.

**TOUTSTATE** The function was issued in the wrong sequence.

**TBADADDR** The specified protocol address was in an incorrect format or contained illegal information.

**TNOADDR** The transport provider could not allocate an address.

**TACCES** The user does not have permission to use the specified address.

**TBUFOVFLW** The number of bytes allowed for an incoming argument (*maxlen*) is greater than 0 but not sufficient to store the value of that argument. The provider's state will change to *T\_IDLE* and the information to be returned in *ret* will be discarded.

**TSYSERR** A system error has occurred during execution of this function.

**TADDRBUSY** The requested address is in use.

**TPROTO** This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t\_errno*).

### Related Information

- “xti.h” on page 56
- “t\_alloc() — Allocate a Library Structure” on page 1498
- “t\_close() — Close a Transport Endpoint” on page 1523
- “t\_open() — Establish a Transport Endpoint” on page 1594
- “t\_optmgmt() — Manage Options for a Transport Endpoint” on page 1596
- “t\_unbind() — Disable a Transport Endpoint” on page 1635



## tcdrain() — Wait Until Output Has Been Transmitted

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <termios.h>
```

```
int tcdrain(int fildev);
```

### General Description

The `tcdrain()` function waits until all output sent to *fildev* has actually been sent to the terminal device.

If `tcdrain()` is called from a background process group against the caller's controlling terminal, a SIGTTOU signal may be generated depending how the process is handling SIGTTOUs:

Processing for SIGTTOU	System Behavior
Default or signal handler	The SIGTTOU signal is generated, and the function is not performed. <code>tcdrain()</code> returns the value <code>-1</code> and sets <code>errno</code> to <code>EINTR</code> .
Ignored or blocked	The SIGTTOU signal is not sent, and the function continues normally.

### Returned Value

If successful, `tcdrain()` returns zero. If unsuccessful, it returns the value `-1` and sets `errno` to one of the following:

EBADF	<i>fildev</i> is not a valid open file descriptor.
EINTR	A signal interrupted <code>tcdrain()</code> .
EIO	The process group of the process issuing the function is an orphaned, background process group, and the process issuing the function is not ignoring or blocking SIGTTOU.
ENOTTY	<i>fildev</i> is not associated with a terminal.

### Example CBC3BT03

```
/* CBC3BT03 */
#define _POSIX_SOURCE
#include <termios.h>
#include <stdio.h>
#include <time.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
```

```

main() {
    char Master[]="/dev/ptyp0010";
    char Slave[]="/dev/ttyp0010";
    char text[]="text to be written to tty";
    char data[80];
    int master, slave;
    time_t T;

    if ((master = open(Master, O_RDWR|O_NONBLOCK)) < 0)
        perror("open() error for master tty");
    else {
        if ((slave = open(Slave, O_RDWR|O_NONBLOCK)) < 0)
            perror("open() error for slave tty");
        else {
            if (fork() == 0) {
                if (write(slave, text, strlen(text)+1) == -1)
                    perror("write() error");
                time(&T);
                printf("child has written to tty, tcdrain() started at %s",
                       ctime(&T));
                if (tcdrain(slave) != 0)
                    perror("tcdrain() error");
                time(&T);
                printf("tcdrain() returned at %s", ctime(&T));
                exit(0);
            }
            time(&T);
            printf("parent is starting nap at %s", ctime(&T));
            sleep(5);
            time(&T);
            printf("parent is done with nap at %s", ctime(&T));
            if (read(master, data, sizeof(data)) == -1)
                perror("read() error");
            else printf("read '%s' from the tty\n", data);
            sleep(5);
            close(slave);
        }
        close(master);
    }
}

```

**Output**

```

parent is starting nap at Fri Jun 16 12:46:28 1995
child has written to tty, tcdrain() started at Fri Jun 16 12:46:28 1995
parent is done with nap at Fri Jun 16 12:46:34 1995
read 'text to be written to tty' from the tty
tcdrain() returned at Fri Jun 16 12:46:34 1995

```

**Related Information**

- “tcflow() — Suspend or Resume Data Flow on a Terminal” on page 1509
- “tcflush() — Flush Input or Output on a Terminal” on page 1512
- “tcgetattr() — Get the Attributes for a Terminal” on page 1515
- “tcgetpgrp() — Get the Foreground Process Group ID” on page 1520
- “tcsetattr() — Set the Attributes for a Terminal” on page 1531
- “tcsendbreak() — Send a Break Condition to a Terminal” on page 1529
- “tcsetpgrp() — Set the Foreground Process Group ID” on page 1546

## tcflow() — Suspend or Resume Data Flow on a Terminal

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <termios.h>
```

```
int tcflow(int fildev, int action);
```

### General Description

Suspends or resumes transmission or reception of data on a terminal device.

*int fildev*     A file descriptor associated with a terminal device.

*int action*     Indicates the action you want to perform, represented by one of the following symbols defined in the `termios.h` include file:

Symbol	Meaning
TCOOFF	Suspends output.
TCOON	Resumes suspended output.
TCIOFF	Sends a STOP character to the terminal, to stop the terminal from sending any further input.
TCION	Sends a START character to the terminal, to tell the terminal that it can resume sending input.

If `tcflow()` is called from a background process group against the caller's controlling terminal, a SIGTTOU signal may be generated depending how the process is handling SIGTTOUs:

Processing for SIGTTOU	System Behavior
Default or signal handler	The SIGTTOU signal is generated, and the function is not performed. <code>tcflow()</code> returns <code>-1</code> and sets <code>errno</code> to <code>EINTR</code> .
Ignored or blocked	The SIGTTOU signal is not sent, and the function continues normally.

### Returned Value

If successful, `tcflow()` returns zero. If unsuccessful, it returns `-1` and sets `errno` to one of the following:

EBADF	<i>fildev</i> is not a valid open file descriptor.
EINTR	A signal interrupted <code>tcflow()</code> .

- EIO For either of the following reasons:
- TCIOFF or TCION was requested, but the other side of the pseudoterminal connection is closed.
  - The process group of the process issuing the function is an orphaned, background process group, and the process issuing the function is not ignoring or blocking SIGTTOU.
- EINVAL *action* had an incorrect value.
- ENOTTY *fildev* is not associated with a terminal.

## Example

### CBC3BT04

```

/* CBC3BT04
   This example suspends and then resumes transmission.
*/
#define _POSIX_SOURCE
#include <termios.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

main() {
    char Master[]="/dev/ptyp0010";
    char Slave[]="/dev/tty0010";
    char text[]="text to be written to tty";
    char data[80];
    int master, slave;

    if ((master = open(Master, O_RDWR|O_NONBLOCK)) < 0) {
        perror("open() error for master tty");
        exit(1);
    }
    if ((slave = open(Slave, O_RDWR|O_NONBLOCK)) < 0) {
        perror("open() error for slave tty");
        exit(1);
    }
    if (write(slave, text, strlen(text)+1) == -1) {
        perror("write() error");
        exit(1);
    }
    if (tcflow(slave, TCIOFF) != 0) {
        perror("tcflow() error");
        exit(1);
    }
    puts("output is suspended to tty");
    if (read(master, data, sizeof(data)) == -1)
        perror("read() error");
    else printf("read '%s' from the tty\n", data);
    if (tcflow(slave, TCOON) != 0) {
        perror("tcflow() error");
        exit(1);
    }
    puts("output is resumed to tty");
    if (read(master, data, sizeof(data)) == -1) {
        perror("read() error");
        exit(1);
    }
    printf("read '%s' from the tty\n", data);
    close(slave);
}

```

```
    close(master);  
}
```

### Output

```
output is suspended to tty  
read() error: Resource temporarily unavailable  
output is resumed to tty  
read 'text to be written to tty' from the tty
```

### Related Information

- “tcdrain() — Wait Until Output Has Been Transmitted” on page 1507
- “tcflush() — Flush Input or Output on a Terminal” on page 1512
- “tcgetattr() — Get the Attributes for a Terminal” on page 1515
- “tcgetpgrp() — Get the Foreground Process Group ID” on page 1520
- “tcsetattr() — Set the Attributes for a Terminal” on page 1531
- “tcsendbreak() — Send a Break Condition to a Terminal” on page 1529
- “tcsetpgrp() — Set the Foreground Process Group ID” on page 1546

## tcflush() — Flush Input or Output on a Terminal

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <termios.h>
```

```
int tcflush(int fildev, int where);
```

### General Description

Flushes input or output on a terminal.

*int fildev*; Indicates a file descriptor associated with a terminal device.

*int where*; Indicates whether the system is to flush input or output, represented by one of the following symbols defined in the `termios.h` header file.

#### Symbol      Meaning

TCIFLUSH      Flushes input data that has been received by the system but not read by an application.

TCOFLUSH      Flushes output data that has been written by an application but not sent to the terminal.

TCIOFLUSH      Flushes both input and output data.

If `tcflush()` is called from a background process group against the caller's controlling terminal, a SIGTTOU signal may be generated depending how the process is handling SIGTTOUs:

Processing for SIGTTOU	System Behavior
Default or signal handler	The SIGTTOU signal is generated, and the function is not performed. <code>tcflush()</code> returns the value <code>-1</code> and sets <code>errno</code> to <code>EINTR</code> .
Ignored or blocked	The SIGTTOU signal is not sent, and the function continues normally.

### Returned Value

If successful, `tcflush()` returns zero. If unsuccessful, it returns the value `-1` and sets `errno` to one of the following:

`EBADF`      *fildev* is not a valid open file descriptor.

`EINTR`      A signal interrupted `tcflush()`.

`EINVAL`      *where* has an incorrect value.

- EIO           The process group of the process issuing the function is an orphaned, background process group, and the process issuing the function is not ignoring or blocking SIGTTOU.
- ENOTTY       *fildev* is not associated with a terminal.

## Example

### CBC3BT05

```

/* CBC3BT05
   This example flushes a string.
*/
#define _POSIX_SOURCE
#include <termios.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

main() {
    char Master[]="/dev/ptyp0010";
    char Slave[]="/dev/ttyp0010";
    char text1[]="string that will be flushed from buffer";
    char text2[]="string that will not be flushed from buffer";
    char data[80];
    int master, slave;

    if ((master = open(Master, O_RDWR|O_NONBLOCK)) < 0) {
        perror("open() error for master tty");
        exit(1);
    }
    if ((slave = open(Slave, O_RDWR|O_NONBLOCK)) < 0) {
        perror("open() error for slave tty");
        exit(1);
    }
    if (write(slave, text1, strlen(text1)+1) == -1) {
        perror("write() error");
        exit(1);
    }
    if (tcflush(slave, TCOFLUSH) != 0) {
        perror("tcflush() error");
        exit(1);
    }
    puts("first string is written and tty flushed");
    puts("now writing string that will not be flushed");
    if (write(slave, text2, strlen(text2)+1) == -1) {
        perror("write() error");
        exit(1);
    }
    if (read(master, data, sizeof(data)) == -1) {
        perror("read() error");
        exit(1);
    }
    printf("read '%s' from the tty\n", data);
    close(slave);
    close(master);
}

```

## Output

```

first string is written and tty flushed
now writing string that will not be flushed
read 'string that will not be flushed from buffer' from the tty

```

### **Related Information**

- “[tcdrain\(\)](#) — Wait Until Output Has Been Transmitted” on page 1507
- “[tcflow\(\)](#) — Suspend or Resume Data Flow on a Terminal” on page 1509
- “[tcgetattr\(\)](#) — Get the Attributes for a Terminal” on page 1515
- “[tcgetpgrp\(\)](#) — Get the Foreground Process Group ID” on page 1520
- “[tcsetattr\(\)](#) — Set the Attributes for a Terminal” on page 1531
- “[tcsendbreak\(\)](#) — Send a Break Condition to a Terminal” on page 1529
- “[tcsetpgrp\(\)](#) — Set the Foreground Process Group ID” on page 1546



## tcgetattr() — Get the Attributes for a Terminal

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <termios.h>
```

```
int tcgetattr(int fildes, struct termios *termptr);
```

### General Description

Gets a `termios` structure, which contains control information for a terminal associated with *fil*des. It stores that information in a memory location that *term*ptr points to. The contents of a `termios` structure are described in “`tcsetattr()` — Set the Attributes for a Terminal” on page 1531.

`tcgetattr()` can run in either a foreground or background process; however, if the process is in the background, a foreground process may subsequently change the attributes.

### Returned Value

If successful, `tcgetattr()` returns zero. If unsuccessful, it returns `-1` and sets `errno` to one of the following:

`EBADF`      *fil*des is not a valid open file descriptor.  
`ENOTTY`     The file associated with *fil*des is not a terminal.

### Example

#### CBC3BT06

```
/* CBC3BT06
/* CBC3BT06
   This example provides information about the attributes.
*/
#define _POSIX_SOURCE
#include <termios.h>
#include <stdio.h>
#include <unistd.h>

main() {
    struct termios term;

    if (tcgetattr(STDIN_FILENO, &term) != 0)
        perror("tcgetattr() error");
    else {
        if (term.c_lflag & ECHO)
            puts("ECHO is set");
        else
            puts("ECHO is not set");
        printf("The end-of-file character is x'%02x'\n",
            term.c_cc[VEOF]);
    }
}
```

**Output**

ECHO is set

The end-of-file character is x'37'

**Related Information**

- “[cfgetispeed\(\)](#) — Determine the Input Baud Rate” on page 156
- “[cfgetospeed\(\)](#) — Determine the Output Baud Rate” on page 159
- “[cfsetispeed\(\)](#) — Set the Input Baud Rate in the Termios” on page 161
- “[cfsetospeed\(\)](#) — Set the Output Baud Rate in the Termios” on page 163
- “[tcdrain\(\)](#) — Wait Until Output Has Been Transmitted” on page 1507
- “[tcflow\(\)](#) — Suspend or Resume Data Flow on a Terminal” on page 1509
- “[tcflush\(\)](#) — Flush Input or Output on a Terminal” on page 1512
- “[tcgetpgrp\(\)](#) — Get the Foreground Process Group ID” on page 1520
- “[tcsetattr\(\)](#) — Set the Attributes for a Terminal” on page 1531
- “[tcsendbreak\(\)](#) — Send a Break Condition to a Terminal” on page 1529
- “[tcsetpgrp\(\)](#) — Set the Foreground Process Group ID” on page 1546

## \_\_tcgetcp() — Get Terminal Code Page Names

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 5.2.2

### Format

```
#define _OPEN_SYS_PTY_EXTENSIONS
#include <termios.h>
```

```
int __tcgetcp(int fildes, size_t termcplen, struct __termcp *termcpptr);
```

### General Description

The `__tcgetcp()` function gets the terminal session code page information contained in the `termcp` structure and the Code Page Change Notification (CPCN) capability for the terminal file.

The following arguments are used:

`fildes`        The file descriptor of the terminal for which you want to get the code page names and CPCN capability.

`termcplen`    The length of the passed `termcp` structure.

`termcpptr`    A pointer to a `__termcp` structure.

`__tcgetcp()` stores the `termcp` information in a memory location pointed to by *termcpptr*. The return value contains the CPCN capability. The following CPCN capabilities are defined:

### Symbol      Meaning

`_CPCN_NAMES`

Forward code page names only

Use the `__tcsetcp()` function to change the terminal session data conversion. The OS/390 UNIX pseudotty device driver supports this CPCN capability.

`_CPCN_TABLES`

Forward code page names and tables

Use the `__tcsettables()` function to change the terminal session data conversion. The OCS remote-tty device driver supports this CPCN capability.

In the returned `termcp` structure, if the `_TCCP_FASTP` bit is set then the data conversion that is specified by the source and target code page names can be performed locally by the data conversion application. This is valid any time that a table-driven conversion can be performed. For example, the data conversion point (application) could use the OS/390 UNIX `iconv()` service to build local data conversion tables and perform all data conversion using the local tables instead of using `iconv()` all in subsequent conversions. This provides for better-performing data conversion.

In the returned termcp structure, if the `_TCCP_BINARY` bit is set then no data conversion is being performed and the code page names contained in the termcp structure should be ignored.

`__tcgetcp()` can run in either a foreground or background process; however, if the process is in the background, a foreground process may subsequently change the terminal code pages.

## Returned Value

If successful, `__tcgetcp()` returns the termcp structure in a memory location pointed to by *termcp\_ptr*. The return value contains the CPCN capability.

If unsuccessful, `__tcgetcp()` returns `-1`, and returns the error value in `errno`. The following are the possible values of `errno`:

`EBADF`      *fildev* is not a valid open file descriptor.

`EINVAL`     The value of *termcplen* was invalid.

`ENODEV`    One of the following error conditions exist:

- The terminal device driver does not support CPCN functions.
- CPCN functions have not been enabled.

For an OS/390 UNIX pseudotty terminal device file, issue the `__tcsetcp()` function against the master pty first to enable CPCN support.

`ENOTTY`     The file associated with *fildev* is not a terminal device.

## Example

The following example retrieves the current code pages used in the data conversion and CPCN capability. Here, the `__tcgetcp()` function is issued against a session using a pty terminal device; ISO8859-1 and IBM-1047 code pages are being used.

```
#define _OPEN_SYS_PTY_EXTENSIONS
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <termios.h>

void main(void)
{
    struct __termcp mytermcp;
    int rv;
    int cterm_fd;

    if ((cterm_fd = open("/dev/tty",O_RDWR)) == -1)
        printf("No controlling terminal established.\n");
    else {
        if ((rv = __tcgetcp(cterm_fd,sizeof(mytermcp),&mytermcp))== -1)
            perror("__tcgetcp() error");
        else {
            if (_CPCN_NAMES == rv)
                printf("Forward Code Page Names Only.\n");
            else
                printf("Forward Code Page Names and Tables.\n");
            if (_TCCP_BINARY == (mytermcp.__tccp_flags & _TCCP_BINARY))
```

```

        printf("Binary mode is in effect.\n");
    else {
        printf("ASCII code page name is %s.\n",
            mytermcp.__tccp_fromname);
        printf("EBCDIC code page name is %s.\n",
            mytermcp.__tccp_toname);
    }
}
close(cterm_fd);
}
} /* main */

```

**Output**

Forward code page names only.  
ASCII code page name is ISO8859-1.  
EBCDIC code page name is IBM-1047.

**Related Information**

- “termios.h” on page 51
- “\_\_tcsetcp() — Set Terminal Code Page Names” on page 1542
- “\_\_tcsettables() — Set Terminal Code Page Names and Conversion Tables” on page 1549

## tcgetpgrp() — Get the Foreground Process Group ID

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
pid_t tcgetpgrp(int fildes);
```

### General Description

Gets the process group ID (PGID) of the foreground process group associated with the terminal referred to by *fildes*. `tcgetpgrp()` can run from a background process, but the information may subsequently be changed by a process in the foreground process group.

### Returned Value

If successful, `tcgetpgrp()` returns the value of the foreground process group's PGID. If unsuccessful, it returns the value `-1` and sets `errno` to one of the following:

**EBADF**        *fildes* is not a valid open file descriptor.

**ENOTTY**      The calling process does not have a controlling terminal, or the file is not the controlling terminal.

### Example

#### CBC3BT07

```
/* CBC3BT07
   This example gets the foreground PGID.
*/
#define _POSIX_SOURCE
#include <termios.h>
#include <unistd.h>
#include <wait.h>
#include <stdio.h>

main() {
    pid_t pid;

    if ((pid = tcgetpgrp(STDOUT_FILENO)) < 0)
        perror("tcgetpgrp() error");
    else
        printf("the foreground process group id of stdout is %d\n",
              (int) pid);
}
```

### Output

```
the foreground process group id of stdout is 4063240
```

**Related Information**

- “unistd.h” on page 53
- “setpgid() — Set Process Group ID for Job Control” on page 1254
- “setsid() — Create Session, Set Process Group ID” on page 1268
- “tcdrain() — Wait Until Output Has Been Transmitted” on page 1507
- “tcflow() — Suspend or Resume Data Flow on a Terminal” on page 1509
- “tcflush() — Flush Input or Output on a Terminal” on page 1512
- “tcgetattr() — Get the Attributes for a Terminal” on page 1515
- “tcsetattr() — Set the Attributes for a Terminal” on page 1531
- “tcsendbreak() — Send a Break Condition to a Terminal” on page 1529
- “tcsetpgrp() — Set the Foreground Process Group ID” on page 1546

## tcgetsid() — Get Process Group ID for Session Leader for Controlling Terminal

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <termios.h>
pid_t tcgetsid(int fildes);
```

### General Description

The `tcgetsid()` obtains the process group ID of the session for which the terminal specified by *fildes* is the controlling terminal.

### Returned Value

Upon successful completion, `tcgetsid()` returns the process group ID associated with the terminal. Otherwise, a value of `(pid_t)-1` is returned and `errno` is set to indicate the error.

The `tcgetsid()` function will fail if:

- `EACCES`     The *fildes* argument is not associated with a controlling terminal.
- `EBADF`     The *fildes* argument is not a valid file descriptor.
- `ENOTTY`     The *fildes* associated with *fildes* is not a terminal.

### Related Information

- “termios.h” on page 51



## t\_close() — Close a Transport Endpoint

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_close(int fd);
```

### General Description

Notifies the transport provider that the user is finished with the transport endpoint specified by *fd*, and frees any local library resources associated with the endpoint. `t_close()` also closes the file associated with the transport endpoint.

`t_close()` should be called from the `T_UNBND` state. However, `t_close()` does not check state information, so it may be called from any state to close a transport endpoint. If this occurs, the local library resources associated with the endpoint are freed automatically. In addition, `close()` is issued for that file descriptor. The `close()` will be abortive if there are no other descriptors in this, or in another process which references the transport endpoint, and in this case will break any transport connection that may be associated with that endpoint.

A `t_close()` issued on a connection endpoint may cause data previously sent, or data not yet received, to be lost. It is the responsibility of the transport user to ensure that data is received by the remote peer.

### Valid States

All - except for `T_UNINIT`

### Returned Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and `t_errno` is set to indicate an error. On failure, `t_errno` is set to the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI ( <code>t_errno</code> ).

### Related Information

- “xti.h” on page 56
- “t\_getstate() — Get the Current State” on page 1569
- “t\_open() — Establish a Transport Endpoint” on page 1594
- “t\_unbind() — Disable a Transport Endpoint” on page 1635

## t\_connect() — Establish a Connection with Another Transport User

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_connect(int fd, struct t_call *call, struct t_call *rcvcall);
```

### General Description

Enables a transport user to request a connection to the specified destination transport user. This function can only be issued in the T\_IDLE state. The parameter *fd* identifies the local transport endpoint where communication will be established, while *sndcall* and *rcvcall* point to a *t\_call* structure which contains the following members:

```
struct netbuf  addr;
struct netbuf  opt;
struct netbuf  udata;
int            sequence;
```

The parameter *sndcall* specifies information needed by the transport provider to establish a connection, and *rcvcall* specifies information that is associated with the newly established connection.

In *sndcall*, *addr* specifies the protocol address of the destination transport user. *opt* presents any protocol-specific information that might be needed by the transport provider. *udata* points to optional user data that may be passed to the destination transport user during connection establishment. *sequence* has no meaning for this function.

On return, in *rcvcall*, *addr* contains the protocol address associated with the responding transport endpoint. *opt* represents any protocol-specific information associated with the connection. *udata* points to optional user data that may be returned by the destination transport user during connection establishment. *sequence* has no meaning for this function.

The *opt* argument permits users to define the options that may be passed to the transport provider. See the discussion of supported options in *t\_optmgmt()*. The user may choose not to negotiate protocol options by setting the *len* field of *opt* to zero. In this case, the provider may use default options.

If used, *sndcall->opt.buf* must point to a buffer with the corresponding options. The *maxlen* and *buf* fields of the netbuf structure pointed by *rcvcall->addr* and *rcvcall->opt* must be set before the call.

Since passing of userdata over a connection request is not supported under TCP, the *udata* argument is always meaningless.

On return, the *addr*, *opt* and *udata* fields of *rcvcall* will be updated to reflect values associated with the connection. Thus, the *maxlen* field of each argument must be

set before issuing this function to indicate the maximum size of the buffer for each. However, *rcvcall* may be a null pointer, in which case no information is given to the user on return from `t_connect()`.

By default, `t_connect()` executes in synchronous mode, and will wait for the destination user's response before returning control to the local user. A successful return (that is, return value of zero) indicates that the requested connection has been established. However, if `O_NONBLOCK` is set (via `t_open()` or `fcntl()`), `t_connect()` executes in asynchronous mode. In this case, the call will not wait for the remote user's response, but will return control immediately to the local user and return -1 with *t\_errno* set to `TNODATA` to indicate that the connection has not yet been established. In this way, the function simply initiates the connection establishment procedure by sending a connect request to the destination transport user. The `t_rcvconnect()` function is used in conjunction with `t_connect()` to determine the status of the requested connection.

When a synchronous `t_connect()` call is interrupted by the arrival of a signal, the state of the corresponding transport endpoint is `T_OUTCON`, allowing a further call to either `t_rcvconnect()`, `t_rcvdis()` or `t_snddis()`.

### Valid States

`T_IDLE`

### Returned Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *t\_errno* is set to indicate an error. On failure, *t\_errno* is set to one of the following:

<code>TBADF</code>	The specified file descriptor does not refer to a transport endpoint.
<code>TOUTSTATE</code>	The function was issued in the wrong sequence.
<code>TNODATA</code>	<code>O_NONBLOCK</code> was set, so the function successfully initiated the connection establishment procedure, but did not wait for a response from the remote user.
<code>TBADADDR</code>	The specified protocol address was in an incorrect format or contained illegal information.
<code>TBADOPT</code>	The specified protocol options were in an incorrect format or contained illegal information.
<code>TBADDATA</code>	The amount of user data specified was not within the bounds allowed by the transport provider.
<code>TACCES</code>	The user does not have permission to use the specified address or options.
<code>TBUFOVFLW</code>	The number of bytes allocated for an incoming argument ( <i>maxlen</i> ) is greater than 0 but not sufficient to store the value of that argument. If executed in synchronous mode, the provider's state, as seen by the user, changes to <code>T_DATAXFER</code> , and the information to be returned in <i>rcvcall</i> is discarded.

TLOOK	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TSYSERR	A system error has occurred during execution of this function.
TADDRBUSY	This transport provider does not support multiple connections with the same local and remote addresses. This error indicates that a connection already exists.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI ( <i>t_errno</i> ).

### Related Information

- “xti.h” on page 56
- “t\_accept() — Accept a Connect Request” on page 1493
- “t\_alloc() — Allocate a Library Structure” on page 1498
- “t\_getinfo() — Get Protocol-specific Service Information” on page 1566
- “t\_listen() — Listen for a Connect Indication” on page 1577
- “t\_open() — Establish a Transport Endpoint” on page 1594
- “t\_optmgmt() — Manage Options for a Transport Endpoint” on page 1596
- “t\_rcvconnect() — Receive the Confirmation from a Connect Request” on page 1607

## tcperror() — Print the Error Messages of a Socket Function

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	

### Format

```
#define _OPEN_SYS_SOCKET_EXT
#include <sys/socket.h>
#include <stdio.h>
#include <errno.h>
void tcperror(const char *s);
```

### General Description

When a socket call produces an error, the call returns a negative value and the variable `errno` is set to an error value found in `ERRNO.H`. The `tcperror()` call prints a short error message describing the last error that occurred. If `s` is non-NULL, `tcperror()` prints the string `s` followed by a colon, followed by a space, followed by the error message, and terminated with a new-line character. If `s` is NULL or points to a NULL string, only the error message and the new-line character are output.

The `tcperror()` function is equivalent to the `perror()` function in UNIX.

### Parameter Description

`s` A NULL or NULL-terminated character string.

### Returned Value

There is no return value.

### Example

The following are examples of the `tcperror()` call.

Example 1:

```
if ((s=socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
    tcperror("socket()");
    exit(2);
}
```

If the `socket()` call produces the error `ENOMEM`, `socket()` returns a negative value and `errno` is set to `ENOMEM`. When `tcperror()` is called, it prints the string:

```
socket(): not enough storage (ENOMEM)
```

Example 2:

```
if ((s=socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    tcperror(NULL);
```

If the `socket()` call produces the error `ENOMEM`, `socket()` returns a negative value and `errno` is set to `ENOMEM`. When `tcperror()` is called, it prints the string:

```
Not enough storage (ENOMEM)
```

**Related Information**

- “sys/socket.h” on page 48
- “perror() — Print Error Message” on page 903

tcsendbreak() — Send a Break Condition to a Terminal

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

Format

```
#define _POSIX_SOURCE
#include <termios.h>
```

```
int tcsendbreak(int fildes, int duration);
```

General Description

Sends a break condition to a terminal (indicated by *fildes*) that is using asynchronous serial data transmission. `tcsendbreak()` sends a continuous stream of zero bits for the specified *duration*. `tcsendbreak()` is the usual method of sending a BREAK on a line.

If `tcsendbreak()` is issued against a pseudoterminal, this function has no effect.

If `tcsendbreak()` is called from a background process group against the caller's controlling terminal, a SIGTTOU signal may be generated depending how the process is handling SIGTTOUs:

Processing for SIGTTOU	System Behavior
Default or signal handler	The SIGTTOU signal is generated, and the function is not performed. <code>tcsendbreak()</code> returns the value <code>-1</code> and sets <code>errno</code> to <code>EINTR</code> .
Ignored or blocked	The SIGTTOU signal is not sent, and the function continues normally.

Returned Value

If successful, `tcsendbreak()` returns zero. If unsuccessful, it returns the value `-1` and sets `errno` to one of the following:

- `EBADF` *fildes* is not a valid open file descriptor.
- `EINTR` A signal interrupted `tcsendbreak()`.
- `EIO` The process group of the process issuing the function is an orphaned, background process group, and the process issuing the function is not ignoring or blocking SIGTTOU.
- `ENOTTY` *fildes* is not associated with a terminal.

Example  
CBC3BT08

```
/* CBC3BT08
   This example breaks terminal transmission.
*/
#define _POSIX_SOURCE
#include <stdio.h>
```

```
#include <termios.h>
#include <unistd.h>

main() {
    if (tcsendbreak(STDIN_FILENO, 100) != 0)
        perror("tcsendbreak() error");
    else
        puts("break sent");
}
```

**Related Information**

- “tcdrain() — Wait Until Output Has Been Transmitted” on page 1507
- “tcflow() — Suspend or Resume Data Flow on a Terminal” on page 1509
- “tcflush() — Flush Input or Output on a Terminal” on page 1512
- “tcgetattr() — Get the Attributes for a Terminal” on page 1515
- “tcgetpgrp() — Get the Foreground Process Group ID” on page 1520
- “tcsetattr() — Set the Attributes for a Terminal” on page 1531
- “tcsetpgrp() — Set the Foreground Process Group ID” on page 1546



## tcsetattr() — Set the Attributes for a Terminal

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <termios.h>
```

```
int tcsetattr(int fd, int when, const struct termios *term_ptr);
```

### General Description

Changes the attributes associated with a terminal. New attributes are specified with a `termios` control structure. Programs should always issue a `tcgetattr()` first, modify the desired fields, and then issue a `tcsetattr()`. `tcsetattr()` should never be issued using a `termios` structure that was not obtained using `tcgetattr()`. `tcsetattr()` should use only a `termios` structure that was obtained by `tcgetattr()`.

*fd* Indicates an open file descriptor associated with a terminal.

*when* Indicates a symbol, defined in the `termios.h` header file, specifying when to change the terminal attributes:

#### Symbol      Meaning

TCSANOW    The change should take place immediately.

TCSADRAIN

The change should take place after all output written to *fd* has been read by the master pseudotermination. Use this value when changing terminal attributes that affect output.

TCSAFLUSH

The change should take place after all output written to *fd* has been sent; in addition, all input that has been received but not read should be discarded (flushed) before the change is made.

*\*term\_ptr*    A pointer to a `termios` control structure containing the desired terminal attributes.

A `termios` structure contains the following members:

`tcflag_t c_iflag`

Input modes. `tcflag_t` is defined in the `termios.h` header file. Each bit in `c_iflag` indicates an input attribute and is associated with a symbol defined in the `termios.h` include file. All symbols are bitwise distinct. Thus `c_iflag` is the bitwise inclusive OR of several of these symbols. Possible symbols are:

Symbol	Meaning
BRKINT	Indicates that an interrupt should be generated if the user types a BREAK.
ICRNL	Automatically converts input carriage returns to new-line (line-feed) characters before they are passed to the application that reads the input.
IGNBRK	<p>Ignores BREAK conditions. If this bit is set to 1, applications are not informed of any BREAK condition on the terminal; the setting of BRKINT has no effect.</p> <p>If IGNBRK is 0 but BRKINT is 1, BREAK flushes all input and output queues. In addition, if the terminal is the controlling terminal of a foreground process group, the BREAK condition generates a single SIGINT signal for that foreground process group.</p> <p>If both IGNBRK and BRKINT are 0, a BREAK condition is taken as the single input character null, if PARMRK is 0, and as the three input characters \377-null-null, if PARMRK is 1.</p>
IGNCR	<p>Ignores input carriage returns. If this bit is set to 1, the setting of ICRNL has no effect.</p> <p>If IGNCR is 0 and ICRNL is 1, input carriage returns are converted to newline characters. For OS/390 UNIX services "NL" or '\n' is the EBCDIC character NL.</p>
IGNPAR	Ignores input characters (other than BREAK) that have parity errors.
INLCR	Automatically converts input new-line (line-feed) characters to carriage returns before they are passed to the application that reads the input.
INPCK	Enables input parity checking. If this bit is set to 0, it allows output parity generation without input parity errors. The enabling of input parity checking is independent of the enabling of parity checking in the control modes field. (See the description of "tcflag_t c_cflag," which follows.) While the control modes may dictate that the hardware recognizes the parity bit, but the terminal special file does not check whether this bit is set correctly.
ISTRIP	<p>Strips valid input bytes to 7 bits. If this bit is set to 0, the complete byte is processed.</p> <p><b>Note:</b> Do not set this bit for pseudoterminals, since it will make the terminal unusable. If you strip the first bit off of EBCDIC characters, you destroy all printable EBCDIC characters.</p>
IUCLC	Map upper case to lower case on the received character. In locales other than the POSIX locale, the mapping is unspecified. Thus, this function only applies to the characters in the POSIX-portable character set that have lower case equivalents, namely the characters A-Z.

**Note:** This function is to be withdrawn in future versions of the standards.

IXANY	Enable any character to restart output. If IXOFF and IXANY are set and a previous STOP character has been received, then receipt of any input character will cause the STOP condition to be removed. For pseudoterminals, data in the output queue is passed to the application during master read() processing, and slave pseudoterminal writes are allowed to proceed. The character which caused the output to restart is also processed normally as well (unless it is a STOP character).
IXOFF	<p>Enables start/stop input control. If this bit is set to 1, the system attempts to prevent the number of bytes in the input queue from exceeding the MAX_INPUT value. It sends one or more STOP characters to the terminal device when the input queue is in danger of filling up. The character used as the STOP character is dictated by the c_cc member of the termios structure. It is intended to tell the terminal to stop sending input for a while. The system transmits one or more START characters when it appears that there is space in the input queues for more input. Again, the character used as the START character is dictated by the c_cc member. It is intended to tell the terminal that it can resume transmission of input.</p> <p><b>Note:</b> Do not use IXOFF while in DBCS mode. If you intersperse STOP and START characters inside DBCS data while using IXOFF, you could corrupt output data,</p>
IXON	<p>Enables start/stop output control. If the system receives a STOP character as input, it will suspend output on the associated terminal until a START character is received. An application reading input from the terminal does not see STOP or START characters; they are intercepted by the system, which does all the necessary processing.</p> <p>If IXON is 0, any STOP or START characters read are passed on as input to an application reading from the terminal.</p>
PARMRK	<p>Marks characters with parity errors. If this bit is set to 1 and IGNPAR is 0, a byte with a framing or parity error is sent to the application as the characters \377 and null, followed by the data part of the byte that had the parity error. If ISTRIP is 0, a valid input character of \377 is sent as a pair of characters \377, \377 to avoid ambiguity.</p> <p>If both PARMRK and IGNPAR are 0, a character with a framing or parity error is sent to the application as null.</p>

tcflag\_t c\_oflag

Output modes. Each bit in c\_oflag indicates an output attribute, and is associated with a symbol defined in the termios.h header file. Thus

c\_oflag is the bitwise inclusive OR of a number of these symbols.  
Possible symbols are:

Symbol	Meaning
OPOST	Modifies lines of text in an implementation-defined way to appear appropriately on the terminal device. If this bit is set to 0, characters that an application puts out are sent without change.
OLCUC	If OPOST and OLCUC are set, then map lower-case to upper-case on the output. In locales other than the POSIX locale, the mapping is unspecified. Thus, this function only applies to the characters in the POSIX-portable character set that have upper-case equivalents, namely the characters a-z.  <b>Note:</b> This function is to be withdrawn in future versions of the standards.
ONLCR	If OPOST and ONLCR are set, the NL character is transmitted as the CR-NL character pair.
OCRNL	If OPOST and OCRNL are set, the CR character is transmitted as the NL character.
ONOCR	If OPOST and ONOCR are set, no CR character is transmitted if the current column is zero.
ONLRET	If OPOST and ONLRET are set, the NL character does the carriage return function; the column pointer is set to 0. If OPOST is set and ONLRET is not set, then the NL does the line-feed function; the column pointer is unchanged.
OFILL	Fill characters are used for delay instead of using a timed delay.
OFDEL	The fill character is DEL. If OFILL is not set, then the fill character is NUL.
NLDLY	Delay associated with new-line character. NL0 No delay. NL1 0.10 seconds delay. If ONLRET is set, then carriage-return delays are used instead of newline delays. If OFILL is set, then two fill characters are transmitted.
CRDLY	Delay associated with carriage-return character. CR0 No delay. CR1 Delay dependent on column position, or if OFILL is set then two fill characters are transmitted. CR2 0.10 seconds delay, or if OFILL is set then four fill characters are transmitted. CR3 0.15 seconds delay.

TABDLY	Delay associated with tab character.
TAB0	No horizontal tab processing.
TAB1	Delay dependent on column position, or if OFILL is set then two fill characters are transmitted.
TAB2	0.10 seconds delay, or if OFILL is set then two fill characters are transmitted.
TAB3	Tabs are expanded into spaces.
BSDLY	Delay associated with back-space character.
BS0	No delay.
BS1	0.05 seconds delay, or if OFILL is set then one fill character is transmitted.
VTDLY	Delay associated with vertical-tab processing.
VT0	No delay.
VT1	2 seconds delay.
FFDLY	Delay associated with form-feed processing.
FF0	No delay.
FF1	2 seconds delay.

tcflag\_t c\_cflag

Control modes. Each bit in `c_cflag` indicates a control attribute and is associated with a symbol defined in the `termios.h` header file. Thus `c_cflag` is the bitwise inclusive OR of several of these symbols. Possible symbols are:

Symbol	Meaning
--------	---------

CLOCAL	Ignores modem status lines. A call to <code>open()</code> returns immediately without waiting for a modem connection to complete. If this bit is set to 0, modem status lines are monitored and <code>open()</code> waits for the modem connection.
CREAD	Enables reception. If this bit is set to 0, no input characters are received from the terminal.  Using OS/390 UNIX pseudoterminal support, this bit is always enabled and set to 1.
CSIZE	Is a collection of bits indicating the number of bits per byte (not counting the parity bit, if any). These bits specify byte size for both transmission and reception. Possible settings of CSIZE are given with the following symbols:  CS5 - 5 bits per byte CS6 - 6 bits per byte CS7 - 7 bits per byte CS8 - 8 bits per byte

Using OS/390 UNIX services pseudoterminal support, all values are accepted, but CSIZE is changed to CS8.  
Using OS/390 UNIX services Outboard Communications Server (OCS) support, the specified value is used.

CSTOPB	<p>Sends two stop bits when necessary. If CSTOPB is 0, only one stop bit is used.</p> <p>Using OS/390 UNIX services pseudoterminal support, this bit is always 0. Using OS/390 UNIX services OCS support, the specified value is used.</p>
HUPCL	<p>Lowers the modem control lines for a port when the last process that has the port open closes the port (or the process ends). In other words, this tells the system to hang up when all relevant processes have finished using the port.</p> <p>For pseudoterminals HUPCL controls what happens when the slave pseudoterminals is closed. If HUPCL is set when the last file descriptor for the slave pseudoterminal is closed, then the slave pseudoterminal cannot be re-opened. The master terminal has to be closed and re-opened before the pair can be used again.</p>
PARENB	<p>Enables parity generation and detection. A parity bit is added to each character on output, and expected from each character on input.</p> <p>Under OS/390 UNIX services, if this bit is set to 1 in a request, it is ignored. It is always set to 0. Using OS/390 UNIX services OCS support, the specified value is used.</p>
PARODD	<p>Indicates odd parity (when parity is enabled). If PARODD is 0, even parity is used (when parity is enabled).</p> <p>Under OS/390 UNIX services, if this bit is set to 1 in a request, it is ignored. It is always set to 0. Using OS/390 UNIX services OCS support, the specified value is used.</p>

If the object for which the control modes are set is not an asynchronous serial connection, some bits may be ignored. For example, on a network connection, it may not be possible to set the baud rate.

#### tcflag\_t c\_lflag

Local modes. Each bit in `c_lflag` indicates a local attribute, and is associated with a symbol defined in the `termios.h` include file. Thus `c_lflag` is the bitwise inclusive OR of a number of these symbols. Possible symbols are:

Symbol	Meaning
ECHO	Echoes input characters back to the terminal. If this bit is 0, input characters are not echoed.
ECHOE	Echoes the ERASE character as an error-correcting backspace. When the user inputs an ERASE character, the terminal erases the last character in the current line from the display (if possible). The character used as the ERASE character is dictated by the <code>c_cc</code> member of the <code>termios</code> structure. ECHOE has an effect only if the ICANON bit is 1.

ECHOK	Either causes the terminal to erase the line from the display, or echoes the KILL character followed by an <code>\n</code> character. ECHOK has an effect only if the ICANON bit is set to 1.
ECHONL	Echoes the new-line (line-feed) character <code>'\n'</code> even if the ECHO bit is off. ECHONL has an effect only if the ICANON bit is set to 1.
ICANON	<p>Enables canonical input processing, also called <i>line mode</i>. Input is not delivered to the application until an entire line has been input. The end of a line is indicated by a new-line, end-of-file, or EOL character (where the character used as the EOL character is directed by the <code>c_cc</code> member of the <code>termios</code> structure [described shortly]). Canonical input processing uses the ERASE character to erase a single input character, and the KILL character to erase an entire line. The <code>MAX_CANON</code> value specifies the maximum number of bytes in an input line in canonical mode.</p> <p>If ICANON is 0, read requests take input directly from the input queue; the system does not wait for the user to enter a complete line. This is called <i>noncanonical mode</i>. ERASE and KILL characters are not handled by the system but passed directly to the application. See also the descriptions of MIN and TIME in the <code>c_cc</code> member.</p>
IEXTEN	<p>Enables extended implementation-defined functions. These are not defined, and IEXTEN is always set to 0.</p> <p>If the ERASE, KILL or EOF character is preceded by a backslash character, the special character is placed in the input queue without doing the "special character" processing and the backslash is discarded.</p>
ISIG	<p>If ISIG is set to 1, signals are generated if special control characters are entered. SIGINT is generated if INTR is entered; SIGQUIT is generated if QUIT is entered; and SIGTSTP is generated if SUSP is entered and job control is supported. The special control characters are controlled by the <code>c_cc</code> member.</p> <p>If ISIG is 0, the system does not generate signals when these special control characters are entered.</p>
NOFLSH	If this bit is set to 1, the system does not flush the input and output queues if a signal is generated by one of the special characters described in ISIG above. If NOFLSH is set to 0, the queues are flushed if one of the special characters is found.
TOSTOP	If this bit is set to 1, a SIGTTOU signal is sent to the process group of a process that tries to write to a terminal when it is not in the terminal's foreground process group. However, if the process that tries to write to the terminal is blocking or ignoring SIGTTOU signals, the system does not raise the SIGTTOU signal.

If TOSTOP is 0, output from background processes is output to the current output stream, and no signal is raised.

**XCASE** Do canonical lower and canonical upper presentation. In locales other than the POSIX locale, the effect is unspecified. XCASE set by itself makes all upper-case letters on input and output be preceded by a "\" character.

Some terminals can generate lower-case characters, but can display only upper-case characters. For these terminals, XCASE would be used by itself. Other terminals cannot generate lower-case characters either. For these terminals, XCASE would be used with IUCLC to generate lower-case characters when characters are typed without the backslash, and upper-case characters when the typed character is preceded by a backslash.

If a terminal can generate only upper-case characters, but can display either upper or lower-case, then XCASE would be used with OLCUC.

**Note:** This function is to be withdrawn in a future version of the standards.

`cc_t c_cc[NCCS]`

Control characters. This is an array of characters that may have special meaning for terminal handling. You can access characters in this array using subscripts that are symbols defined in the `termios.h` header file. For example, the STOP character is given by `c_cc[VSTOP]`. Possible subscript symbols are:

Symbol	Meaning
VEOF	Gives the end-of-file character EOF. It is recognized only in canonical (line) mode. When this is found in input, all bytes waiting to be read are immediately passed to the application without waiting for the end of the line. The EOF character itself is discarded. If EOF occurs at the beginning of a line, the read function that tries to read that line receives an end-of-file indication. Note that EOF results in end-of-file only if it is at the beginning of a line; if it is preceded by one or more characters, it indicates only end-of-line.
VEOL	Gives the end-of-line character EOL. It is recognized only in canonical (line) mode. This is an alternate character for marking the end of a line (in addition to the new line <code>\n</code> ).
VERASE	Gives the ERASE character. It is recognized only in canonical (line) mode. It deletes the last character in the current line. It cannot delete beyond the beginning of the line.
VINTR	Gives the interrupt character INTR. It is recognized only if ISIG is set to 1 in <code>c_lflag</code> . If the character is received, the system sends a SIGINT signal to all the processes in



	the foreground process group that has this device as its controlling terminal.
VKILL	Gives the KILL character. It is recognized only in canonical (line) mode. It deletes the entire contents of the current line.
VMIN	<p>Gives the MIN value for noncanonical mode processing.</p> <p>This is the minimum number of bytes that a call to read should return in noncanonical mode; it is not used in canonical mode.</p> <p>If both MIN and TIME are greater than zero, read returns when MIN characters are available or when the timer associated with TIME runs out (whichever comes first). The timer starts running as soon as a single character has been entered; if there is already a character in the queue when read is called, the timer starts running immediately.</p> <p>If MIN is greater than zero and TIME is zero, read waits for MIN characters to be entered, no matter how long that takes.</p> <p>If MIN is zero and TIME is greater than zero, read returns when the timer runs out or when a single character is received (whichever comes first). read returns either one character (if one is received) or zero (if the timer runs out). The timer starts running as soon as read is called. (Contrast this with the case where MIN and TIME are both greater than zero, and the timer starts only when a character is received.)</p> <p>If both MIN and TIME are zero, read returns immediately from every call. It returns the number of bytes that are immediately available, up to the maximum specified in the call to read.</p>
VQUIT	Gives the quit character QUIT. It is recognized only if ISIG is set to 1 in c_lflag. If the character is received, the system sends a SIGQUIT signal to all the processes in the foreground process group that has this device as its controlling terminal.
VSUSP	Gives the suspend character SUSP. It is recognized only if ISIG is set to 1 in c_lflag. If the character is received, the system sends a SIGTSTP signal to all the processes in the foreground process group that has this device as its controlling terminal.
VTIME	Gives the TIME value, used in noncanonical mode in connection with MIN. It expresses a time in terms of tenths of a second.
VSTOP	Gives the STOP character. You can use this to suspend output temporarily when IXON is set to 1 in c_iflag. Users can enter the STOP character to prevent output from running off the top of a display screen.

**VSTART** Gives the START character. You can use this to resume suspended output when IXON is set to 1 in `c_iflag`.

When `tcsetattr()` is called from a background session against a controlling terminal, SIGTTOU processing is as follows:

Processing for SIGTTOU	Expected Behavior
Default or signal handler	The SIGTTOU signal is generated, and the function is not performed. <code>tcsetattr()</code> returns <code>-1</code> and sets <code>errno</code> to <code>EINTR</code> .
Ignored or blocked	The SIGTTOU signal is not sent, and the function continues normally.

## Returned Value

If successful, `tcsetattr()` returns zero. If unsuccessful, it returns the value `-1` and sets `errno` to one of the following:

**EBADF** *fildev* is not a valid open file descriptor.

**EINTR** A signal interrupted `tcsetattr()`.

**EINVAL** *when* is not a recognized value, or some entry in the supplied `termios` structure had an incorrect value.

**EIO** The process group of the process issuing the function is an orphaned, background process group, and the process issuing the function is not ignoring or blocking SIGTTOU.

**ENOTTY** *fildev* is not associated with a terminal.

## Example CBC3BT09

```
/* CBC3BT09
   The following attributes changes the terminal attributes.
*/
#define _POSIX_SOURCE
#include <termios.h>
#include <unistd.h>
#include <stdio.h>

main() {
    struct termios term1, term2;

    if (tcgetattr(STDIN_FILENO, &term1) != 0)
        perror("tcgetattr() error");
    else {
        printf("the original end-of-file character is x'%02x'\n",
            term1.c_cc[VEOF]);
        term1.c_cc[VEOF] = 'z';
        if (tcsetattr(STDIN_FILENO, TCSANOW, &term1) != 0)
            perror("tcsetattr() error");
        if (tcgetattr(STDIN_FILENO, &term1) != 0)
            perror("tcgetattr() error");
        else
            printf("the new end-of-file character is x'%02x'\n",
                term1.c_cc[VEOF]);
    }
}
```

**Output**

the original end-of-file character is x'37'  
 the new end-of-file character is x'a9'

**Related Information**

- “cfgetispeed() — Determine the Input Baud Rate” on page 156
- “cfgetospeed() — Determine the Output Baud Rate” on page 159
- “cfsetispeed() — Set the Input Baud Rate in the Termios” on page 161
- “cfsetospeed() — Set the Output Baud Rate in the Termios” on page 163
- “open() — Open a File” on page 872
- “read() — Read From a File or Socket” on page 1080
- “tcdrain() — Wait Until Output Has Been Transmitted” on page 1507
- “tcflow() — Suspend or Resume Data Flow on a Terminal” on page 1509
- “tcflush() — Flush Input or Output on a Terminal” on page 1512
- “tcgetattr() — Get the Attributes for a Terminal” on page 1515
- “tcgetpgrp() — Get the Foreground Process Group ID” on page 1520
- “tcsendbreak() — Send a Break Condition to a Terminal” on page 1529
- “tcsetpgrp() — Set the Foreground Process Group ID” on page 1546

## `__tcsetcp()` — Set Terminal Code Page Names

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 5.2.2

### Format

```
#define _OPEN_SYS_PTY_EXTENSIONS
#include <termios.h>
```

```
int __tcsetcp(int fildes, size_t termcplen, const struct __termcp *termcpptr);
```

### General Description

The `__tcsetcp()` function sets (or changes) the terminal session code page information contained in the `termcp` structure.

The following arguments are used:

`fildes`        The file descriptor of the terminal for which you want to get the code page names and CPCN capability.

`termcplen`    The length of the passed `termcp` structure.

`termcpptr`    A pointer to a `__termcp` structure.

Use the `__tcsetcp()` function to send new code page information to the data conversion point in order to change the data conversion environment for the terminal session. This function is used with terminal devices that support the “forward code page names only” Code Page Change Notification (CPCN) capability. The OS/390 UNIX pseudotty (pty) device driver supports this capability.

For terminal sessions that use the OS/390 UNIX pty device driver, the data conversion point is the application that uses the master pty device. An example data conversion point is the OS/390 UNIX **rlogin** server. Here, **rlogin** uses CPCN functions to determine the ASCII source and/or EBCDIC target code pages to use for the conversion of the terminal data. During its processing of the `__tcsetcp()` function, the pty device driver applies the `__termcp` structure once the pty outbound data queue is drained. When this occurs, the pty input data queue is also flushed and a `TIOCPKT_CHCP` packet exception event is generated if extended packet mode is enabled (`PKTXTND` is set in the `termios` structure) to notify the application using the master pty that the code page information has been changed. The master pty application can then use the `__tcgetcp()` function to retrieve the new code page information and establish a new data conversion environment.

The `__tcsetcp()` function is supported by both the master and slave pty device drivers, however, CPCN functions first must be enabled by the application that uses the master pty; enabling CPCN functions is performed by the system during the initial `__tcsetcp()` invocation against the master pty device. Once the `__tcsetcp()` function is performed against the master pty then it may be subsequently issued against the slave pty.

**Note:** The data conversion for an OS/390 UNIX terminal session is performed on a session (terminal file) basis. If you change the data conversion character-

istics for one file descriptor, the new data conversion will apply to all open file descriptors associated with this terminal file.

**Attention:** Use this service carefully. By changing the code pages for the data conversion you may cause unpredictable behavior in the terminal session if the actual data used for the session is not encoded to the specified source (ASCII) and target (EBCDIC) code pages.

A `__termcp` structure contains the following members:

`__tccp_flags`

Flags. The following symbols are defined as bitwise distinct values. Thus, `__tccp_flags` is the bitwise inclusive OR of these symbols:

Symbol	Meaning
--------	---------

<code>_TCCP_BINARY</code>	Use <code>_TCCP_BINARY</code> to notify the data conversion point to stop data conversion. If this flag is set, the source and target code page names ( <code>__tccp_fromname</code> and <code>__tccp_toname</code> respectively) are not changed from their current values.
<code>_TCCP_FASTP</code>	Use <code>_TCCP_FASTP</code> to indicate to the data conversion point (e.g. <b>rlogin</b> ) that the data conversion specified by the source and target code page names can be performed locally to the application. This is valid any time that a table-driven conversion can be performed. For example, the data conversion point (application) could use the OS/390 UNIX <code>iconv()</code> service to build the local data conversion tables and perform all data conversion using the local tables instead of using <code>iconv()</code> in subsequent conversions. This provides for better-performing data conversion.

`__tccp_fromname`

The source code page name; typically this is the ASCII code page name. `__tccp_fromname` is a null terminated string with a maximum length of `_TCCP_CPNAME_MAX`, including the null (`\00`) character.

`__tccp_fromname` is case sensitive.

`__tccp_toname`

The target code page name; typically this is the EBCDIC code page name. `__tccp_toname` is a null terminated string with a maximum length of `_TCCP_CPNAME_MAX`, including the null (`\00`) character.

`__tccp_toname` is case sensitive.

When `__tcsetcp()` is issued against the slave pty from a process in a background process group, SIGTTOU processing is as follows:

SIGTTOU Processing	Expected Behavior
Default or signal handler	The <b>SIGTTOU</b> signal is generated. The function is not performed. <code>__tcsetcp()</code> returns a <code>-1</code> and sets <code>errno</code> to <code>EINTR</code> .
Ignored or blocked	The <b>SIGTTOU</b> signal is not sent. The function continues normally.

## Returned Value

If successful, `__tcsetcp()` returns zero.

If unsuccessful, `__tcsetcp()` returns `-1`, and returns the error value in `errno`. The following are the possible values of `errno`:

- |        |   |
|--------|---|
| EBADF  | <i>fildev</i> is not a valid open file descriptor.  |
| EINTR  | A signal interrupted the call.  |
| EINVAL | The value of <i>termcplen</i> was invalid.  |
| EIO    | The process group of the process issuing the function is an orphaned, background process group, and the process issuing the function is not ignoring or blocking SIGTTOU.   |
| ENODEV | One of the following error conditions exist: <ul style="list-style-type: none"> <li>• CPCN functions have not been enabled.<br/>The <code>__tcsetcp()</code> function must be issued against the master pty before any CPCN function can be issued against the slave pty.</li> <li>• The terminal device driver does not support the “forward code page names only” CPCN capability.</li> </ul> |
| ENOTTY | The file associated with <i>fildev</i> is not a terminal device.  |

## Example

The following example retrieves the CPCN capability and code pages and then changes the ASCII code page to IBM-850.

```
#define _OPEN_SYS_PTY_EXTENSIONS
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <termios.h>

void main(void)
{
    struct __termcp mytermcp;
    int rv;
    int cterm_fd;

    if ((cterm_fd = open("/dev/tty", O_RDWR)) == -1)
        printf("No controlling terminal established.\n");
    else {
        if ((rv = __tcgetcp(STDIN_FILENO, sizeof(mytermcp), &mytermcp)) == -1)
            perror("__tcgetcp() error");
        else {
            if (rv == _CPCN_NAMES) {
                if (_TCCP_BINARY == (mytermcp.__tccp_flags & _TCCP_BINARY))
                    printf("Binary mode is in effect. No change made.\n");
                else {
```

```

        strcpy(mytermcp.__tccp_fromname,"IBM-850");
        if (__tcsetcp(STDOUT_FILENO,sizeof(mytermcp),&mytermcp)!=0)
            perror("__tcsetcp() error");
        else
            printf("ASCII code page changed to IBM-850.\n");
    } /*not binary mode */
} /* _CPCN_NAMES */
} /* __tcgetcp success */
close(cterm_fd);
} /* controlling terminal established */
} /* main */

```

**Output**

ASCII code page changed to IBM-850.

**Related Information**

- “termios.h” on page 51
- “\_\_tcgetcp() — Get Terminal Code Page Names” on page 1517
- “\_\_tcsettables() — Set Terminal Code Page Names and Conversion Tables” on page 1549

## tcsetpgrp() — Set the Foreground Process Group ID

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
int tcsetpgrp(int fildes, pid_t newid);
```

### General Description

Sets the process group ID (PGID) of the foreground process group associated with the terminal referred to by *fildes*. This terminal must be the controlling terminal of the process calling `tcsetpgrp()` and must be currently associated with the session of the calling process. *newid* must match a PGID of a process in the same session as the calling process.

After the PGID associated with the terminal is set, reads by the process group formerly associated with the terminal fail or cause the process group to stop from a SIGTTIN signal. Writes may also cause the process to stop (from a SIGTTOU signal), or they may succeed, depending on how `tcsetattr()` sets TOSTOP and the signal options for SIGTTOU.

*fildes* can be any of the descriptors representing the controlling terminal (such as standard input, standard output, and standard error), and the function affects future access from any file descriptor in use for the terminal. Consider using redirection when specifying the file descriptor.

If `tcsetpgrp()` is called from a background process group against the caller's controlling terminal, a SIGTTOU signal may be generated depending how the process is handling SIGTTOUs:

Processing for SIGTTOU	System Behavior
Default or signal handler	The SIGTTOU signal is generated, and the function is not performed. <code>tcsetpgrp()</code> returns the value <code>-1</code> and sets <code>errno</code> to <code>EINTR</code> .
Ignored or blocked	The SIGTTOU signal is not sent, and the function continues normally.

### Returned Value

If successful, `tcsetpgrp()` returns zero. If unsuccessful, it returns the value `-1` and sets `errno` to one of the following:

EBADF	<i>fildes</i> is not a valid open file descriptor.
EINTR	A signal interrupted the <code>tcsetpgrp()</code> function.
EINVAL	The <i>newid</i> value is not supported by this implementation.



- ENOTTY** The process calling `tcsetpgrp()` does not have a controlling terminal, or *fildev* is not associated with the controlling terminal, or the controlling terminal is no longer associated with the session of the calling process.
- EPERM** The *newid* value is supported by the implementation but does not match the process group ID of any process in the same session as the process calling `tcsetpgrp()`.

### Example

#### CBC3BT10

```

/* CBC3BT10
   This example changes the PGID.
*/
#define _POSIX_SOURCE
#include <termios.h>
#include <unistd.h>
#include <wait.h>
#include <stdio.h>
#include <signal.h>

main() {
    pid_t pid;
    int status;
    struct sigaction sa;

    if (fork() == 0) {
        sa.sa_handler=SIG_IGN;
        sigemptyset(&sa.sa_mask);
        sa.sa_flags=0;
        if (sigaction(SIGTTOU,&sa,0) < 0)
            perror("sigaction() set error");
        else {
            if ((pid = tcgetpgrp(STDOUT_FILENO)) < 0)
                perror("tcgetpgrp() error");
            else {
                printf("original foreground process group id of stdout was %d\n",
                    (int) pid);
                if (setpgid(getpid(), 0) != 0)
                    perror("setpgid() error");
                else {
                    printf("now setting to %d\n", (int) getpid());
                    if (tcsetpgrp(STDOUT_FILENO, getpid()) != 0)
                        perror("tcsetpgrp() error");
                    else if ((pid = tcgetpgrp(STDOUT_FILENO)) < 0)
                        perror("tcgetpgrp() error");
                    else
                        printf("new foreground process group id of stdout was %d\n",
                            (int) pid);
                }
            }
        }
    }
    else wait(&status);
}

```

### Output

```

original foreground process group id of stdout was 2228230
now setting to 2949128
new foreground process group id of stdout was 2949128

```

## **Related Information**

- “unistd.h” on page 53
- “tcdrain() — Wait Until Output Has Been Transmitted” on page 1507
- “tcflow() — Suspend or Resume Data Flow on a Terminal” on page 1509
- “tcflush() — Flush Input or Output on a Terminal” on page 1512
- “tcgetattr() — Get the Attributes for a Terminal” on page 1515
- “tcgetpgrp() — Get the Foreground Process Group ID” on page 1520
- “tcsetattr() — Set the Attributes for a Terminal” on page 1531
- “tcsendbreak() — Send a Break Condition to a Terminal” on page 1529

# \_\_tcsettables() — Set Terminal Code Page Names and Conversion Tables

## Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 5.2.2

## Format

```
#define _OPEN_SYS_PTY_EXTENSIONS
#include <termios.h>

int __tcsettables(int fildes, size_t termcplen,
                  const struct __termcp *termcpptr,
                  const char atoe[256],
                  const char etoe[256]);
```

## General Description

The \_\_tcsettables() function changes the data conversion environment for terminal sessions that support the “forward code page names and tables” Code Page Change Notification (CPCN) capability. The OCS remote-tty (rty) device driver supports this capability.

The following arguments are used:

- fildes** The file descriptor of the terminal for which you want to set the code page names and data conversion tables.
- termcplen** The length of the passed termcp structure.
- termcp**ptr A pointer to a \_\_termcp structure. A \_\_termcp structure contains the following members:

\_\_tccp\_flags

Flags. The following symbols are defined as bitwise distinct values. Thus, \_\_tccp\_flags is the bitwise inclusive OR of these symbols:

### Symbol      Meaning

\_TCCP\_BINARY

Use \_TCCP\_BINARY to notify the data conversion point to stop data conversion. If this flag is set the source and target code page names (\_\_tccp\_fromname and \_\_tccp\_toname respectively) are not changed, and the data conversion tables *atoe* and *etoe* are not used.

**Attention:** Use this option carefully. Once the data conversion is disabled the OS/390 shell cannot be used until the data conversion is re-enabled, using valid code pages for the terminal session.

`_TCCP_FASTP`

Use `_TCCP_FASTP` to indicate to the data conversion point that the data conversion specified by the source and target code page names can be performed locally by the application that performs the data conversion. This is valid any time that a table-driven conversion can be performed.

This value is not used by the OCS rty device driver and thus has no effect.

`__tccp_fromname`

The source code page name; typically this is the ASCII code page name. `__tccp_fromname` is a null terminated string with a maximum length of `_TCCP_CPNAME_MAX`, including the null (`\0`) character.

`__tccp_fromname` is case sensitive.

`__tccp_toname`

The target code page name; typically this is the EBCDIC code page name. `__tccp_toname` is a null terminated string with a maximum length of `_TCCP_CPNAME_MAX`, including the null (`\0`) character.

`__tccp_toname` is case sensitive.

`const char atoe[256]`

A 256-byte data conversion table for the source-to-target (ASCII-to-EBCDIC) data conversion. The byte offset into this table corresponds to the character code from the source (ASCII) code page. The data value at each offset is the “converted” target (EBCDIC) character code.

`const char etoa[256]`

A 256-byte data conversion table for the target-to-source (EBCDIC-to-ASCII) data conversion. The byte offset into this table corresponds to the character code from the target (EBCDIC) code page. The data value at each offset is the “converted” source (ASCII) character code.

**Note:** The data conversion for an OS/390 UNIX terminal session is performed on a session (terminal file) basis. If you change the data conversion characteristics for one file descriptor, the new data conversion will apply to all open file descriptors associated with this terminal file.

For terminal sessions that use the OCS rty device driver, the ASCII/EBCDIC data conversion is performed outboard by OCS on the AIX server system. Use the `__tcsettables()` function to specify new code pages and conversion tables to be used in the data conversion.

During its processing of the `__tcsettables()` function, the OCS rty device driver applies the new code page names once the outbound data queue is drained. When this occurs, the rty input data queue is also flushed and the new conversion environment takes effect.

OCS processing of the `atoe` and `etoa` arguments is as follows:

- If the code page names specified in the `__termcp` structure are for supported double-byte data conversion then the `atoe` and `etoe` arguments are not used. The following double-byte translation is supported for OCS sessions:
- If `__fromname` specifies **ISO8859-1** and `__toname` specifies **IBM-1047** then OCS uses its own data conversion tables and `atoe` and `etoe` arguments are not used.
- Otherwise the conversion tables in `atoe` and `etoe` are used.

**Attention:** Use this service carefully. By changing the code pages for the data conversion you may cause unpredictable behavior in the terminal session if the actual data used for the session is not encoded to the specified source (ASCII) and target (EBCDIC) code pages.

When `__tcsettables()` is issued from a process in a background process group, SIGTTOU is processing in this way:

SIGTTOU Processing	Expected Behavior
Default or signal handler	The <b>SIGTTOU</b> signal is generated. The function is not performed. <code>__tcsettables()</code> returns a <code>-1</code> and sets <code>errno</code> to <code>EINTR</code> .
Ignored or blocked	The <b>SIGTTOU</b> signal is not sent. The function continues normally.

## Returned Value

If successful, `__tcsettables()` returns zero.

If unsuccessful, `__tcsettables()` returns `-1`, and returns the error value in `errno`. The following are the possible values of `errno`:

- EBADF** *fd* is not a valid open file descriptor.
- EINTR** A signal interrupted the call.
- EINVAL** One of the following error conditions exists:
- The value of `termcplen` was invalid.
  - An invalid combination of multi-byte code page names was specified in the `__termcp` structure.
- One of the following applies:
- The source code page specified in `__tccp_fromname` specified a supported ASCII multi-byte code page and the `__tccp_toname` did not specify a supported EBCDIC multi-byte code page.
  - The target code page specified in `__tccp_toname` specified a supported EBCDIC multi-byte code page and the `__tccp_fromname` did not specify a supported ASCII multi-byte code page.
- EIO** The process group of the process issuing the function is an orphaned, background process group, and the process issuing the function is not ignoring or blocking SIGTTOU.

- ENODEV     The terminal device driver does not support the “forward code page names and tables” CPCN capability.
- ENOTTY     The file associated with *fildev* is not a terminal device.

**Example**

The following example retrieves the current code pages used in the data conversion and CPCN capability. The conversion tables using ASCII code page IBM-850 and the current EBCDIC code page are generated and exported to the data conversion point using `__tcsettables()`.

```
#define _OPEN_SYS_PTY_EXTENSIONS

#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <termios.h>
#include <iconv.h>

main()
{
    struct __termcp mytermcp;          /* local __termcp          */
    unsigned char *intabptr;            /* pointer to input table   */
    unsigned char *outtabptr;           /* pointer to output table  */
    unsigned char intab[256],
                                   atoe[256],
                                   etoa[256];          /* conversion tables       */
    iconv_t cd;                        /* conversion descriptor    */
    size_t inleft;                     /* number of bytes left in input */
    size_t outleft;                    /* number of bytes left in output */
    int i;                             /* loop variable           */
    int rv;                            /* return value            */
    int cterm_fd;                      /* file descriptor for controlling
                                   terminal                  */

    if ((cterm_fd = open("/dev/tty", O_RDWR)) == -1)
    {
        printf("No controlling terminal established. ");
        printf("Code pages were not changed.\n");
        exit(0);
    }
    if ((rv = __tcgetcp(cterm_fd, sizeof(mytermcp), &mytermcp)) == -1)
    {
        perror("__tcgetcp() error");
        exit(1);
    }

    if (_TCCP_BINARY == (mytermcp.__tccp_flags & _TCCP_BINARY))
    {
        printf("Binary mode is in effect. No change made.\n");
        exit(0);
    }

    if (rv == _CPCN_TABLES) {

        /* build ASCII -> EBCDIC conversion table */

        strcpy(mytermcp.__tccp_fromname, "IBM-850");
    }
}
```

```

if ((cd = iconv_open(mytermcp.__tccp_toname,
                    mytermcp.__tccp_fromname)) ==
    (iconv_t) (-1)) {
    fprintf(stderr, "Cannot open converter from %s to %s\n",
            mytermcp.__tccp_fromname, mytermcp.__tccp_toname);
    exit(1);
}

/* build input table with character values of 00 - FF */

for (i=0; i<256; i++ ) {
    intab[i] = (unsigned char) i;
} /* endfor */

inleft = 256;
outleft = 256;
intabptr = intab;
outtabptr = atoe;

/* build ASCII -> EBCDIC conversion table. */

rv = iconv(cd, &intabptr, &inleft, &outtabptr, &outleft);
if (rv == -1) {
    fprintf(stderr, "Error in building ASCII to EBCDIC table\n");
    exit(1);
}
iconv_close(cd);

/* build EBCDIC -> ASCII conversion table */

if ((cd = iconv_open(mytermcp.__tccp_fromname,
                    mytermcp.__tccp_toname)) ==
    (iconv_t) (-1)) {
    fprintf(stderr, "Cannot open converter from %s to %s\n",
            mytermcp.__tccp_toname, mytermcp.__tccp_fromname);
    exit(1);
}
inleft = 256;
outleft = 256;
intabptr = intab;
outtabptr = etoa;
rv = iconv(cd, &intabptr, &inleft, &outtabptr, &outleft);
if (rv == -1) {
    fprintf(stderr, "Error in building EBCDIC to ASCII table\n");
    exit(1);
}
iconv_close(cd);

/*
 * Change the data conversion to use IBM-850 as the ASCII source
 */

if (__tcsettables(cterm_fd, sizeof(mytermcp), &mytermcp,
    atoe, etoa) == -1) {
    perror("__tcsettables() error");
    exit(1);
} else {
    printf("Data conversion now using ASCII IBM-850\n");
}

```

```
    } /* endif */  
  } /* endif */  
  close(cterm_fd);  
} /* main */
```

### **Output**

Data conversion now using ASCII IBM-850.

### **Related Information**

- “termios.h” on page 51
- “\_\_tcgetcp() — Get Terminal Code Page Names” on page 1517
- “\_\_tcsetcp() — Set Terminal Code Page Names” on page 1542



## tdelete() — Binary Tree Delete

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <search.h>
```

```
void *tdelete(const void *key, void **rootp,
              int (*compar)(const void *, const void *));
```

### General Description

The `tdelete()` function deletes a node from a binary search tree. The arguments are the same as for the `tsearch()` function. The variable pointed to by `rootp` will be changed if the deleted node was the root of the tree. The `tdelete()` function returns a pointer to the parent of the deleted node, or a null pointer if the node is not found. If the deleted node was the root of the tree, the function returns a pointer to the deleted node, since it had no parent. It frees the storage for this node before returning, so the contents of storage at the returned address are unreliable in this case.

Comparisons are made with a user-supplied routine, the address of which is passed as the `compar` argument. This routine is called with two arguments, the pointers to the elements being compared. The user-supplied routine must return an integer less than, equal to or greater than 0, according to whether the first argument is to be considered less than, equal to or greater than the second argument. The comparison functions need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Threading Behavior: see “`tsearch()` — Binary Tree Search” on page 1617.

### Special Behavior for C++

Because C and C++ linkage conventions are incompatible, `tdelete()` cannot receive a C++ function pointer as the comparator argument. If you attempt to pass a C++ function pointer to `tdelete()`, the compiler will flag it as an error. You can pass a C or C++ function to `tdelete()` by declaring it as `extern "C"`.

### Returned Value

The `tdelete()` function returns a pointer to the parent of the deleted node, or a null pointer if the node is not found. A null pointer is returned if `rootp` is a null pointer on entry.

No errors are defined.

### **Related Information**

- “search.h” on page 40
- “tsearch() — Binary Tree Search” on page 1617
- “twalk() — Binary Tree Walk” on page 1636
- “tfind() — Binary Tree Find Node” on page 1563
- “bsearch() — Search Arrays” on page 137
- “hsearch() — Search Hash Tables” on page 647
- “lsearch() — Linear Search and Update” on page 775

## telldir() — Current Location of Directory Stream

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <dirent.h>
```

```
long telldir(DIR *dirp);
```

### General Description

The `telldir()` function obtains the current location associated with the directory stream specified by *dirp*.

If the most recent operation on the directory stream was a `seekdir()`, then the directory position returned from `telldir()` is the same as that supplied as a **loc** argument to `seekdir()`.

### Returned Value

If successful, `telldir()` returns the current location of the specified directory stream.

If the *dirp* argument supplied is null or invalid, `telldir()` returns `-1`, and returns the error value in `errno`. The following are the possible values of `errno`:

**EBADF**      The *dirp* argument was invalid.

### Related Information

- “dirent.h” on page 24
- “stdio.h” on page 43
- “sys/types.h” on page 49
- “closedir() — Close a Directory” on page 194
- “opendir() — Open a Directory” on page 877
- “readdir() — Read an Entry from a Directory” on page 1086
- “rewinddir() — Reposition a Directory Stream to the Beginning” on page 1141
- “seekdir() — Set Position of Directory Stream” on page 1158

## tempnam() — Generate a Temporary File Name

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	POSIX(ON) MVS 4.3

### Format

```
#define _XOPEN_SOURCE
#include <stdio.h>
```

```
char *tempnam(const char *dir, const char *pfx);
```

### General Description

The `tempnam()` function generates a pathname that may be used for a temporary file. If the environment variable `TMPDIR` is set, then the directory it specifies will be used as the directory part of the generated pathname if it is accessible. Otherwise, if the *dir* argument is non-null and accessible, it will be used in the generated pathname. Otherwise, the value of `{P_tmpdir}` defined in the `<stdio.h>` header is used as the directory component of the name. If that is inaccessible, then `/tmp` is used.

The *pfx* argument can be used to specify an initial component of the filename part of the pathname. It may be a null pointer or point to a string of up to five bytes to be used as the beginning of a filename.

The names generated are unique across processes and threads, and over time, so multiple threads should be able to each repeatedly call `tempnam()` and consistently obtain unique names.

This function is supported only in a POSIX program. See “OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions” on page 9 for more information.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

If successful, `tempnam()` allocates space for the generated name, copies the name into it, and returns a pointer to the name.

If unsuccessful, `tempnam()` returns a null pointer and returns the error value in `errno`. The following are the possible values of `errno`:

`ENOMEM` Insufficient storage is available.

`ENAMETOOLONG`

The generated name exceeded the maximum allowable pathname length.

**Related Information**

- “stdio.h” on page 43
- “fopen() — Open a File” on page 417
- “free() — Free a Block of Storage” on page 458
- “open() — Open a File” on page 872
- “tmpfile() — Create Temporary File” on page 1582
- “tmpnam() — Produce Temporary File Name” on page 1584
- “unlink() — Remove a Directory Entry” on page 1660

## terminate() — Terminate After Failures in C++ Error Handling

### Standards

Standards / Extensions	C or C++	Dependencies
	C++ only	

### Format

```
#include <terminate.h>
```

```
void terminate();
```

### General Description

Called when the C++ error handling mechanism fails. In turn, `terminate()` calls the function most recently specified by `set_terminate()`. If `set_terminate()` has not yet been called, then `terminate()` calls `abort()`.

In a multi-threaded environment, if a thread issues a throw, the stack is unwound until a matching catcher is found, up to and including the thread start routine. (The thread start routine is the function passed to `pthread_create()`.) If the exception is not caught, then the `terminate()` function is called, which in turn defaults to calling `abort()`, which in turn causes a SIGABRT signal to be generated to the thread issuing the throw. If the SIGABRT signal is not caught, the process is terminated. You can replace the default `terminate()` behavior for all threads in the process by using the `set_terminate()` function. One possible use of `set_terminate()` is to call a function which issues a `pthread_exit()`. If this is done, a throw of a condition by a thread that is uncaught results in thread termination but not process termination.

### Returned Value

Returns void.

Refer to *OS/390 C/C++ Language Reference* for more information about C++ exception handling including the `terminate()` function.

### Related Information

- “`abort()` — Stop a Program” on page 71
- “`unexpected()` — Handle Exception Not Listed in Exception Specification” on page 1654
- “`set_terminate()` — Register a Function for `terminate()`” on page 1276

## t\_error() — Produce Error Message

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_error(char *errmsg);
```

### General Description

Produces a language-dependent message on the standard error output which describes the last error encountered during a call to a transport function. The argument string *errmsg* is a user-supplied error message that gives context to the error.

The error message is written as follows: first (if *errmsg* is not a null pointer and the character pointed to by *errmsg* is not the null character) the string pointed to by *errmsg* followed by a colon and a space; then a standard error message string for the current error defined in *t\_errno*. If *t\_errno* has a value different from *TSYSERR*, the standard error message string is followed by a newline character. If, however, *t\_errno* is equal to *TSYSERR*, the *t\_errno* string is followed by the standard error message string for the current error defined in *errno* followed by a newline.

If the calling program is running in any one of the SAA, S370, C or POSIX locales, the error message string describing the value in *t\_errno* is identical to the comments following the *t\_errno* codes defined in *xti.h*. It is noteworthy that message numbers are not produced in this situation. The contents of the error message strings describing the value in *errno* are the same as those returned by the *strerror(3C)* function with an argument of *errno*.

The error number, *t\_errno*, is only set when an error occurs and it is not cleared on successful calls.

### Valid States

All - except for *T\_UNINIT*

### Returned Value

No errors are defined for the *t\_error()* function.

### Example

If a *t\_connect()* function fails on transport endpoint *fd2* because a bad address was given, the following call might follow the failure:

```
t_error("t_connect failed on fd2");
```

The diagnostic message to be printed would look like:

```
t_connect failed on fd2: incorrect addr format
```

where *incorrect addr format* identifies the specific error that occurred, and *t\_connect failed on fd2* tells the user which function failed on which transport endpoint.

**Related Information**

- “xti.h” on page 56



## tfind() — Binary Tree Find Node

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <search.h>
```

```
void *tfind(const void *key, void *const *rootp,
            int (*compar)(const void *, const void *));
```

### General Description

The `tfind()` function, like `tsearch()`, will search for a node in the tree, returning a pointer to it if found. However, if it is not found, the `tfind()` function will return a null pointer. The arguments for the `tfind()` function are the same as for the `tsearch()` function.

Comparisons are made with a user-supplied routine, the address of which is passed as the *compar* argument. This routine is called with two arguments, the pointers to the elements being compared. The user-supplied routine must return an integer less than, equal to or greater than 0, according to whether the first argument is to be considered less than, equal to or greater than the second argument. The comparison functions need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

Threading Behavior: see “`tsearch()` — Binary Tree Search” on page 1617.

### Special Behavior for C++

Because C and C++ linkage conventions are incompatible, `tfind()` cannot receive a C++ function pointer as the comparator argument. If you attempt to pass a C++ function pointer to `tfind()`, the compiler will flag it as an error. You can pass a C or C++ function to `tfind()` by declaring it as `extern "C"`.

### Returned Value

If the node is found, the `tfind()` function returns a pointer to it, otherwise it returns a null pointer. A null pointer is returned if *rootp* is a null pointer on entry.

No errors are defined.

### Related Information

- “`search.h`” on page 40
- “`tsearch()` — Binary Tree Search” on page 1617
- “`twalk()` — Binary Tree Walk” on page 1636
- “`tdelete()` — Binary Tree Delete” on page 1555
- “`bsearch()` — Search Arrays” on page 137
- “`hsearch()` — Search Hash Tables” on page 647
- “`lsearch()` — Linear Search and Update” on page 775

## t\_free() — Free a Library Structure

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_free(char *ptr, int struct_type);
```

### General Description

Frees memory previously allocated by `t_alloc()`. This function frees memory for the specified structure, and also frees memory for buffers referenced by the structure.

The argument *ptr* points to one of the seven structure types described for `t_alloc()` , and *struct\_type* identifies the type of that structure which must be one of the following:

T_BIND	struct	t_bind
T_CALL	struct	t_call
T_OPTMGMT	struct	t_optmgmt
T_DIS	struct	t_discon
T_UNITDATA	struct	t_unitdata
T_UDERROR	struct	t_uderr
T_INFO	struct	t_info

where each of these structures is used as an argument to one or more transport functions.

`t_free()` checks the *addr*, *opt* and *udata* fields of the given structure (as appropriate) and frees the buffers pointed to by the *buf* field of the netbuf structure. If *buf* is a null pointer, `t_free()` does not attempt to free memory. After all buffers are freed, `t_free()` frees the memory associated with the structure pointed to by *ptr*.

Undefined results occur if *ptr* or any of the *buf* pointers points to a block of memory that was not previously allocated by `t_alloc()` .

### Valid States

All - except for T\_UNINIT

### Returned Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *t\_errno* is set to indicate an error. On failure, *t\_errno* is set to the following:

TSYSERR A system error has occurred during execution of this function.

TNOSTRUCTYPE  
Unsupported *struct\_type* requested.

TPROTO This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t\_errno*).

**Related Information**

- “xti.h” on page 56
- “t\_alloc() — Allocate a Library Structure” on page 1498

## t\_getinfo() — Get Protocol-specific Service Information

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_getinfo(int fd, struct t_info *info);
```

### General Description

Returns the current characteristics of the underlying transport protocol and/or transport connection associated with file descriptor *fd*. The *info* pointer is used to return the same information returned by *t\_open()*, although not necessarily precisely the same values. This function enables a transport user to access this information during any phase of communication. This argument points to a *t\_info* structure which contains the following members:

```
long addr;      /* max size of the transport protocol address      */
long options;   /* max number of bytes of protocol-specific options */
long tsdu;      /* max size of a transport service data unit (TSDU) */
long etsdu;     /* max size of an expedited transport service      */
                /* data unit (ETSDU)                                */
long connect;   /* max amount of data allowed on connection        */
                /* establishment functions                          */
long discon;    /* max amount of data allowed on t_snddis()         */
                /* and t_rcvdis() functions                        */
long servtype;  /* sdis() functions                                */
long servtype;  /* service type supported by the transport provider */
long flags;     /* other info about the transport provider          */
```

The fields take on the following values:

addr	The size of a struct sockaddr_in is returned.
options	The value 304, which is the maximum number of bytes of options which can possibly be specified or requested, is returned.
tsdu	Zero is returned, indicating that the TCP transport provider does not support the concept of TSDUs.
etsdu	A value of -1 is returned, indicating that there is no limit on the size of an ETSDU.
connect	A value of -2 is returned, indicating that the TCP transport provider does not allow data to be sent with connection establishment functions.
discon	A value of -2 is returned, indicating that the transport provider does not allow data to be sent with the abortive release functions.
servtype	T_COTS is always returned, since this is the only service type supported.
flags	The T_SENDZERO bit is always set in this field, indicating that the TCP transport provider supports the sending of zero-length TSDUs.

If a transport user is concerned with protocol independence, the above sizes may be accessed to determine how large the buffers must be to hold each piece of information. Alternatively, the `t_alloc()` function may be used to allocate these buffers. An error results if a transport user exceeds the allowed data size on any function. The value of each field may change as a result of protocol option negotiation during connection establishment (the `t_optmgmt()` call has no affect on the values returned by `t_getinfo()`). These values will only change from the values presented to `t_open()` after the endpoint enters the `T_DATAXFER` state.

### Valid States

All - except for `T_UNINIT`

### Returned Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and `t_errno` is set to indicate an error. On failure, `t_errno` is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TSYSERR	A system error has occurred during execution of this function.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI ( <code>t_errno</code> ).

### Related Information

- “xti.h” on page 56
- “t\_alloc() — Allocate a Library Structure” on page 1498
- “t\_open() — Establish a Transport Endpoint” on page 1594

## t\_getprotaddr — Get the Protocol Addresses

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_getprotaddr(int fd, struct t_bind *boundaddr,
                  struct t_bind *peeraddr);
```

### General Description

Returns local and remote protocol addresses currently associated with the transport endpoint specified by *fd*. In *boundaddr* and *peeraddr* the user specifies *maxlen*, which is the maximum size of the address buffer, and *buf* which points to the buffer where the address is to be placed. On return, the *buf* field of *boundaddr* points to the address, if any, currently bound to *fd*, and the *len* field specifies the length of the address. If the transport endpoint is in the T\_UNBND state, zero is returned in the *len* field of *boundaddr*. The *buf* field of *peeraddr* points to the address, if any, currently connected to *fd*, and the *len* field specifies the length of the address. If the transport endpoint is not in the T\_DATAXFER state, zero is returned in the *len* field of *peeraddr*.

### Valid States

All - except for T\_UNINIT

### Returned Value

Upon successful completion, a value of zero is returned. Otherwise, a value of -1 is returned and *t\_errno* is set to indicate the error. On failure, *t\_errno* is set to one of the following:

TBADF      The specified file descriptor does not refer to a transport endpoint.

TBUFOVFLW

The number of bytes allocated for an incoming argument (*maxlen*) is greater than 0 but not sufficient to store the value of that argument.

TSYSERR    A system error has occurred during execution of this function.

TPROTO     This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t\_errno*).

### Related Information

- “xti.h” on page 56
- “t\_bind() — Bind an Address to a Transport Endpoint” on page 1504

## t\_getstate() — Get the Current State

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_getstate(int fd);
```

### General Description

Returns the current state of the provider associated with the transport endpoint specified by *fd*.

### Valid States

All - except for T\_UNINIT

### Returned Value

State is returned upon successful completion. Otherwise, a value of -1 is returned and *t\_errno* is set to indicate an error. The current state is one of the following:

T\_UNBND Unbound.

T\_IDLE Idle.

T\_OUTCON Outgoing connection pending.

T\_INCON Incoming connection pending.

T\_DATAXFER  
Data transfer.

If the provider is undergoing a state transition when *t\_getstate()* is called, the function will fail. On failure, *t\_errno* is set to one of the following:

TBADF The specified file descriptor does not refer to a transport endpoint.

TSTATECHNG  
The transport provider is undergoing a transient state change.

TSYSERR A system error has occurred during execution of this function.

TPROTO This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t\_errno*).

### Related Information

- “xti.h” on page 56
- “t\_open() — Establish a Transport Endpoint” on page 1594

## time() — Determine Current Time

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <time.h>
```

```
time_t time(time_t *timeptr);
```

### General Description

Determines the current calendar time.

**Note** This function is sensitive to time zone information which is provided by:

- The TZ environmental variable when POSIX(ON) and TZ is correctly defined, or by the \_TZ environmental variable when POSIX(OFF) and \_TZ is correctly defined.
- The LC\_TOD category of the current locale if POSIX(OFF) or TZ is not defined.

The time zone external variables tzname, timezone, and daylight declarations remain feature test protected in time.h.

### Returned Value

Returns the current calendar time. The returned value is also stored in the location given by *timeptr*. If *timeptr* is NULL, the returned value is not stored. If the calendar time is not available, the value (time\_t)(-1) is returned.

time() returns the current value of the S/390 TOD clock value obtained with the STCK instruction, rounded off to the nearest second, and normalized to the Posix Epoch, January 1, 1970. The S/390 TOD clock value does not account for leap seconds. If you need more accuracy, use the STCK instruction or the MVS TIME macro which does account for leap seconds using whatever value the system operator has entered for number of leap seconds in the CVT field. For more information about the STCK instruction, refer to the *ESA/390 Principles of Operations*.

A returned value of 0 indicates the epoch, which was at the Coordinated Universal Time of 00:00:00 on January 1, 1970.

### Example CBC3BT11

```
/* CBC3BT11
   This example gets the time and assigns it to ltime, then uses the
   ctime() function to convert the number of seconds to the current date
   and time.
   Finally, it prints a message giving the current time.
*/
#include <time.h>
#include <stdio.h>
```



```
int main(void)
{
    time_t ltime;

    time(&ltime);
    printf("The time is %s\n", ctime(&ltime));
}
```

**Output**

The time is Fri Jun 16 11:01:41 1995

**Related Information**

- “time.h” on page 51
- “asctime() — Convert Time to Character String” on page 106
- “clock() — Determine Processor Time” on page 189
- “ctime() — Convert Time to a Character String” on page 250
- “gmtime() — Convert Time to Broken-Down UTC Time” on page 640
- “localdtconv() — Date/Time Formatting Convention Inquiry” on page 754
- “localtime() — Convert Time and Correct for Local Time” on page 758
- “mktime() — Convert Local Time” on page 828
- “strftime() — Convert to Formatted Time” on page 1432
- “tzset() — Set the Time Zone” on page 1637

## times() — Get Process and Child Process Times

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <sys/times.h>
```

```
clock_t times(struct tms *buffer);
```

### General Description

Gets processor times of interest to a process.

```
struct tms *buffer
```

Points to a memory location where times() can store a structure of information describing processor time used by the current process and other related processes.

times() returns information in a tms structure, which has the following elements:

```
clock_t tms_utime
```

Amount of processor time used by instructions in the calling process.

Under OS/390 UNIX services, this does not include processor time spent running in the kernel. It does include any processor time accumulated for the address space before it became an OS/390 UNIX services process.

```
clock_t tms_stime
```

Amount of processor time used by the system.

Under OS/390 UNIX services, this value represents kernel busy time running on behalf of the calling process. It does not include processor time performing other MVS system functions on behalf of the process.

```
clock_t tms_cutime
```

The sum of tms\_utime and tms\_cutime values for all waited-for child processes which have terminated.

```
clock_t tms_cstime
```

The sum of tms\_stime and tms\_cstime values for all terminated child processes of the calling process.

clock\_t is an integral type determined in the time.h header file. It measures times in terms of *clock ticks*. The number of clock ticks in a second (for your installation) can be found in sysconf(\_SC\_CLK\_TCK).

Times for a terminated child can be determined once wait() or waitpid() have reported the child's termination.

## Returned Value

Returns a value giving the elapsed time since the process was last dubbed (for example, system startup). If this time value cannot be determined, times() returns (clock\_t) - 1.

When unsuccessful, times() sets errno to ERANGE, which indicates an overflow having occurred computing time values.

## Example CBC3BT12

```
/* CBC3BT12
   This example provides the amount of processor time used by instructions
   and the system for the parent and child processes.
*/
#define _POSIX_SOURCE
#include <sys/times.h>
#include <time.h>
#include <sys/types.h>
#include <wait.h>
#include <stdio.h>
#include <unistd.h>

main() {
    int status;
    long i, j;
    struct tms t;
    clock_t dub;

    int tics_per_second;

    tics_per_second = sysconf(_SC_CLK_TCK);

    if (fork() == 0) {
        for (i=0, j=0; i<1000000; i++)
            j += i;
        exit(0);
    }

    if (wait(&status) == -1)
        perror("wait() error");
    else if (!WIFEXITED(status))
        puts("Child did not exit successfully");
    else if ((dub = times(&t)) == -1)
        perror("times() error");
    else {
        printf("process was dubbed %f seconds ago.\n\n",
            ((double) dub)/tics_per_second);
        printf("          utime          stime\n");
        printf("parent:      %f      %f\n",
            ((double) t.tms_utime)/tics_per_second,
            ((double) t.tms_stime)/tics_per_second);
        printf("child:       %f      %f\n",
            ((double) t.tms_cutime)/tics_per_second,
            ((double) t.tms_cstime)/tics_per_second);
    }
}
```

## Output

process was dubbed 1.600000 seconds ago.

	utime	stime
parent:	0.000000	0.020000
child:	0.320000	0.000000

### **Related Information**

- “sys/times.h” on page 49
- “time.h” on page 51
- “exec Functions” on page 322
- “fork() — Create a New Process” on page 422
- “wait() — Wait for a Child Process to End” on page 1687
- “waitpid() — Wait for a Specific Child Process to End” on page 1692
- “time() — Determine Current Time” on page 1570

## tinit() — Attach and Initialize Subtasks

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	C only	

### Format

```
#include <mtf.h>
```

```
int tinit(const char *parallel_loadmod_name, int
num_subtasks);
```

### General Description

Initializes the MTF environment under MVS.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

tinit() is invoked from a main task to dynamically attach and initialize the number of subtasks specified by *num\_subtasks*, where *num\_subtasks* ranges from 1 to MAXTASK (which is defined in the header file mtf.h). After the subtasks have been attached and initialized by tinit(), each of the subtasks will be given a *task\_id* and can then compute independent pieces of the program, in parallel with the main task, under the control of the tsched() and tsyncro() library functions.

The parallel load module (*parallel\_loadmod\_name*) must contain all parallel functions and reside in a partitioned data set named in the STEPLIB DD statement of the JCL that runs the program.

The tinit() function may be called by a main task only once prior to invoking tterm(). Invocations of tinit() after the first are one are terminated with a returned value indicating that MTF is already active.

After tterm() has been called to terminate and remove the MTF environment, or after an abend, tinit() can be called again to create a new MTF environment. The new initialization is independent of the old one and may provide a different number of tasks and/or a different parallel load module.

If tinit() is called from a parallel function, tinit() will be terminated with a returned value indicating that MTF calls cannot be issued from a parallel function.

If tinit() is called by a program running under IMS, CICS, or DB2, the request will not be processed and the returned value will indicate that MTF calls are not supported under these systems.

## Returned Value

Table 40. Returned values for *tinit()*

Value	Meaning
MTF_OK	The subtasks have been successfully attached, and the MTF environment has been created.
ESUBCALL	The MTF call was issued from a subtask.
EWRONGOS	MTF is not supported under IMS, CICS or DB2*.
EACTIVE	MTF has already been initialized and is active.
ENAME2LNG	Parallel load module name is longer than 8 characters.
ETASKNUM	Number of tasks specified is invalid (<1 or >MAXTASK).
ENOMEM	There was insufficient storage for MTF-internal areas.
EMODFIND	Parallel load module was not found.
EMODREAD	Parallel load module was not successfully read.
EMODFMT	Parallel load module has an invalid format.
EAUTOALC	Automatic allocation of standard stream DD has failed.
ETASKFAIL	The attempt to attach task(s) failed.
ETASKABND	One or more subtasks have terminated abnormally.
<b>Note:</b> These values are macros and can be found in the mtf.h header file.	

## Related Information

- “mtf.h” on page 37
- “tsched() — Schedule MTF Subtask” on page 1615
- “tsyncro() — Wait for MTF Subtask Termination” on page 1628
- “terminate() — Terminate After Failures in C++ Error Handling” on page 1560.

## t\_listen() — Listen for a Connect Indication

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_listen(int fd, struct t_call *call);
```

### General Description

Listens for a connect request from a calling transport user. The argument *fd* identifies the local transport endpoint where connect indications arrive, and on return, *call* contains information describing the connect indication. The parameter *call* points to a *t\_call* structure which contains the following members:

```
struct netbuf  addr;
struct netbuf  opt;
struct netbuf  udata;
int            sequence;
```

In *call*, *addr* returns the protocol address of the calling transport user. This address is in a format usable in future calls to *t\_connect()*. However, *t\_connect()* may fail for other reasons; For example *TADDRBUSY*. *opt* returns options associated with the connect request. *udata* is meaningless because transmission of user data is not supported across a connect request. *sequence* is a number that uniquely identifies the returned connect indication. The value of *sequence* enables the user to listen for multiple connect indications before responding to any of them.

Since this function returns values for the *addr*, *opt* and *udata* fields of *call*, the *maxlen* field of each must be set before issuing the *t\_listen()* to indicate the maximum size of the buffer for each.

By default, *t\_listen()* executes in synchronous mode and waits for a connect indication to arrive before returning to the user. However, if *O\_NONBLOCK* is set via *t\_open()* or *fcntl()*, *t\_listen()* executes asynchronously, reducing to a poll for existing connect indications. If none are available, it returns -1 and sets *t\_errno* to *TNODATA*.

### Valid States

T\_IDLE, T\_INCON

### Returned Value

Upon successful completion, a value of 0 is returned. The TCP transport provider does not differentiate between a connect indication and the connection itself. A successful return of *t\_listen()* indicates an existing connection.

On failure, a value of -1 is returned and *t\_errno* is set to indicate an error. On failure, *t\_errno* is set to one of the following:

TBADF The specified file descriptor does not refer to a transport endpoint.

TBADQLEN The argument *qlen* of the endpoint referenced by *fd* is zero.

**TBUFOVFLW**

The number of bytes allocated for an incoming argument (*maxlen*) is greater than 0 but not sufficient to store the value of that argument. The provider's state, as seen by the user, changes to T\_INCON, and the connect indication information to be returned in call is discarded. The value of sequence returned can be used to do a `t_snddis()` .

**TNODATA** O\_NONBLOCK was set, but no connect indications had been queued.

**TLOOK** An asynchronous event has occurred on this transport endpoint and requires immediate attention.

**TNOTSUPPORT**

This function is not supported by the underlying transport provider.

**TOUTSTATE**

The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

**TSYSERR** A system error has occurred during execution of this function.

**TQFULL** The maximum number of outstanding indications has been reached for the endpoint referenced by *fd*.

**TPROTO** This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t\_errno*).

**Related Information**

- “xti.h” on page 56
- “fcntl() — Control Open File Descriptors” on page 350
- “t\_accept() — Accept a Connect Request” on page 1493
- “t\_alloc() — Allocate a Library Structure” on page 1498
- “t\_bind() — Bind an Address to a Transport Endpoint” on page 1504
- “t\_connect() — Establish a Connection with Another Transport User” on page 1524
- “t\_open() — Establish a Transport Endpoint” on page 1594
- “t\_optmgmt() — Manage Options for a Transport Endpoint” on page 1596
- “t\_rcvconnect() — Receive the Confirmation from a Connect Request” on page 1607



## t\_look() — Look at the Current Event on a Transport Endpoint

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_look(int fd);
```

### General Description

Returns the current event on the transport endpoint specified by *fd*. This function enables a transport provider to notify a transport user of an asynchronous event when the user is calling functions in synchronous mode. Certain events require immediate notification of the user and are indicated by a specific error, TL00K, on the current or next function to be executed. This function also enables a transport user to poll a transport endpoint periodically for asynchronous events.

Additional functionality for handling events is provided through select and poll.

### Valid States

All - except for T\_UNINIT

The following list describes the asynchronous events which cause an XTI call to return with a TL00K error:

t\_accept() T\_DISCONNECT, T\_LISTEN

t\_connect() T\_DISCONNECT, T\_LISTEN

This occurs only when a t\_connect is done on an endpoint which has been bound with a *qlen* > 0 and for which a connect indication is pending.

t\_listen() T\_DISCONNECT

This event indicates a disconnect on an outstanding connect indication.

t\_rcv() T\_DISCONNECT

This occurs only when all pending data has been read.

t\_rcvconnect()  
T\_DISCONNECT

t\_rcvudata() T\_UDERR

t\_snd() T\_DISCONNECT

t\_sndudata()  
T\_UDERR

t\_unbind() T\_LISTEN, T\_DATA

T\_DATA may only occur for the connectionless mode.

t\_snddis() T\_DISCONNECT

Once a TL00K error has been received on a transport endpoint via an XTI function, subsequent calls to that and other XTI functions, to which the same TL00K error applies, will continue to return TL00K until the event is consumed. An event causing the TL00K error can be determined by calling t\_look() and then can be consumed by calling the corresponding consuming XTI function as defined in Table 41.

Table 41. Events and t_look()		
Event	Cleared on t_look()?	Consuming XTI functions
T_LISTEN	No	t_listen()
T_CONNECT	No	t_{rcv}connect()  In the case of the t_connect() function the T_CONNECT event is both generated and consumed by the execution of the function and is therefore not visible to the application.
T_DATA	No	t_rcv()
T_EXDATA	No	t_rcv()
T_DISCONNECT	No	t_rcvdis()
T_GODATA	Yes	t_snd()
T_GOEXDATA	Yes	t_snd()

### Returned Value

Upon success, t\_look() returns a value that indicates which of the allowable events has occurred, or returns zero if no event exists. One of the following events is returned:

T\_LISTEN Connection indication received.

T\_CONNECT  
Connect confirmation received.

T\_DATA Normal data received.

T\_EXDATA Expedited data received.

T\_DISCONNECT  
Disconnect received.

T\_GODATA Flow control restrictions on normal data flow that led to a TFLOW error have been lifted. Normal data may be sent again.

T\_GOEXDATA  
Flow control restrictions on expedited data flow that led to a TFLOW error have been lifted. Expedited data may be sent again.

On failure, -1 is returned and t\_errno is set to indicate the error. t\_errno is set to one of the following:

TBADF The specified file descriptor does not refer to a transport endpoint.

TSYSERR A system error has occurred during execution of this function.

TPROTO     This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t\_errno*).

### Related Information

- “xti.h” on page 56
- “t\_open() — Establish a Transport Endpoint” on page 1594
- “t\_snd() — Send Data or Expedited Data Over a Connection” on page 1619
- “t\_sndudata — Send a Data Unit” on page 1624

## tmpfile() — Create Temporary File

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
FILE *tmpfile(void);
```

### General Description

Creates a temporary binary file. It opens the temporary file in `wb+` mode. The file is automatically removed when it is closed or when the program is terminated.

### Returned Value

Returns a pointer to the stream associated with the file created, if successful. If it cannot open the file, it returns a NULL pointer. On normal termination (`exit()`), these temporary files are removed. On abnormal termination, an effort is made to remove these files.

### Returned Value for OS/390 UNIX Services

When the calling application is an OS/390 UNIX services program, the temporary file is created in the hierarchical file system. The file is created in the directory referred to by the `TMPDIR` environment variable, or `/tmp` if `TMPDIR` is not defined.

### Special Behavior for XPG4

The following are the possible values of `errno`:

<code>EINTR</code>	A signal was caught during <code>tmpfile()</code> .
<code>EMFILE</code>	<code>{OPEN_MAX}</code> file descriptors are currently open in the calling process. <code>{FOPEN_MAX}</code> streams are currently open in the calling process.
<code>ENFILE</code>	The maximum allowable number of files is currently open in the system.
<code>ENOSPC</code>	The directory or file system which would contain the new file cannot be expanded.
<code>ENOMEM</code>	Insufficient storage space is available.

### Example

#### CBC3BT13

```
/* CBC3BT13
   This example creates a temporary file and if successful, writes
   tmpstring to it.
   At program termination, the file is removed.
*/
#include <stdio.h>
```

```
int main(void) {
    FILE *stream;
    char tmpstring[ ] = "This string will be written";

    {
        if((stream = tmpfile( )) == NULL)
            printf("Cannot make a temporary file\n");
        else
            fprintf(stream, "%s", tmpstring);
    }
}
```

### Related Information

- “stdio.h” on page 43
- “fopen() — Open a File” on page 417

## tmpnam() — Produce Temporary File Name

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
char *tmpnam(char *string);
```

### General Description

Produces a valid file name that is not the same as the name of any existing file. It stores this name in *string*. If *string* is a NULL pointer, tmpnam() leaves the result in an internal static buffer. Any subsequent calls may modify this object. If *string* is not a NULL pointer, it must point to an array of at least L\_tmpnam bytes. The value of L\_tmpnam is defined in the stdio.h header file.

To provide an ASCII input/output format for applications using this function, define feature test macro \_\_LIBASCII as described on page 22.

### Returned Value

If string is a NULL pointer, tmpnam() returns the pointer to the internal static object in which the generated unique name is placed. Otherwise, if *string* is not a NULL pointer, it returns the value of *string*. The tmpnam() function produces a different name each time it is called within a module up to at least TMP\_MAX names. Files created using names returned by tmpnam() are not automatically discarded at the end of the program.

### Returned Value for OS/390 UNIX Services

When the calling application is an OS/390 UNIX services program, the file name returned is a unique file name in the hierarchical file system (HFS). The directory component of the file name will be the value of the TMPDIR environment variable, or '/tmp' if TMPDIR is not defined.

### Example

#### CBC3BT14

```
/* CBC3BT14
   This example calls tmpnam() to produce a valid file name.
*/
#include <stdio.h>

int main(void)
{
    char *name1;
    if ((name1 = tmpnam(NULL)) != NULL)
        printf("%s can be used as a file name.\n", name1);
    else printf("Cannot create a unique file name\n");
}
```

**Related Information**

- “stdio.h” on page 43
- “fopen() — Open a File” on page 417

## toascii() — Translate Integer to a 7-bit ASCII Character

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	POSIX(ON)

### Format

#### **\_XOPEN\_SOURCE**

```
#define _XOPEN_SOURCE
#include <ctype.h>

int toascii(int c);
```

#### **\_ALL\_SOURCE**

```
#define _ALL_SOURCE
#include <ctype.h>

int toascii(int c);
```

### General Description

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

#### **Special Behavior for \_XOPEN\_SOURCE**

The `toascii()` function converts its argument to a 7-bit US-ASCII character code.

The `toascii()` function is not intended to be used to convert EBCDIC characters to ASCII, attempts to use it in this manner will not function as expected.

#### **Special Behavior for \_ALL\_SOURCE**

`toascii()` assumes `c` modulo 256 is a single byte EBCDIC encoding for a Latin 1 character, `<input-character>`, in the current locale. Then, `toascii()` determines to what character, `<output-character>`, `toascii()` would map `<input-character>` in an ASCII locale (e.g., on an \*IX system) and returns the EBCDIC encoding for `<output-character>` in the current locale.

For example, if the program invoking `toascii()` was compiled with `_ALL_SOURCE` defined and if the value `c` input to `toascii()` modulo 256 is the EBCDIC encoding for `<international-currency-symbol>` in the current locale, `toascii()` returns the EBCDIC encoding for `<dollar>` in the current locale because `toascii()` maps `<international-currency-symbol>` to `<dollar>` on ASCII platforms.



## Returned Value

### Special Behavior for `_XOPEN_SOURCE`

The `toascii()` function returns the value `(c & 0x7f)`.

### Special Behavior for `_ALL_SOURCE`

If the current locale is not a single byte locale (i.e., `mb_cur_max > 1`), `toascii()` sets `errno` to **ENOSYS** and returns -1. Otherwise, `toascii()` assumes `c` modulo 256 is the encoding of a Latin 1 character, `<input-character>`, in the current locale and returns the EBCDIC encoding of the same or another Latin 1 character, `<output-character>`, in the current locale; where `<output-character>` corresponds to the character to which `toascii()` would map `<input-character>` on an ASCII platform.

EBCDIC and ASCII encodings and `<input-character>` to `<output-character>` mapping performed by `toascii()` in an IBM-1047 locale are as follows:

```

/*****
/*
/* IBM-1047 toascii table (sorted by ebcdic)
/*
/* For ISO-8859 character encoding toascii(ch) returns ch for
/* values of ch less than 128 and ch-128 for values between
/* 128 and 255, inclusive. Table below shows corresponding
/* toascii(ch) equivalence for IBM-1047 character encoding
/* of the Latin character set.
/*
/*
/* Character          IBM 1047          ISO 8859-1
/* (Symbolic Name)   Encoding          Encoding
/*                   (Hexadecimal)      (Hexadecimal)
/*
/* ch    toascii(ch)  ch toascii(ch)    ch toascii(ch)
/*
/* <NUL>    <NUL>      00    00          00    00
/* <SOH>    <SOH>      01    01          01    01
/* <STX>    <STX>      02    02          02    02
/* <ETX>    <ETX>      03    03          03    03
/* <SEL>    <IFS/IS4>  04    1C          9C    1C
/* <HT>     <HT>       05    05          09    09
/* <RNL>    <ACK>      06    2E          86    06
/* <DEL>    <DEL>      07    07          7F    7F
/* <GE>     <ETB>      08    26          97    17
/* <SPS>    <CR>       09    0D          8D    0D
/* <RPT>    <SO>       0A    0E          8E    0E
/* <VT>     <VT>       0B    0B          0B    0B
/* <FF>     <FF>       0C    0C          0C    0C
/* <CR>     <CR>       0D    0D          0D    0D
/* <SO>     <SO>       0E    0E          0E    0E
/* <SI>     <SI>       0F    0F          0F    0F
/* <DLE>    <DLE>      10    10          10    10
/* <DC1>    <DC1>      11    11          11    11
/* <DC2>    <DC2>      12    12          12    12
/* <DC3>    <DC3>      13    13          13    13
/* <RES/ENP> <IGS/IS3>  14    1D          9D    1D
/* <NL>     <NL>       15    15          0A    0A
/* <BS>     <BS>       16    16          08    08
/* <POC>    <BEL>      17    2F          87    07
/* <CAN>    <CAN>      18    18          18    18
/* <EM>     <EM>       19    19          19    19
/* <UBS>    <DC2>      1A    12          92    12
/* <CU1>    <SI>       1B    0F          8F    0F
/* <IFS/IS4> <IFS/IS4> 1C    1C          1C    1C

```

/*	<IGS/IS3>	<IGS/IS3>	1D	1D	1D	1D	*/
/*	<IRS/IS2>	<IRS/IS2>	1E	1E	1E	1E	*/
/*	<IUS/IS1>	<IUS/IS1>	1F	1F	1F	1F	*/
/*	<DS>	<NUL>	20	00	80	00	*/
/*	<SOS>	<SOH>	21	01	81	01	*/
/*	<FS>	<STX>	22	02	82	02	*/
/*	<WUS>	<ETX>	23	03	83	03	*/
/*	<BYP/INP>	<EOT>	24	37	84	04	*/
/*	<LF>	<ENQ>	25	2D	85	05	*/
/*	<ETB>	<ETB>	26	26	17	17	*/
/*	<ESC>	<ESC>	27	27	1B	1B	*/
/*	<SA>	<BS>	28	16	88	08	*/
/*	<SFE>	<HT>	29	05	89	09	*/
/*	<SM/SW>	<NL>	2A	15	8A	0A	*/
/*	<CSP>	<VT>	2B	0B	8B	0B	*/
/*	<MFA>	<FF>	2C	0C	8C	0C	*/
/*	<ENQ>	<ENQ>	2D	2D	05	05	*/
/*	<ACK>	<ACK>	2E	2E	06	06	*/
/*	<BEL>	<BEL>	2F	2F	07	07	*/
/*	(reserved)	<DLE>	30	10	90	10	*/
/*	(reserved)	<DC1>	31	11	91	11	*/
/*	<SYN>	<SYN>	32	32	16	16	*/
/*	<IR>	<DC3>	33	13	93	13	*/
/*	<PP>	<DC4>	34	3C	94	14	*/
/*	<TRN>	<NAK>	35	3D	95	15	*/
/*	<NBS>	<SYN>	36	32	96	16	*/
/*	<EOT>	<EOT>	37	37	04	04	*/
/*	<SBS>	<CAN>	38	18	98	18	*/
/*	<IT>	<EM>	39	19	99	19	*/
/*	<RFF>	<SUB>	3A	3F	9A	1A	*/
/*	<CU3>	<ESC>	3B	27	9B	1B	*/
/*	<DC4>	<DC4>	3C	3C	14	14	*/
/*	<NAK>	<NAK>	3D	3D	15	15	*/
/*	(reserved)	<IRS/IS2>	3E	1E	9E	1E	*/
/*	<SUB>	<SUB>	3F	3F	1A	1A	*/
/*							*/
/*	<space>	<space>	40	40	20	20	*/
/*	<nobrk-sp>	<space>	41	40	A0	20	*/
/*	<a-circum>	<b>	42	82	E2	62	*/
/*	<a-diaere>	<d>	43	84	E4	64	*/
/*	<a-grave>	<grave>	44	79	E0	60	*/
/*	<a-acute>	<a>	45	81	E1	61	*/
/*	<a-tilde>	<c>	46	83	E3	63	*/
/*	<a-ring>	<e>	47	85	E5	65	*/
/*	<c-cedilla>	<g>	48	87	E7	67	*/
/*	<n-tilde>	<q>	49	98	F1	71	*/
/*	<cent-sign>	<quote>	4A	7F	A2	22	*/
/*	<period>	<period>	4B	4B	2E	2E	*/
/*	<lt>	<lt>	4C	4C	3C	3C	*/
/*	<l-paren>	<l-paren>	4D	4D	28	28	*/
/*	<plus>	<plus>	4E	4E	2B	2B	*/
/*	<ver-line>	<ver-line>	4F	4F	7C	7C	*/
/*	<ampersand>	<ampersand>	50	50	26	26	*/
/*	<e-acute>	<i>	51	89	E9	69	*/
/*	<e-circum>	<j>	52	91	EA	6A	*/
/*	<e-diaere>	<k>	53	92	EB	6B	*/
/*	<e-grave>	<h>	54	88	E8	68	*/
/*	<i-acute>	<m>	55	94	ED	6D	*/
/*	<i-circum>	<n>	56	95	EE	6E	*/
/*	<i-diaere>	<o>	57	96	EF	6F	*/
/*	<i-grave>	<l>	58	93	EC	6C	*/
/*	<s-sharp>	<underscr>	59	6D	DF	5F	*/
/*	<exclama>	<exclama>	5A	5A	21	21	*/
/*	<dollar>	<dollar>	5B	5B	24	24	*/

/* <asterisk> <asterisk>	5C	5C	2A	2A	*/
/* <r-paren> <r-paren>	5D	5D	29	29	*/
/* <semicolon><semicolon>	5E	5E	3B	3B	*/
/* <circum> <circum>	5F	5F	5E	5E	*/
/* <hyphen> <hyphen>	60	60	2D	2D	*/
/* <slash> <slash>	61	61	2F	2F	*/
/* <A-circum> <B>	62	C2	C2	42	*/
/* <A-diaere> <D>	63	C4	C4	44	*/
/* <A-grave> <at>	64	7C	C0	40	*/
/* <A-acute> <A>	65	C1	C1	41	*/
/* <A-tilde> <C>	66	C3	C3	43	*/
/* <A-ring> <E>	67	C5	C5	45	*/
/* <C-cedilla><G>	68	C7	C7	47	*/
/* <N-tilde> <Q>	69	D8	D1	51	*/
/* <brok-bar> <ampersand>	6A	50	A6	26	*/
/* <comma> <comma>	6B	6B	2C	2C	*/
/* <percent> <percent>	6C	6C	25	25	*/
/* <underscr> <underscr>	6D	6D	5F	5F	*/
/* <gt> <gt>	6E	6E	3E	3E	*/
/* <question> <question>	6F	6F	3F	3F	*/
/* <o-stroke> <x>	70	A7	F8	78	*/
/* <E-acute> <I>	71	C9	C9	49	*/
/* <E-circum> <J>	72	D1	CA	4A	*/
/* <E-diaere> <K>	73	D2	CB	4B	*/
/* <E-grave> <H>	74	C8	C8	48	*/
/* <I-acute> <M>	75	D4	CD	4D	*/
/* <I-circum> <N>	76	D5	CE	4E	*/
/* <I-diaere> <O>	77	D6	CF	4F	*/
/* <I-grave> <L>	78	D3	CC	4C	*/
/* <grave> <grave>	79	79	60	60	*/
/* <colon> <colon>	7A	7A	3A	3A	*/
/* <num-sign> <num-sign>	7B	7B	23	23	*/
/* <at> <at>	7C	7C	40	40	*/
/* <apostro> <apostro>	7D	7D	27	27	*/
/* <eq> <eq>	7E	7E	3D	3D	*/
/* <quote> <quote>	7F	7F	22	22	*/
/* <O-stroke> <X>	80	E7	D8	58	*/
/* <a> <a>	81	81	61	61	*/
/* <b> <b>	82	82	62	62	*/
/* <c> <c>	83	83	63	63	*/
/* <d> <d>	84	84	64	64	*/
/* <e> <e>	85	85	65	65	*/
/* <f> <f>	86	86	66	66	*/
/* <g> <g>	87	87	67	67	*/
/* <h> <h>	88	88	68	68	*/
/* <i> <i>	89	89	69	69	*/
/* <l-guille> <plus>	8A	4E	AB	2B	*/
/* <r-guille> <semicolon>	8B	5E	BB	3B	*/
/* <eth> <p>	8C	97	F0	70	*/
/* <y-acute> <r-brace>	8D	D0	FD	7D	*/
/* <thorn> <tilde>	8E	A1	FE	7E	*/
/* <plusminus><one>	8F	F1	B1	31	*/
/* <degree> <zero>	90	F0	B0	30	*/
/* <j> <j>	91	91	6A	6A	*/
/* <k> <k>	92	92	6B	6B	*/
/* <l> <l>	93	93	6C	6C	*/
/* <m> <m>	94	94	6D	6D	*/
/* <n> <n>	95	95	6E	6E	*/
/* <o> <o>	96	96	6F	6F	*/
/* <p> <p>	97	97	70	70	*/
/* <q> <q>	98	98	71	71	*/
/* <r> <r>	99	99	72	72	*/
/* <fem-ind> <asterisk>	9A	5C	AA	2A	*/
/* <mas-ind> <colon>	9B	7A	BA	3A	*/

/* <ae>	<f>	9C	86	E6	66	*/
/* <cedilla>	<eight>	9D	F8	B8	38	*/
/* <AE>	<F>	9E	C6	C6	46	*/
/* <cur-sign>	<dollar>	9F	5B	A4	24	*/
/* <mu>	<five>	A0	F5	B5	35	*/
/* <tilde>	<tilde>	A1	A1	7E	7E	*/
/* <s>	<s>	A2	A2	73	73	*/
/* <t>	<t>	A3	A3	74	74	*/
/* <u>	<u>	A4	A4	75	75	*/
/* <v>	<v>	A5	A5	76	76	*/
/* <w>	<w>	A6	A6	77	77	*/
/* <x>	<x>	A7	A7	78	78	*/
/* <y>	<y>	A8	A8	79	79	*/
/* <z>	<z>	A9	A9	7A	7A	*/
/* <inv-excl>	<exclama>	AA	5A	A1	21	*/
/* <inv-ques>	<question>	AB	6F	BF	3F	*/
/* <Eth>	<P>	AC	D7	D0	50	*/
/* <l-brk>	<l-brk>	AD	AD	5B	5B	*/
/* <Thorn>	<circum>	AE	5F	DE	5E	*/
/* <register>	<period>	AF	4B	AE	2E	*/
/* <not-sign>	<comma>	B0	6B	AC	2C	*/
/* <pound>	<num-sign>	B1	7B	A3	23	*/
/* <yen>	<percent>	B2	6C	A5	25	*/
/* <mid-dot>	<seven>	B3	F7	B7	37	*/
/* <copyright>	<r-paren>	B4	5D	A9	29	*/
/* <section>	<apostro>	B5	7D	A7	27	*/
/* <paragraph>	<six>	B6	F6	B6	36	*/
/* <1/4>	<lt>	B7	4C	BC	3C	*/
/* <1/2>	<eq>	B8	7E	BD	3D	*/
/* <3/4>	<gt>	B9	6E	BE	3E	*/
/* <Y acute>	<r-brk>	BA	BD	DD	5D	*/
/* <diaeresis>	<l-paren>	BB	4D	A8	28	*/
/* <macron>	<slash>	BC	61	AF	2F	*/
/* <r-brk>	<r-brk>	BD	BD	5D	5D	*/
/* <acute>	<four>	BE	F4	B4	34	*/
/* <multiply>	<W>	BF	E6	D7	57	*/
/* <l-brace>	<l-brace>	C0	C0	7B	7B	*/
/* <A>	<A>	C1	C1	41	41	*/
/* <B>	<B>	C2	C2	42	42	*/
/* <C>	<C>	C3	C3	43	43	*/
/* <D>	<D>	C4	C4	44	44	*/
/* <E>	<E>	C5	C5	45	45	*/
/* <F>	<F>	C6	C6	46	46	*/
/* <G>	<G>	C7	C7	47	47	*/
/* <H>	<H>	C8	C8	48	48	*/
/* <I>	<I>	C9	C9	49	49	*/
/* <soft-hyp>	<hyphen>	CA	60	AD	2D	*/
/* <o-circum>	<t>	CB	A3	F4	74	*/
/* <o-diaere>	<v>	CC	A5	F6	76	*/
/* <o-grave>	<r>	CD	99	F2	72	*/
/* <o-acute>	<s>	CE	A2	F3	73	*/
/* <o-tilde>	<u>	CF	A4	F5	75	*/
/* <r-brace>	<r-brace>	D0	D0	7D	7D	*/
/* <J>	<J>	D1	D1	4A	4A	*/
/* <K>	<K>	D2	D2	4B	4B	*/
/* <L>	<L>	D3	D3	4C	4C	*/
/* <M>	<M>	D4	D4	4D	4D	*/
/* <N>	<N>	D5	D5	4E	4E	*/
/* <O>	<O>	D6	D6	4F	4F	*/
/* <P>	<P>	D7	D7	50	50	*/
/* <Q>	<Q>	D8	D8	51	51	*/
/* <R>	<R>	D9	D9	52	52	*/
/* <super-1>	<nine>	DA	F9	B9	39	*/
/* <u-circum>	<l-brace>	DB	C0	FB	7B	*/

```

/* <u-diaere> <ver_line> DC      4F      FC      7C      */
/* <u-grave> <y> DD      A8      F9      79      */
/* <u-acute> <z> DE      A9      FA      7A      */
/* <y-diaere> <DEL> DF      07      FF      7F      */
/* <backslash><backslash> E0      E0      5C      5C      */
/* <division> <w> E1      A6      F7      77      */
/* <S> <S> E2      E2      53      53      */
/* <T> <T> E3      E3      54      54      */
/* <U> <U> E4      E4      55      55      */
/* <V> <V> E5      E5      56      56      */
/* <W> <W> E6      E6      57      57      */
/* <X> <X> E7      E7      58      58      */
/* <Y> <Y> E8      E8      59      59      */
/* <Z> <Z> E9      E9      5A      5A      */
/* <super-2> <two> EA      F2      B2      32      */
/* <O-circum> <T> EB      E3      D4      54      */
/* <O-diaere> <V> EC      E5      D6      56      */
/* <O-grave> <R> ED      D9      D2      52      */
/* <O-acute> <S> EE      E2      D3      53      */
/* <O-tilde> <U> EF      E4      D5      55      */
/* <zero> <zero> F0      F0      30      30      */
/* <one> <one> F1      F1      31      31      */
/* <two> <two> F2      F2      32      32      */
/* <three> <three> F3      F3      33      33      */
/* <four> <four> F4      F4      34      34      */
/* <five> <five> F5      F5      35      35      */
/* <six> <six> F6      F6      36      36      */
/* <seven> <seven> F7      F7      37      37      */
/* <eight> <eight> F8      F8      38      38      */
/* <nine> <nine> F9      F9      39      39      */
/* <super-3> <three> FA      F3      B3      33      */
/* <U-circum> <l-brk> FB      AD      DB      5B      */
/* <U-diaere> <backslash> FC      E0      DC      5C      */
/* <U-grave> <Y> FD      E8      D9      59      */
/* <U-acute> <Z> FE      E9      DA      5A      */
/* <E0> <IUS/IS1> FF      1F      9F      1F      */
/*
/*****

```

## Related Information

- “isascii() — Test for 7-bit US-ASCII Character” on page 697

## tolower() - toupper() — Convert Character Case

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <ctype.h>
```

```
int tolower(int c); /* Convert c to lowercase if appropriate */
int toupper(int c); /* Convert c to uppercase if appropriate */
```

### General Description

Converts *c* to a lowercase letter, if possible. Conversely, the `toupper()` function converts *c* to an uppercase letter, if possible.

The DBCS is not supported. The use of characters from the DBCS results in unspecified behavior.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

`tolower()` and `toupper()` return the corresponding character, as defined in the `LC_CTYPE` category of the current locale, if such a character exists; otherwise, they return the unchanged value *c*.

### Example

#### CBC3BT15

```
/* CBC3BT15
   This example demonstrates the result of using toupper() and tolower()
   on a lower-case a.
*/
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    int ch;

    ch = 0x81;
    printf("toupper=%#04x\n", toupper(ch));
    printf("tolower=%#04x\n", tolower(ch));
}
```

### Related Information

- “`ctype.h`” on page 24
- “`isalnum()` to `isxdigit()` — Test Integer Value” on page 694
- “`towlower()` - `towupper()` — Convert Wide Character Case” on page 1604

## `_tolower()` — Translate Uppercase Characters to Lower Case

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <ctype.h>
```

```
int _tolower(int c);
```

### General Description

The `_tolower()` macro is equivalent to `tolower(c)` except that the argument `c` must be an upper-case letter.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

The `_tolower()` macro returns the lower-case letter corresponding to the argument passed.

### Related Information

- “`tolower()` - `toupper()` — Convert Character Case” on page 1592

## t\_open() — Establish a Transport Endpoint

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_open(char *name, int oflag, struct
           t_info *info);
```

### General Description

t\_open() must be called as the first step in the initialization of a transport endpoint. This function establishes a transport endpoint by supplying a transport provider identifier that indicates a particular transport provider (that is, transport protocol) and returning a file descriptor that identifies that endpoint.

The argument *name* points to a transport provider identifier. The only supported transport provider is "/dev/tcp", indicating a TCP transport provider. No device by that name actually exists in the filesystem. It is purely used to follow historical convention. The argument *oflag* identifies any open flags (as in open() ). It is constructed from O\_RDWR optionally bitwise inclusive-OR'ed with O\_NONBLOCK. These flags are defined by the header <fcntl.h>. The file descriptor returned by t\_open() will be used by all subsequent functions to identify the particular local transport endpoint.

This function also returns various default characteristics of the underlying transport protocol by setting fields in the *info* structure. This argument points to a t\_info structure which contains the following members:

```
long addr;      /* max size of the transport protocol address */
long options;   /* max number of bytes of options */
               /* protocol-specific options */
long tsdu;      /* max size of a transport service data */
               /* unit (TSDU) */
long etsdu;     /* max size of an expedited transport */
               /* service data unit (ETSDU) */
long connect;   /* max amount of data allowed on */
               /* connection establishment functions */
long discon;    /* max amount of data allowed on */
               /* t_snddis() and t_rcvdis() functions */
long servtype;  /* service type supported by the */
               /* transport provider */
long flags;     /* other info about the transport provider */
```

The fields take on the following values:

addr	The size of a struct sockaddr_in is returned.
options	The value 304, which is the maximum number of bytes of options which can possibly be specified or requested, is returned.
tsdu	Zero is returned, indicating that the TCP transport provider does not support the concept of TSUs.



etsdu	A value of -1 is returned, indicating that there is no limit on the size of an ETSDU.
connect	A value of -2 is returned, indicating that the TCP transport provider does not allow data to be sent with connection establishment functions.
discon	A value of -2 is returned, indicating that the transport provider does not allow data to be sent with the abortive release functions.
servtype	T_COTS is always returned, since this is the only service type supported.
flags	The T_SENDZERO bit is always set in this field, indicating that the TCP transport provider supports the sending of zero-length TSDUs.

If info is set to a null pointer by the transport user, no protocol information is returned by t\_open() .

### Valid States

T\_UNINIT

### Returned Value

A valid file descriptor is returned upon successful completion. Otherwise, a value of -1 is returned and *t\_errno* is set to indicate an error. On failure, *t\_errno* is set to the following:

TBADFLAG An invalid flag is specified.

TBADNAME

Invalid transport provider name.

TSYSERR A system error has occurred during execution of this function.

TPROTO This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t\_errno*).

### Related Information

- “xti.h” on page 56
- “open() — Open a File” on page 872

## t\_optmgmt() — Manage Options for a Transport Endpoint

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_optmgmt(int fd, struct t_optmgmt *req,
              struct t_optmgmt *ret);
```

### General Description

Enables a transport user to retrieve, verify or negotiate protocol options with the transport provider. The argument *fd* identifies a transport endpoint. The *req* and *ret* arguments point to a *t\_optmgmt* structure containing the following members:

```
struct netbuf  opt;
long          flags;
```

The *opt* field identifies protocol options and the *flags* field is used to specify the action to take with those options.

The options are represented by a *netbuf* structure in a manner similar to the address in *t\_bind()*. The *netbuf* structure contains the following members:

```
unsigned int  maxlen  maximum buffer value length
unsigned int  len     actual buffer value length
char *       buf     pointer to buffer
```

The argument *req* is used to request a specific action of the provider and to send options to the provider. The argument *len* specifies the number of bytes in the options, *buf* points to the options buffer, and *maxlen* has no meaning for the *req* argument. The transport provider may return options and flag values to the user through *ret*. For *ret*, *maxlen* specifies the maximum size of the options buffer, and *buf* points to the buffer where the options are to be placed. On return, *len* specifies the number of bytes of options returned. The value in *maxlen* has no meaning for the *req* argument, but must be set in the *ret* argument to specify the maximum number of bytes the options buffer can hold.

Each option in the options buffer is of the form struct *t\_opthdr* possibly followed by an option value. The *t\_opthdr* structure contains the following members:

```
unsigned long  len      sizeof(t_opthdr)+optval len
unsigned long  level    protocol affected
unsigned long  name     option value
unsigned long  status   status value
```

The *level* field of struct *t\_opthdr* identifies the XTI level or a protocol of the transport provider. The *name* field identifies the option within the level, and *len* contains its total length. The total length is the length of the option header *t\_opthdr* plus the length of the option value. If *t\_optmgmt()* is called with the action *T\_NEGOTIATE* set, the *status* field of the returned options contains information about the success or failure of a negotiation.

Each option in the input or output option buffer must start at a long-word boundary. The macro `OPT_NEXTHDR(pbuf, buflen, poption)` can be used for that purpose. The parameter *pbuf* denotes a pointer to an option buffer *opt.buf*, and *buflen* is its length. The parameter *poption* points to the current option in the option buffer. `OPT_NEXTHDR` returns a pointer to the position of the next option, or returns a null pointer if the option buffer is exhausted. The macro is helpful for writing and reading. See “xti.h” on page 56 for the exact definition.

If the transport user specifies several options on input, all options must address the same level.

If any option in the options buffer does not indicate the same level as the first option, or the level specified is unsupported, then the `t_optmgmt()` request will fail with `TBADOPT`. If the error is detected, some options have possibly been successfully negotiated. The transport user can check the current status by calling `t_optmgmt()` with the `T_CURRENT` flag set.

Before using this function, you should read Chapter 6, “Use of Options in XTI”, in *X/Open CAE Specification, Networking Services, Issue 4*

The *flags* field of *req* must specify one of the following actions:

#### T\_NEGOTIATE

This action enables the transport user to negotiate option values.

The user specifies the options of interest and their values in the buffer specified by *req->opt.buf* and *req->opt.len*. The negotiated option values are returned in the buffer pointed to by *ret->opt.buf*. The *status* field of each returned option is set to indicate the result of the negotiation. The value is `T_SUCCESS` if the proposed value was negotiated, `T_PARTSUCCESS` if a degraded value was negotiated, `T_FAILURE` if the negotiation failed (according to the negotiation rules), `T_NOTSUPPORT` if the transport provider does not support this option or illegally requests negotiation of a privileged option, and `T_READONLY` if modification of a read-only option was requested. If the status is `T_SUCCESS`, `T_FAILURE`, `T_NOTSUPPORT` or `T_READONLY`, the returned option value is the same as the one requested on input.

The overall result of the negotiation is returned in *ret->flags*.

This field contains the worst single result, whereby the rating is done according to the order `T_NOTSUPPORT`, `T_READONLY`, `T_FAILURE`, `T_PARTSUCCESS`, `T_SUCCESS`. The value `T_NOTSUPPORT` is the worst result and `T_SUCCESS` is the best.

For each level, the option `T_ALLOPT` (see below) can be requested on input. No value is given with this option; only the *t\_opthdr* part is specified. This input requests to negotiate all supported options of this level to their default values. The result is returned option by option in *ret->opt.buf*. (Note that depending on the state of the transport endpoint, not all requests to negotiate the default value may be successful.)

**T\_CHECK** This action enables the user to verify whether the options specified in *req* are supported by the transport provider.

If an option is specified with no option value (it consists only of a

`t_opthdr` structure), the option is returned with its status field set to `T_SUCCESS` if it is supported, `T_NOTSUPPORT` if it is not or needs additional user privileges, and `T_READONLY` if it is read-only (in the current XTI state). No option value is returned.

If an option is specified with an option value, the *status* field of the returned option has the same value, as if the user had tried to negotiate this value with `T_NEGOTIATE`. If the status is `T_SUCCESS`, `T_FAILURE`, `T_NOTSUPPORT` or `T_READONLY`, the returned option value is the same as the one requested on input.

The overall result of the option checks is returned in *ret->flags*. This field contains the worst single result of the option checks, whereby the rating is the same as for `T_NEGOTIATE`.

Note that no negotiation takes place. All currently effective option values remain unchanged.

#### T\_DEFAULT

This action enables the transport user to retrieve the default option values. The user specifies the options of interest in *req->opt.buf*. The option values are irrelevant and will be ignored. It is sufficient to specify the *t\_opthdr* part of an option only. The default values are then returned in *ret->opt.buf*.

The status field returned is `T_NOTSUPPORT` if the protocol level does not support this option or the transport user illegally requested a privileged option, `T_READONLY` if the option is read-only, and set to `T_SUCCESS` in all other cases. The overall result of the request is returned in *ret->flags*. This field contains the worst single result, whereby the rating is the same as for `T_NEGOTIATE`.

For each level, the option `T_ALLOPT` (see below) can be requested on input. All supported options of this level with their default values are then returned. In this case, *ret->opt.maxlen* must be given at least the value *info->options* (see `t_getinfo()`, `t_open()`) before the call.

#### T\_CURRENT

This action enables the transport user to retrieve the currently effective option values. The user specifies the options of interest in *req->opt.buf*. The option values are irrelevant and will be ignored. It is sufficient to specify the *t\_opthdr* part of an option only. The currently effective values are then returned in *ret->opt.buf*.

The status field returned is `T_NOTSUPPORT` if the protocol level does not support this option, or the transport user illegally requested a privileged option, `T_READONLY` if the option is read-only, and set to `T_SUCCESS` in all other cases. The overall result of the request is returned in *ret->flags*. This field contains the worst single result, whereby the rating is the same as for `T_NEGOTIATE`.

For each level, the option `T_ALLOPT` (see below) can be requested on input. All supported options of this level with their currently effective values are then returned.

The option `T_ALLOPT` can only be used with `t_optmgmt()` and the actions `T_NEGOTIATE`, `T_DEFAULT` and `T_CURRENT`. It can be used with any supported level and addresses all supported options of this level. The option has no value; it consists of a *t\_opthdr* only. Since in a `t_optmgmt()` call only options of one level

may be addressed, this option should not be requested together with other options. The function returns as soon as this option has been processed.

Options are independently processed in the order they appear in the input option buffer. If an option is multiply input, it depends on the implementation whether it is multiply output or whether it is returned only once.

The function `t_optmgmt()` may block under various circumstances and depending on the implementation. The function will block, for instance, if the protocol addressed by the call resides on a separate controller. It may also block due to flow control constraints. For example, if data sent previously across this transport endpoint has not yet been fully processed. If the function is interrupted by a signal, the option negotiations that have been done so far may remain valid. The behavior of the function is not changed if `O_NONBLOCK` is set.

### Valid States

All - except for `T_UNINIT`

### XTI-level options

XTI-level options are not specific for a particular transport provider. An XTI implementation supports none, all or any subset of the options defined below. An implementation may restrict the use of any of these options by offering them only in the privileged or read-only mode, or if *fd* relates to specific transport providers.

The subsequent options are not association-related. They may be negotiated in all XTI states except `T_UNINIT`. See Chapter 6, "Use of Options in XTI", in *X/Open CAE Specification, Networking Services, Issue 4* for more information.

The protocol level is `XTI_GENERIC`. For this level, the following options are defined:

#### XTI\_DEBUG

This option enables debugging. The valid values of this option are:

- None (option header only) - indicating that debug is to be turned off.
- -1 - indicating that debug output is to go to `stderr`.
- A file descriptor - indicating the destination file for debug output.

The debug output contains varying information depending on the XTI services invoked. It is meant to be used by customer support personnel.

#### XTI\_LINGER

This option is used to linger the execution of a `t_close()` or `close()` if send data is still queued in the send buffer. The option value specifies the linger period. If a `close()` or `t_close()` is issued and the send buffer is not empty, the system attempts to send the pending data within the linger period before closing the endpoint. Data still pending after the linger period has elapsed is discarded.

Depending on the implementation, `t_close()` or `close()` either block for at maximum the linger period, or immediately return, whereupon the system holds the connection in existence for at most the linger period.

The *option* value consists of a structure `t_linger` declared as:

```
struct t_linger {
    long l_onoff; /* switch option on/off */
    long l_linger; /* linger period in seconds */
}
```

Legal values for the field *l\_onoff* are:

T\_NO            switch option off  
T\_YES          activate option

The value *l\_onoff* is an absolute requirement.

The field *l\_linger* determines the linger period in seconds. The transport user can request the default value by setting the field to T\_UNSPEC. The default timeout value depends on the underlying transport provider (it is often T\_INFINITE). Legal values for this field are T\_UNSPEC, T\_INFINITE and all non-negative numbers.

The *l\_linger* value is not an absolute requirement. The implementation may place upper and lower limits to this value. Requests that fall short of the lower limit are negotiated to the lower limit.

Note that this option does not linger the execution of `t_snddis()`.

#### XTI\_RCVBUF

This option is used to adjust the internal buffer size allocated for the receive buffer. The buffer size may be increased for high-volume connections, or decreased to limit the possible backlog of incoming data.

This request is not an absolute requirement. The implementation may place upper and lower limits on the option value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

#### XTI\_SNDBUF

This option is used to adjust the internal buffer size allocated for the send buffer.

This request is not an absolute requirement. The implementation may place upper and lower limits on the option value. Requests that fall short of the lower limit are negotiated to the lower limit.

Legal values are all positive numbers.

### TCP-level options

The protocol level is INET\_TCP. The following TCP-level options are supported. They are not association-related.

#### TCP\_KEEPAIVE

If this option is set, a keep-alive timer is activated to monitor idle connections that might no longer exist. If a connection has been idle since the last keep-alive timeout, a keep-alive packet is sent to check if the connection is still alive or broken.

The option value consists of a structure *t\_kpalive* declared as:

```

struct t_kpalive {
    long    kp_onoff;        /* switch option on/off */
    long    kp_timeout;     /* keep-alive timeout in minutes */
}

```

Legal values for the field *kp\_onoff* are:

T\_NO            switch keep-alive timer off

T\_YES          activate keep-alive timer

The field *kp\_timeout* determines the frequency of keep-alive packets being sent, in minutes. The transport user can request the default value by setting the field to T\_UNSPEC. The default is 120 minutes. Legal values for this field are T\_UNSPEC and all positive numbers.

The timeout value is not an absolute requirement. However, no limits are currently specified by the TCP transport provider.

### IP-level options

The protocol level is INET\_IP. The following IP-level options are supported. The are not association-related.

#### IP\_REUSEADDR

Generally, users are not allowed to bind more than one transport endpoint to addresses with identical port numbers. If IP\_REUSEADDR is set to T\_YES this restriction is relaxed in the sense that it is now permissible to bind a transport endpoint to an address with a port number and an under-specified internet address and further endpoints to addresses with the same port number and (mutually exclusive) fully specified internet addresses.

### Returned Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *t\_errno* is set to indicate an error. On failure, *t\_errno* is set to one of the following:

TBADF          The specified file descriptor does not refer to a transport endpoint.

#### TOUTSTATE

The function was issued in the wrong sequence.

TACCES        The user does not have permission to negotiate the specified options.

TBADOPT       The specified options were in an incorrect format or contained illegal information.

TBADFLAG     An invalid flag was specified.

#### TBUFOVFLW

The number of bytes allowed for an incoming argument (*maxlen*) is greater than 0 but not sufficient to store the value of that argument. The information to be returned in *ret* will be discarded.

TSYSERR       A system error has occurred during execution of this function.

TPROTO        This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t\_errno*).

**TNOTSUPPORT**

This action is not supported by the transport provider.

**Related Information**

- “xti.h” on page 56
- “t\_accept() — Accept a Connect Request” on page 1493
- “t\_alloc() — Allocate a Library Structure” on page 1498
- “t\_connect() — Establish a Connection with Another Transport User” on page 1524
- “t\_getinfo() — Get Protocol-specific Service Information” on page 1566
- “t\_listen() — Listen for a Connect Indication” on page 1577
- “t\_open() — Establish a Transport Endpoint” on page 1594
- “t\_rcvconnect() — Receive the Confirmation from a Connect Request” on page 1607
- *X/Open CAE Specification, Networking Services, Issue 4* Chapter 6, “Use of Options in XTI”.



## `_toupper()` — Translate Lowercase Characters to Upper-case

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE  
#include <ctype.h>
```

```
int _toupper(int c);
```

### General Description

The `_toupper()` macro is equivalent to `toupper(c)` except that the argument `c` must be a lower-case letter.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

### Returned Value

The `_toupper()` macro returns the upper-case letter corresponding to the argument passed.

### Related Information

- “`isalnum()` to `isxdigit()` — Test Integer Value” on page 694
- “`tolower()` - `toupper()` — Convert Character Case” on page 1592

## towlower() - towupper() — Convert Wide Character Case

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <wctype.h>
```

```
wint_t tolower(wint_t wc);  
wint_t towupper(wint_t wc);
```

### General Description

Converts *wc* to the corresponding lowercase letter. The `towupper()` function converts *wc* to the corresponding uppercase letter.

### Returned Value

If *wc* is a wide character for which `iswupper()` (or `iswlower()`) is true and there is a corresponding wide character for which `iswlower()` (or `iswupper()`) is true, the `towlower()` (or `towupper()`) function returns the corresponding wide character; otherwise, *wc* is returned unchanged.

### Related Information

- “wctype.h” on page 56
- “iswalnum() to iswxdigit() — Test Wide Integer Value” on page 715
- “tolower() - toupper() — Convert Character Case” on page 1592

## t\_rcv() — Receive Data or Expedited Data Sent Over a Connection

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_rcv(int fd, char *buf, unsigned int nbytes,
          int *flags);
```

### General Description

Receives either normal or expedited data. The argument *fd* identifies the local transport endpoint through which data will arrive. *buf* points to a receive buffer where user data is placed. *nbytes* specifies the size of the receive buffer. The argument *flags* may be set on return from `t_rcv()` and specifies optional flags as described below.

By default, `t_rcv()` operates in synchronous mode and will wait for data to arrive if none is currently available. However, if `O_NONBLOCK` is set (via `t_open()` or `fcntl()`), `t_rcv()` will execute in asynchronous mode and will fail if no data is available. (See `TNODATA` below.)

On return from the call, if `T_MORE` is set in *flags*, this indicates that there is more data, and the current expedited transport service data unit (ETSDU) must be received in multiple `t_rcv()` calls. In the asynchronous mode, the `T_MORE` flag may be set on return from the `t_rcv()` call even when the number of bytes received is less than the size of the receive buffer specified. Each `t_rcv()` with the `T_MORE` flag set indicates that another `t_rcv()` must follow to get more data for the current ETSDU. The end of the ETSDU is identified by the return of a `t_rcv()` call with the `T_MORE` flag not set. The `T_MORE` flag is not meaningful for normal data when using the TCP transport provider and should be ignored. If *nbytes* is greater than zero on the call to `t_rcv()`, `t_rcv()` will return 0 only if the end of a TSDU is being returned to the user.

On return, the data returned is expedited data if `T_EXPEDITED` is set in *flags*. If the number of bytes of expedited data exceeds *nbytes*, `t_rcv()` will set `T_EXPEDITED` and `T_MORE` on return from the initial call. Subsequent calls to retrieve the remaining ETSDU will have `T_EXPEDITED` set on return. The end of the ETSDU is identified by the return of a `t_rcv()` call with the `T_MORE` flag not set.

In synchronous mode, the only way for the user to be notified of the arrival of normal or expedited data is to issue this function or check for the `T_DATA` or `T_EXDATA` events using the `t_look()` function. Additionally, the process can arrange to be notified via the select/poll interface.

### Valid States

`T_DATAXFER`

## Returned Value

On successful completion, `t_rcv()` returns the number of bytes received. Otherwise, it returns -1 on failure and `t_errno` is set to indicate the error. On failure, `t_errno` is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TNODATA	O_NONBLOCK was set, but no data is currently available from the transport provider.
TLOOK	An asynchronous event has occurred on this transport endpoint and requires immediate attention.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TOUTSTATE	The function was issued in the wrong sequence on the transport endpoint referenced by <i>fd</i> .
TSYSERR	A system error has occurred during execution of this function.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI ( <code>t_errno</code> ).

## Related Information

- “xti.h” on page 56
- “fcntl() — Control Open File Descriptors” on page 350
- “t\_getinfo() — Get Protocol-specific Service Information” on page 1566
- “t\_look() — Look at the Current Event on a Transport Endpoint” on page 1579
- “t\_open() — Establish a Transport Endpoint” on page 1594
- “t\_snd() — Send Data or Expedited Data Over a Connection” on page 1619

## t\_rcvconnect() — Receive the Confirmation from a Connect Request

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_rcvconnect(int fd, struct t_call *call);
```

### General Description

Enables a calling transport user to determine the status of a previously sent connect request. `t_rcvconnect()` is used in conjunction with `t_connect()` to establish a connection in asynchronous mode. The connection will be established on successful completion of this function.

The argument *fd* identifies the local transport endpoint where communication will be established, and *call* contains information associated with the newly established connection. The argument *call* points to a `t_call` structure which contains the following members:

```
struct netbuf  addr;
struct netbuf  opt;
struct netbuf  udata;
int            sequence;
```

In *call*, *addr* returns the protocol address associated with the responding transport endpoint. *opt* presents any options associated with the connection. *udata* is meaningless since the TCP transport provider does not support transmission of user data during connection establishment. *sequence* has no meaning for this function.

The *maxlen* field of each argument must be set before issuing this function to indicate the maximum size of the buffer for each. However, *call* may be a null pointer, in which case no information is given to the user on return from `t_rcvconnect()`. By default, `t_rcvconnect()` executes in synchronous mode and waits for the connection to be established before returning. On return, the *addr* field contains the address of the remote endpoint, and *opt* reflects the result of negotiation of the options the user specified on input.

If `O_NONBLOCK` is set (via `t_open()` or `fcntl()`), `t_rcvconnect()` executes in asynchronous mode, and reduces to a poll for existing connect confirmations. If none are available, `t_rcvconnect()` fails and returns immediately without waiting for the connection to be established. (See `TNODATA` below.) In this case, `t_rcvconnect()` must be called again to complete the connection establishment phase and retrieve the information returned in *call*.

### Valid States

`T_OUTCON`

## Returned Value

Upon successful completion, a value of zero is returned. Otherwise, a value of -1 is returned, and *t\_errno* is set to indicate an error. On failure, *t\_errno* is set to one of the following:

TBADF	The specified file descriptor does not refer to a transport endpoint.
TBUFOVFLW	The number of bytes allocated for an incoming argument ( <i>maxlen</i> ) is greater than 0 but not sufficient to store the value of that argument, and the connect information to be returned in <i>call</i> will be discarded. The provider's state, as seen by the user, will be changed to T_DATAXFER.
TNODATA	O_NONBLOCK was set, but a connect confirmation has not yet arrived.
TLOOK	An asynchronous event has occurred on this transport connection and requires immediate attention.
TNOTSUPPORT	This function is not supported by the underlying transport provider.
TOUTSTATE	The function was issued in the wrong sequence on the transport endpoint referenced by <i>fd</i> .
TSYSERR	A system error has occurred during execution of this function.
TPROTO	This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI ( <i>t_errno</i> ).

## Related Information

- “xti.h” on page 56
- “t\_accept() — Accept a Connect Request” on page 1493
- “t\_alloc() — Allocate a Library Structure” on page 1498
- “t\_bind() — Bind an Address to a Transport Endpoint” on page 1504
- “t\_connect() — Establish a Connection with Another Transport User” on page 1524
- “t\_listen() — Listen for a Connect Indication” on page 1577
- “t\_open() — Establish a Transport Endpoint” on page 1594
- “t\_optmgmt() — Manage Options for a Transport Endpoint” on page 1596

## t\_rcvdis() — Retrieve Information from Disconnect

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_rcvdis(int fd, struct t_discon *discon);
```

### General Description

Used to identify the cause of a disconnect. The argument *fd* identifies the local transport endpoint where the connection existed, and *discon* points to a *t\_discon* structure containing the following members:

```
struct netbuf  udata;
int            reason;
int            sequence;
```

The field *reason* specifies the reason for the disconnect through a protocol-dependent reason code, *udata* is always empty since the TCP transport provider does not support sending of user data with a disconnect, and *sequence* may identify an outstanding connect indication with which the disconnect is associated. The field *sequence* is only meaningful when *t\_rcvdis()* is issued by a passive transport user who has executed one or more *t\_listen()* functions and is processing the resulting connect indications. If a disconnect indication occurs, *sequence* can be used to identify which of the outstanding connect indications is associated with the disconnect.

If a user does not care if there is incoming data and does not need to know the value of *reason* or *sequence*, *discon* may be a null pointer. However, if a user has retrieved more than one outstanding connect indication (via *t\_listen()* ) and *discon* is a null pointer, the user will be unable to identify with which connect indication the disconnect is associated.

### Valid States

T\_DATAXFER,T\_OUTCON,T\_INCON(ocnt > 0)

### Returned Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *t\_errno* is set to indicate an error.

On failure, *t\_errno* is set to one of the following:

TBADF      The specified file descriptor does not refer to a transport endpoint.  
 TNODIS     No disconnect indication currently exists on the specified transport endpoint.

TBUFOVFLW

The number of bytes allocated for incoming data (maxlen) is greater than 0 but not sufficient to store the data. If *fd* is a passive endpoint

with `ocnt > 1`, it remains in state `T_INCON`; otherwise, the endpoint state is set to `T_IDLE`.

**TNOTSUPPORT**

This function is not supported by the underlying transport provider.

**TSYSERR** A system error has occurred during execution of this function.

**TOUTSTATE**

The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

**TPROTO** This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t\_errno*).

**Related Information**

- “xti.h” on page 56
- “t\_alloc() — Allocate a Library Structure” on page 1498
- “t\_connect() — Establish a Connection with Another Transport User” on page 1524
- “t\_listen() — Listen for a Connect Indication” on page 1577
- “t\_open() — Establish a Transport Endpoint” on page 1594
- “t\_snddis() — Send User-initiated Disconnect Request” on page 1621



## t\_rcvrel() — Acknowledge Receipt of an Orderly Release Indication

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_rcvrel(int fd);
```

### General Description

Since orderly release is not supported, this function always fails.

### Returned Value

The function always returns -1, and sets *t\_errno* to TNOTSUPPORT.

### Related Information

- “xti.h” on page 56
- “t\_getinfo() — Get Protocol-specific Service Information” on page 1566
- “t\_open() — Establish a Transport Endpoint” on page 1594
- “t\_sndrel() — Initiate an Orderly Release” on page 1623

## t\_rcvudata — Receive a Data Unit

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>

int t_rcvudata(int fd, struct t_unitdata *unitdata,
               int *flags);
```

### General Description

T\_CLTS service is not supported in this implementation, so this function always fails.

### Returned Value

This function always returns -1, and sets `t_errno` to `TNOTSUPPORT`.

### Related Information

- “xti.h” on page 56
- “t\_open() — Establish a Transport Endpoint” on page 1594
- “t\_sndudata — Send a Data Unit” on page 1624
- “t\_rcvuderr — Receive a Unit Data Error Indication” on page 1613

## t\_rcvuderr — Receive a Unit Data Error Indication

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_rcvuderr(int fd, struct t_uderr *uderr);
```

### General Description

T\_CLTS service is not supported in this implementation, so this function always fails.

### Returned Value

This function always returns -1, and sets `t_errno` to `TNOTSUPPORT`.

### Related Information

- “xti.h” on page 56
- “t\_open() — Establish a Transport Endpoint” on page 1594
- “t\_rcvudata — Receive a Data Unit” on page 1612
- “t\_sndudata — Send a Data Unit” on page 1624

## truncate() — Truncate a File to a Specified Length

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
int truncate(const char *path, off_t length);
```

### General Description

Truncates the file indicated by the *path* to the indicated *length*. The calling process must have write permission for the file. If the file size exceeds *length*, any extra data is discarded. If the file size is smaller than *length*, bytes between the old and new lengths are read as zeros. A change to the size of the file has no impact on the file offset.

If truncate() would cause the file size to exceed the soft file size limit for the process, truncate() will fail and a SIGXFSZ signal will be generated for the process.

When truncate() is completed successfully, it marks the st\_ctime and st\_mtime fields of the file. If truncate() is unsuccessful, the file is unchanged.

### Returned Value

If successful, truncate() returns 0. If unsuccessful, it returns -1 and sets errno to one of the following:

EFBIG	The file is too large. Cannot write beyond the file size limit.
EINTR	A signal was caught during execution
EINVAL	<i>path</i> does not refer to a regular file, or the length specified is incorrect.
EIO	An I/O error occurred while reading from or writing to a file system.
EISDIR	The file specified is a directory. The system cannot perform the requested function on a directory.
EROFS	The file resides on a read-only file system.

### Related Information

- “unistd.h” on page 53
- “open() — Open a File” on page 872

## tsched() — Schedule MTF Subtask

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	C only	

### Format

```
#include <mtf.h>
```

```
int tsched(int task_id, const char *func_name,...);
```

### General Description

The `tsched()` built-in library function is used to schedule parallel functions under MVS.

The `tsched()` library function schedules a parallel function (*func\_name*) to be executed in a subtask that has been attached and initialized by `tinit()`. The first argument, *task\_id*, can be used to specify the task where the parallel function is to be executed. The *task\_id* can be set to `MTF_ANY` to indicate that the parallel function can be run in any task or, *task\_id* can be set to a specific task number between 1 and the number of subtasks specified on `tinit()`.

You can call `tsched()` from your main C task as often as necessary to schedule parallel functions for execution. If all of the subtasks are busy running previously scheduled functions, each call in excess of the number of subtasks will cause *func\_name* to be run after previously scheduled functions in the first qualifying subtask that becomes available.

All scheduled functions must be computationally independent. A function cannot use variables that are modified by another scheduled function or the scheduling function. To determine if all other scheduled functions have completed, use the `tsyncro()` library function (see “`tsyncro()` — Wait for MTF Subtask Termination” on page 1628).

The name of the parallel function, *func\_name*, must not be longer than 8 characters. If it is, the name will be truncated to the first 8 characters with no warning.

Usually `tsched()` returns to the calling main task program before the scheduled parallel function has completed execution. Therefore, you must call `tsyncro()` to ensure that your parallel functions have completed execution.

If `tinit()` has not been successfully called prior to `tsched()`, `tsched()` indicates that MTF is inactive.

If `tinit()` is called by a program running under IMS, CICS, or DB2, the request will not be processed and the returned value will indicate that MTF calls are not supported under these systems.

**Note:** This function is *not* supported under the OS/390 UNIX services with the POSIX(ON) runtime option.

## Returned Value

*Table 42. Returned Values for tsched()*

Value	Meaning
MTF_OK	The parallel function was successfully scheduled for execution in a subtask.
EINACTIVE	MTF is inactive.
EBADLNKG	tsched() has been invoked via an invalid linkage. The header file mtf.h may have been missing from the source at compile.
ETASKID	The task_id specified is not valid.
EENTRY	The parallel function was not found in the parallel module.
ENOMEM	There was insufficient storage for MTF-internal areas.
ETASKABND	One or more subtasks have terminated abnormally.

**Note:** These values are macros. They can be found in the mtf.h header file.

## Related Information

- “mtf.h” on page 37
- “tinit() — Attach and Initialize Subtasks” on page 1575
- “tsyncro() — Wait for MTF Subtask Termination” on page 1628
- “terminate() — Terminate After Failures in C++ Error Handling” on page 1560.

## tsearch() — Binary Tree Search

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <search.h>
```

```
void *tsearch(const void *key, void **rootp,
              int (*compar)(const void *, const void *));
```

### General Description

The `tsearch()` function is used to build and access a binary search tree. The *key* argument is a pointer to an element to be accessed or stored. If there is a node in the tree whose element is equal to the value pointed to by *key*, a pointer to this found node is returned. Otherwise, the value pointed to by *key* is inserted (that is, a new node is created and the value of *key* is copied to this node), and a pointer to this node returned. Only pointers are copied, so the calling routine must store the data. The *rootp* argument points to a variable that points to the root node of the tree. A null pointer value for the variable pointed to by *rootp* denotes an empty tree; in this case, the variable will be set to point to the node which will be at the root of the new tree.

Comparisons are made with a user-supplied routine, the address of which is passed as the *compar* argument. This routine is called with two arguments, the pointers to the elements being compared. The user-supplied routine must return an integer less than, equal to or greater than 0, according to whether the first argument is to be considered less than, equal to or greater than the second argument. The comparison functions need not compare every byte, so arbitrary data may be contained in the elements in addition to the values being compared.

**Threading Behavior:** Since the tree is anchored by the user's *rootp* pointer, the tree storage is visible to the user and could be shared among threads. The user would be responsible for serializing access to a shared tree. There are no variables related to these functions which are internal to the library and/or give rise to multi-threading considerations.

### Special Behavior for C++

Because C and C++ linkage conventions are incompatible, `tsearch()` cannot receive a C++ function pointer as the comparator argument. If you attempt to pass a C++ function pointer to `tsearch()`, the compiler will flag it as an error. You can pass a C or C++ function to `tsearch()` by declaring it as `extern "C"`.

### Returned Value

If the node is found, the `tsearch()` function returns a pointer to it, otherwise it returns a pointer to the inserted item. A null pointer is returned if there is not enough space available to create a new node, or if *rootp* is a null pointer on entry.

No errors are defined.

### Related Information

- “search.h” on page 40
- “twalk() — Binary Tree Walk” on page 1636
- “tdelete() — Binary Tree Delete” on page 1555
- “tfind() — Binary Tree Find Node” on page 1563
- “bsearch() — Search Arrays” on page 137
- “hsearch() — Search Hash Tables” on page 647
- “lsearch() — Linear Search and Update” on page 775



## t\_snd() — Send Data or Expedited Data Over a Connection

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_snd(int fd, char *buf, unsigned int nbytes,
          int flags);
```

### General Description

Sends either normal or expedited data. The argument *fd* identifies the local transport endpoint over which data should be sent, *buf* points to the user data, *nbytes* specifies the number of bytes of user data to be sent, and *flags* specifies any optional flags described below:

#### T\_EXPEDITED

If set in *flags*, the data will be sent as expedited data and will be subject to the interpretations of the transport provider.

#### T\_MORE

Since the TCP transport provider does not support the concept of a TSDU, the T\_MORE flag is not meaningful and will be ignored if set.

By default, *t\_snd()* operates in synchronous mode and may wait if flow control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if *O\_NONBLOCK* is set (via *t\_open()* or *fcntl()*), *t\_snd()* will execute in asynchronous mode, and will fail immediately if there are flow control restrictions. The process can arrange to be informed when the flow control restrictions are cleared via either *t\_look()* or *select/poll*.

On successful completion, *t\_snd()* returns the number of bytes accepted by the transport provider. Normally this will equal the number of bytes specified in *nbytes*. However, if *O\_NONBLOCK* is set, it is possible that only part of the data will actually be accepted by the transport provider. In this case, *t\_snd()* will return a value that is less than the value of *nbytes*.

The size of each TSDU or ETSDU must not exceed the limits of the transport provider as specified by the current values in the TSDU or ETSDU fields in the *info* argument returned by *t\_getinfo()*. The error *TL00K* may be returned to inform the process that an event (for example, a disconnect) has occurred.

### Valid States

#### T\_DATAXFER

## Returned Value

On successful completion, `t_snd()` returns the number of bytes accepted by the transport provider. Otherwise, -1 is returned on failure and `t_errno` is set to indicate the error.

Note that in asynchronous mode, if the number of bytes accepted by the transport provider is less than the number of bytes requested, this may indicate that the transport provider is blocked due to flow control.

On failure, `t_errno` is set to one of the following:

**TBADF** The specified file descriptor does not refer to a transport endpoint.

**TBADDATA** Illegal amount of data:

- A single send was attempted specifying a TSDU (ETSDU) or fragment TSDU (ETSDU) greater than that specified by the current values of the TSDU or ETSDU fields in the *info* argument.
- Multiple sends were attempted resulting in a TSDU (ETSDU) larger than that specified by the current value of the TSDU or ETSDU fields in the *info* argument

**TBADFLAG** An invalid flag was specified.

**TFLOW** `O_NONBLOCK` was set, but the flow control mechanism prevented the transport provider from accepting any data at this time.

**TNOTSUPPORT**

This function is not supported by the underlying transport provider.

**TLOOK** An asynchronous event has occurred on this transport endpoint.

**TOUTSTATE**

The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

**TSYSERR** A system error has occurred during execution of this function.

**TPROTO** This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (`t_errno`).

It is important to remember that the transport provider treats all users of a transport endpoint as a single user. Therefore if several processes issue concurrent `t_snd()` calls then the different data may be intermixed. Multiple sends which exceed the maximum TSDU or ETSDU size may not be discovered by XTI. In this case an implementation-dependent error will result (generated by the transport provider) perhaps on a subsequent XTI call. This error may take the form of a connection abort, a `TSYSERR`, a `TBADDATA` or a `TPROTO` error. If multiple sends which exceed the maximum TSDU or ETSDU size are detected by XTI, `t_snd()` fails with `TBADDATA`.

## Related Information

- “xti.h” on page 56
- “`t_getinfo()` — Get Protocol-specific Service Information” on page 1566
- “`t_open()` — Establish a Transport Endpoint” on page 1594
- “`t_rcv()` — Receive Data or Expedited Data Sent Over a Connection” on page 1605

## t\_snddis() — Send User-initiated Disconnect Request

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_snddis(int fd, struct t_call *call);
```

### General Description

Initiates an abortive release on an already established connection, or rejects a connect request. The argument *fd* identifies the local transport endpoint of the connection, and *call* specifies information associated with the abortive release. The argument *call* points to a *t\_call* structure which contains the following members:

```
struct netbuf  addr;
struct netbuf  opt;
struct netbuf  udata;
int            sequence;
```

The values in *call* have different semantics, depending on the context of the call to *t\_snddis()*. When rejecting a connect request, *call* must be non-null and contain a valid value of *sequence* to uniquely identify the rejected connect indication to the transport provider. The *sequence* field is only meaningful if the transport connection is in the T\_INCON state. The *addr* and *opt* fields of *call* are ignored. In all other cases, *call* should be a null pointer, since its only use would be to specify user data to be passed on the disconnect request, which is not supported by the TCP transport provider.

*t\_snddis()* is an abortive disconnect. Therefore a *t\_snddis()* issued on a connection endpoint may cause data previously sent via *t\_snd()*, or data not yet received, to be lost (even if an error is returned).

Because of implementation restrictions, a *t\_snddis()* called on one descriptor referring to an endpoint will not affect descriptors in other processes referring to the same endpoint. If descriptors in multiple processes refer to the same endpoint, the endpoint will not actually be disconnected by a *t\_snddis* in one process. Multiple processes cooperating on an endpoint are responsible for providing their own explicit synchronization to support coordinated disconnects.

### Valid States

T\_DATAXFER, T\_OUTCON, T\_INCON(ocnt > 0)

### Returned Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *t\_errno* is set to indicate an error.

On failure, *t\_errno* is set to one of the following:

TBADF      The specified file descriptor does not refer to a transport endpoint.

**TOUTSTATE**

The function was issued in the wrong sequence on the transport endpoint referenced by *fd*.

**TBADDATA** The amount of user data specified was not within the bounds allowed by the transport provider.

**TBADSEQ** An invalid sequence number was specified, or a null *call* pointer was specified, when rejecting a connect request.

**TNOTSUPPORT**

This function is not supported by the underlying transport provider.

**TSYSERR** A system error has occurred during execution of this function.

**TLOOK** An asynchronous event, which requires attention, has occurred.

**TPROTO** This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t\_errno*).

**Related Information**

- “xti.h” on page 56
- “t\_connect() — Establish a Connection with Another Transport User” on page 1524
- “t\_getinfo() — Get Protocol-specific Service Information” on page 1566
- “t\_listen() — Listen for a Connect Indication” on page 1577
- “t\_open() — Establish a Transport Endpoint” on page 1594

## t\_sndrel() — Initiate an Orderly Release

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_sndrel(int fd);
```

### General Description

Since orderly release is not supported, t\_sndrel() always fails.

### Returned Value

The function always returns -1, and sets *t\_errno* to TNOTSUPPORT.

### Related Information

- “xti.h” on page 56
- “t\_getinfo() — Get Protocol-specific Service Information” on page 1566
- “t\_open() — Establish a Transport Endpoint” on page 1594
- “t\_rcvrel() — Acknowledge Receipt of an Orderly Release Indication” on page 1611

## t\_sndudata — Send a Data Unit

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_sndudata(int fd, struct t_unitdata *unitdata);
```

### General Description

T\_CLTS service is not supported in this implementation, so this function always fails.

### Returned Value

This function always returns -1, and sets `t_errno` to `TNOTSUPPORT`.

### Related Information

- “xti.h” on page 56
- “t\_open() — Establish a Transport Endpoint” on page 1594
- “t\_rcvudata — Receive a Data Unit” on page 1612
- “t\_rcvuderr — Receive a Unit Data Error Indication” on page 1613

## t\_strerror() — Produce an Error Message String

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
char *t_strerror(int errnum);
```

### General Description

Maps the error number in *errnum* that corresponds to an XTI error to a language-dependent error message string and returns a pointer to the string. The string pointed to should not be modified by the program, but may be overwritten by a subsequent call to the `t_strerror` function. The string is not terminated by a newline character. If the calling program is operating in any one of the C, POSIX, SAA or S370 locales, then the error message string describing the value in *t\_errno* is identical to the comments following the *t\_errno* codes defined in `<xti.h>`. Note that no message number is prefixed to the message text in this situation. If an error code is unknown, and the language is English, `t_strerror()` returns the string:

```
"<error>: error unknown"
```

where `<error>` is the error number supplied as input. In other languages, an equivalent text is provided.

### Valid States

All - except for T\_UNINIT

### Returned Value

The function `t_strerror()` returns a pointer to the generated message string.

### Related Information

- “xti.h” on page 56
- “t\_error() — Produce Error Message” on page 1561

## t\_sync() — Synchronize Transport Library

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_sync(int fd);
```

### General Description

Synchronizes the data structures managed by the transport library with information from the underlying transport provider, for the transport endpoint specified by *fd*. In doing so, it can convert an uninitialized file descriptor (obtained via `open()`, `dup()`, or as a result of a `fork()` and `exec()` ) to an initialized transport endpoint, assuming that the file descriptor referenced a transport endpoint, by updating and allocating the necessary library data structures. This function also allows two cooperating processes to synchronize their interaction with a transport provider.

For example, if a process forks a new process and issues an `exec()` , the new process must issue a `t_sync()` to build the private library data structure associated with a transport endpoint and to synchronize the data structure with the relevant provider information.

It is important to remember that the transport provider treats all users of a transport endpoint as a single user. If multiple processes are using the same endpoint, they should coordinate their activities so as not to violate the state of the transport endpoint. The function `t_sync()` returns the current state of the transport endpoint to the user, thereby enabling the user to verify the state before taking further action. This coordination is only valid among cooperating processes. It is possible that a process or an incoming event could change the endpoint's state after a `t_sync()` is issued.

If the transport endpoint is undergoing a state transition when `t_sync()` is called, the function will fail.

### Valid States

All - except for `T_UNINIT`

### Returned Value

On successful completion, the state of the transport endpoint is returned. Otherwise, a value of -1 is returned and `t_errno` is set to indicate an error. The state returned is one of the following:

`T_UNBND` Unbound.

`T_IDLE` Idle.

`T_OUTCON` Outgoing connection pending.

`T_INCON` Incoming connection pending.



**T\_DATAXFER**

Data transfer.

On failure, *t\_errno* is set to one of the following:

- TBADF**      The specified file descriptor does not refer to a transport endpoint. This error may be returned when the fd has been previously closed or an erroneous number may have been passed to the call.
- TSTATECHNG**      The transport endpoint is undergoing a state change.
- TSYSERR**      A system error has occurred during execution of this function.
- TPROTO**      This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (*t\_errno*).

**Related Information**

- “xti.h” on page 56
- “dup() — Duplicate an Open File Descriptor” on page 288
- “exec Functions” on page 322
- “fork() — Create a New Process” on page 422
- “open() — Open a File” on page 872

## tsyncro() — Wait for MTF Subtask Termination

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	C only	

### Format

```
#include <mtf.h>
```

```
int tsyncro(int MTF_ANY|MTF_ALL|nn);
```

### General Description

Waits for termination of parallel functions under MVS.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

The `tsyncro()` library function causes the main task to wait for the first subtask, a particular subtask, or all subtasks to finish executing all of the parallel functions that have been scheduled for them. You can monitor the completion of any, all, or a specific subtask by specifying:

`MTF_ANY` To wait for the completion of any subtask.

`MTF_ALL` To wait for the completion of all subtasks.

`nn` To wait for the completion of the subtask having a *task\_id* of *nn*. See “`tinit()` — Attach and Initialize Subtasks” on page 1575 for a description of *task\_id*.

You can invoke `tsyncro()` from your main task program as often as necessary.

If `tinit()` is called by a program running under IMS, CICS, or DB2, the request will not be processed and the returned value will indicate that MTF calls are not supported under these systems.

**Note:** This function is *not* supported under the OS/390 UNIX services with the POSIX(ON) runtime option.

### Returned Value

If successful, `tsyncro()` will always return a value suitable for use as a target task on a subsequent `tsched()`. In particular, the return codes on success depend on the nature of the `tsyncro()` call, and the state of the subtasks at the time of the `tsyncro()` call as follows:

Task_id passed to <code>tsyncro()</code>	Return code (if successful)
MTF_ANY (and at least one subtask not previously returned from a <code>tsyncro()</code> )	<i>nn</i> = <i>task_id</i> of first subtask to become free
MTF_ANY (and all subtasks have previously been returned from a <code>tsyncro()</code> )	MTF_ALL
MTF_ALL	MTF_ANY
<i>nn</i> = <i>task_id</i>	<i>nn</i>

If `tinit()` has not been successfully called prior to the `tsyncro()` call, `tsyncro()` indicates that MTF is inactive.

The returned values for `tsyncro()` have the following meanings:

Value	Meaning
EINACTIVE	MTF is inactive.
ETASKID	The <i>task_id</i> argument specified is out of range.
ETASKABND	One or more subtasks have terminated abnormally.

**Note:** These values are macros. They can be found in the `mtf.h` header file.

### Related Information

- “`mtf.h`” on page 37
- “`tsched()` — Schedule MTF Subtask” on page 1615
- “`tinit()` — Attach and Initialize Subtasks” on page 1575
- “`terminate()` — Terminate After Failures in C++ Error Handling” on page 1560.

## tterm() — Terminate Subtasks

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	C only	

### Format

```
#include <mtf.h>
```

```
int tterm(void);
```

### General Description

Terminates the MTF environment under MVS. The function is invoked by a main task to await the completion of all parallel functions that were scheduled by `tsched()` and to detach all subtasks and remove the MTF environment created by `tinit()`, see “`tinit()` — Attach and Initialize Subtasks” on page 1575.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use `LANGLVL(EXTENDED)`.

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with `LANGLVL(EXTENDED)`. When you use `LANGLVL(EXTENDED)` any relevant information in the header is also exposed.

If `tinit()` has not been successfully called prior to a `tterm()` call, `tterm()` indicates that MTF is already inactive.

If a `tterm()` call is not issued before main program termination, a system abend with completion code A03 will occur. The program's termination will terminate the main task while subtasks are still active even if all scheduled parallel functions have completed execution.

If `tinit()` is called by a program running under IMS, CICS, or DB2, the request will not be processed and the returned value will indicate that MTF calls are not supported under these systems.

**Note:** This function is *not* supported under the OS/390 UNIX services with the `POSIX(ON)` runtime option.

### Returned Value

Table 43. Returned Values for `tterm()`

Value	Meaning
MTF_OK	The subtasks have been detached and the MTF environment removed.
EINACTIVE	MTF is inactive.
ETASKABND	One or more subtasks have terminated abnormally.

**Note:** These values are macros. They can be found in the `mtf.h` header file (“`mtf.h`” on page 37).

**Related Information**

- “mtf.h” on page 37
- “tsched() — Schedule MTF Subtask” on page 1615
- “tinit() — Attach and Initialize Subtasks” on page 1575
- “tsyncro() — Wait for MTF Subtask Termination” on page 1628

## ttyname() — Get the Name of a Terminal

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
char *ttyname(int fildev);
```

### General Description

Returns a string containing the path name of the terminal associated with the given file descriptor, *fildev*. Subsequent calls to `ttyname()` may overwrite this string, because the pointer returned may point to static data.

### Returned Value

If successful, `ttyname()` returns a string containing a path name. If unsuccessful because *fildev* is not a terminal, or the path name cannot be determined, `ttyname()` returns a NULL pointer.

### Special Behavior for XPG4

`ttyname()` sets `errno` to indicate the error.

**EBADF**      The *fildev* argument is not a valid open file descriptor.

**ENOTTY**    The *fildev* argument is not associated with a terminal.

### Example

#### CBC3BT16

```
/* CBC3BT16
   This example provides the path name of the terminal associated with stdin.
*/
#define _POSIX_SOURCE
#include <unistd.h>
#include <stdio.h>

main() {
    char *ret, tty[40];

    if ((ret = ttyname(STDIN_FILENO)) == NULL)
        perror("ttyname() error");
    else {
        strcpy(tty, ret);
        printf("The ttyname associated with my stdin is %s\n", tty);
    }
}
```

### Output

The ttyname associated with my stdin is /dev/tty0000

**Related Information**

- “unistd.h” on page 53
- “isatty() — Test if Descriptor Represents a Terminal” on page 703
- “ctermid() — Generate Path Name for Controlling Terminal” on page 246

## ttyslot() — Find the Slot in the utmpx File of the Current User

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>
```

```
int ttyslot(void);
```

### General Description

The `ttyslot()` function returns the index of the current user's entry in the utmpx database. The current user's entry is an entry for which the `ut_line` member matches the name of a terminal device associated with any of the process' file descriptors 0, 1 or 2. The `ttyname()` function is used to obtain the terminal device. The `"/dev/"` part returned by `ttyname()` is not used when searching the utmpx database member `ut_line`.

### Returned Value

The `ttyslot()` function returns one of the following values: Upon successful completion, `ttyslot()` returns the index of the current user's entry in the utmpx database. The `ttyslot()` function returns -1 if an error was encountered while searching the database or if none of the file descriptors 0, 1, or 2 is associated with a terminal device.

No errors are defined.

### Related Information

- “`endutxent()` — Close the utmpx Database” on page 313
- “`ttyname()` — Get the Name of a Terminal” on page 1632



## t\_unbind() — Disable a Transport Endpoint

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <xti.h>
```

```
int t_unbind(int fd);
```

### General Description

Disables the transport endpoint specified by *fd* which was previously bound by `t_bind()`. On completion of this call, no further data or events destined for this transport endpoint will be accepted by the transport provider. An endpoint which is disabled by using `t_unbind()` can be enabled by a subsequent call to `t_bind()`.

Due to implementation-imposed restrictions, `t_unbind` does not affect descriptors in processes other than the caller which were derived from *fd* by normal descriptor inheritance. Processes cooperating on an endpoint in this way must explicitly provide their own synchronization for endpoint takedown.

### Valid States

T\_IDLE

### Returned Value

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and `t_errno` is set to indicate an error. On failure, `t_errno` is set to one of the following:

- TBADF      The specified file descriptor does not refer to a transport endpoint.
- TOUTSTATE      The function was issued in the wrong sequence.
- TLOOK      An asynchronous event has occurred on this transport endpoint.
- TSYSERR      A system error has occurred during execution of this function.
- TPROTO      This error indicates that a communication problem has been detected between XTI and the transport provider for which there is no other suitable XTI (`t_errno`).

### Related Information

- “xti.h” on page 56
- “t\_bind() — Bind an Address to a Transport Endpoint” on page 1504

## twalk() — Binary Tree Walk

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <search.h>
```

```
void *twalk(const void *root,
            void (*action)(const void *, VISIT, int));
```

### General Description

The `twalk()` function traverses a binary search tree. The *root* argument is a pointer to the root node of the tree to be traversed. (Any node in a tree may be used as the root for a walk below that node.) The argument *action* is the name of a routine to be invoked at each node. This routine is, in turn, called with three arguments. The first argument is the address of the node being visited. The structure pointed to by this argument is unspecified and must not be modified by the application, but is guaranteed that a pointer-to-node can be converted to pointer-to-pointer-to-element to access the element stored in the node. The second argument is a value from an enumeration data type:

```
typedef enum {preorder, postorder, endorder, leaf } VISIT;
```

(defined in the `<search.h>` header), depending on whether this is the first, second or third time that the node is visited (during a depth-first, left-to-right traversal of the tree), or whether the node is a leaf. The third argument is the level of the node in the tree, with the root being level zero.

Threading Behavior: see “`tsearch()` — Binary Tree Search” on page 1617.

### Special Behavior for C++

Because C and C++ linkage conventions are incompatible, `twalk()` cannot receive a C++ function pointer as the argument. If you attempt to pass a C++ function pointer to `twalk()`, the compiler will flag it as an error. You can pass a C or C++ function to `twalk()` by declaring it as `extern "C"`.

### Returned Value

The `twalk()` function returns no value.

No errors are defined.

### Related Information

- “`search.h`” on page 40
- “`tsearch()` — Binary Tree Search” on page 1617
- “`tdelete()` — Binary Tree Delete” on page 1555
- “`tfind()` — Binary Tree Find Node” on page 1563
- “`bsearch()` — Search Arrays” on page 137
- “`hsearch()` — Search Hash Tables” on page 647
- “`lsearch()` — Linear Search and Update” on page 775

## tzset() — Set the Time Zone

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 OS/390 UNIX	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <time.h>
```

```
void tzset(void);
```

### General Description

The `tzset()` function uses the value of the environment variable `TZ` to set time conversion information used by `ctime()`, `localtime()`, `mktime()`, and `strftime()`. If `TZ` is absent from the environment, or it is incorrect, time-conversion information is obtained from the `LC_TOD` locale category.

`tzset()` also sets the external variable `tzname`:

```
extern char *tzname[2] = {"std", "dst"};
```

Here, *std* and *dst* are Standard and Daylight Savings time zone names, specified by `TZ` or the `LC_TOD` local category, respectively.

`tzset()` is called by `ctime()`, `localtime()`, `mktime()`, `setlocale()`, and `strftime()`. It can also be called explicitly by an application program.

The format of `TZ` values recognized by `tzset()` is as follows:

```
stdoffset[dst[offset]][,rule]]
```

*std* and *dst* Indicate no fewer than three, but not more than `TZNAME_MAX`, bytes that are the designation for the standard (*std*) and daylight savings (*dst*) time zones. If more than `TZNAME_MAX` bytes are specified for *std* or *dst*, `tzset()` truncates to `TZNAME_MAX` bytes. Only *std* is required; if *dst* is missing, daylight savings time does not apply in this locale. Uppercase and lowercase letters are explicitly allowed. Any character except a leading colon (:) or digits, the comma (,), the minus (−), the plus (+), and the NULL character are permitted to appear in these fields. The meaning of these letters and characters is unspecified.

*offset* Indicates the value that must be added to the local time to arrive at coordinated universal time (UTC). *offset* has the form: `hh[:mm[:ss]]`. The minutes (mm) and seconds (ss) are optional. The hour (hh) is required and may be a single digit. *offset* following *std* is required. If no *offset* follows *dst*, daylight savings time is assumed to be 1 hour ahead of standard time. One or more digits may be used; the value is always interpreted as a decimal number. The hour must be between 0 and 24; minutes and seconds, if present, between 0 and 59. The difference between standard time *offset* and daylight savings time *offset*

must be greater than or equal to 0, but the difference may not be greater than 24 hours. Use of values outside of these ranges causes `tzset()` to use the `LC_TOD` category rather than the `TZ` environment variable for time conversion information. An *offset* preceded by a minus (–) indicates a time zone east of the Prime Meridian. A plus (+) preceding *offset* is optional and indicates the time zone west of the Prime Meridian.

*rule* Indicates when to change to and back from daylight savings time. The rule has the form: `date[/time],date[/time]`

The first date describes when the change from standard to daylight savings time occurs and the second date describes when the change back happens. Each time field describes when, in current local time, the change to the other time is made.

The format of date must be one of the following:

- Jn* The Julian day *n* ( $1 \leq n \leq 365$ ). Leap days are not counted. That is, in all years—including leap years—February 28 is day 59 and March 1 is day 60. It is impossible to explicitly refer to the occasional February 29.
- n* The zero-based Julian day ( $0 \leq n \leq 365$ ). Leap days are counted, and it is possible to refer to February 29.
- Mm.n.d* The *d*th day ( $0 \leq d \leq 6$ ) of week *n* of month *m* of the year ( $1 \leq n \leq 5$ , and  $1 \leq m \leq 12$ , where week 5 means “the last *d* day in month *m*,” which may occur in either the fourth or the fifth week). Week 1 is the first week in which the *d*th day occurs. Day zero is Sunday.

The time has the same format as *offset* except that no leading sign, minus (–) or plus (+), is allowed. The default, if time is not given, is 02:00:00.

If *dst* is specified and *rule* is not specified by `TZ` or in `LC_TOD` category, the default for the daylight savings time start date is M4.1.0 and for the daylight savings time end date is M10.5.0.

### Special Behavior for XPG4

`tzset()` sets the external variable *timezone* to the difference, in seconds, between Coordinated Universal Time (UTC) and local standard time. `tzset()` sets the external variable *daylight* to 0 if summer time conversions should never be applied for the time zone in use; otherwise to nonzero.

Since the external variables *timezone* and *daylight* are global to the process, they cannot be reliably used in a multi-threaded application or an application running from a DLL. The run-time library provides two special functions, `__tzzone()` and `__dlght()` that return the address of thread-specific versions of these external variables. See “External Variables” on page 66.

### Special Behavior for OS/390 UNIX Services

The `tzset()` function only parses the `TZ` environment variable if it is called from the initial processing thread (IPT) by a threaded application.

**Note** This function is sensitive to time zone information which is provided by:

- The TZ environmental variable when POSIX(ON) and TZ is correctly defined, or by the \_TZ environmental variable when POSIX(OFF) and \_TZ is correctly defined.
- The LC\_TOD category of the current locale if POSIX(OFF) or TZ is not defined.

The time zone external variables tzname, timezone, and daylight declarations remain feature test protected in time.h.

## Returned Value

There is no returned value. There are no documented errnos.

## Example

### CBC3BT17

```
/* CBC3BT17
   This example set time conversion information for Eastern Standard and
   Eastern Daylight Savings Time in the United States.
*/
#define _POSIX_SOURCE
#include <env.h>
#include <time.h>

int main(void)
{
    setenv("TZ", "EST5EDT", 1);
    tzset();
}
```

## Related Information

- “time.h” on page 51
- “asctime() — Convert Time to Character String” on page 106
- “ctime() — Convert Time to a Character String” on page 250
- “gmtime() — Convert Time to Broken-Down UTC Time” on page 640
- “localdtconv() — Date/Time Formatting Convention Inquiry” on page 754
- “localtime() — Convert Time and Correct for Local Time” on page 758
- “mktime() — Convert Local Time” on page 828
- “strftime() — Convert to Formatted Time” on page 1432
- “time() — Determine Current Time” on page 1570

## ualarm() — Set the Interval Timer

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON) MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
```

```
useconds_t ualarm(useconds_t uscs, useconds_t intrval)
```

### General Description

The `ualarm()` function causes the SIGALRM signal to be generated for the calling process after the number of real-time microseconds specified by the *uscs* argument has elapsed. When the *intrval* argument is nonzero, repeated time-out notification occurs with a period in microseconds specified by the *intrval* argument. If the notification signal, SIGALRM, is not caught or ignored, the calling process is terminated.

The `ualarm()` function is a simplified interface to `setitimer()` and uses the ITIMER\_REAL interval timer.

### Returned Value

The `ualarm()` function returns the number of microseconds remaining from the previous `ualarm()`, `alarm()`, or `setitimer(ITIMER_REAL)` call. If no timeouts are pending, `ualarm()` returns 0.

No `errno`s are defined for the `ualarm()` function.

### Related Information

- “`unistd.h`” on page 53
- “`alarm()` — Set an Alarm” on page 102
- “`sigaction()` — Examine or Change a Signal Action” on page 1295
- “`setitimer()` — Set Value of an Interval Timer” on page 1232
- “`sleep()` — Suspend Execution of a Thread” on page 1364
- “`usleep()` — Suspend Execution for an Interval” on page 1663

## \_\_ucreate() — Create a Heap Using User-Provided Storage

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	OS/390 V2R5

### Format

```
#include <uheap.h>
```

```
__uheapid_t __ucreate(void *block,
                      size_t size,
                      __uheap_cellpool_attrib_table_t *
                        cellpool_attrib_table,
                      void *rsvd1,
                      void *rsvd2,
                      void *rsvd3,
                      void *rsvd4);
```

### General Description

The \_\_ucreate() function creates a heap out of storage that is provided by the caller. The heap is divided up into cell pools based on the information provided in the cellpool\_attrib\_table. Up to 6 cell pools can be created within the heap. Note that this is a fixed-size heap; when storage within a given cell pool is exhausted, no additional storage will be allocated. \_\_ucreate() returns a uheapid that is used to identify the heap on subsequent user-created heap function calls, such as \_\_umalloc(), \_\_ufree(), and \_\_uheapreport() calls.

#### Parameter

#### Description

*block*

A pointer to the storage which is to be used for the heap.

*size*

The size of the block of storage. Note that Language Environment reserves approximately 328 bytes of this storage for use in allocating heap management control blocks. Additional storage is reserved if storage report usage statistics are being collected for the heap. The amount of this storage is related to the largest cell size and the granularity of the statistics, and is calculated as:  
 storage amount =  
 ((largestcellsize+granularity-1)/granularity)\*4.

*cellpool\_attrib\_table*

A pointer to a structure describing the attributes of the cell pools to be created by \_\_ucreate().

The first field of the structure, number\_of\_pools, indicates the number of cell pools to be created. Up to 6 cell pools can be created in the heap.

The second field of the structure, granularity, indicates the granularity to which storage usage statistics is to be collected. This value must be zero, or a power of 2 greater than or equal to 8. If the value is zero, then statistics are not collected.

Following these words are pairs of words describing the attributes of each cell pool in the heap:

The first field in the pair, size, is the size of the cell in the cell pool. The cell size must be a multiple of 8 and greater than or equal to 8. Note that Language Environment adds an additional 8 bytes to the size of the cell for use in managing the cells. The second field in the pair, percentage, is the percentage of the total block size to be allocated for the cell pool.

*rsvd1-rsvd4*

Reserved for future use.

### Returned Value

The \_\_ucreate() function returns a uheapid if the request is successful. The \_\_ucreate() function returns -1 if it is not successful. If \_\_ucreate is not successful because an input is not valid, then \_\_ucreate() sets errno to EINVAL as well.

### Related Information

- “uheap.h” on page 52
- “\_\_umalloc() — Allocate Storage from a User-Created Heap” on page 1646
- “\_\_ufree() — Return Storage to a User-Created Heap” on page 1643
- “\_\_uheapreport() — Produce a Storage Report for a User-Created Heap” on page 1644



## \_\_ufree() — Return Storage to a User-Created Heap

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	OS/390 V2R5

### Format

```
#include <uheap.h>
```

```
void __ufree(__uheapid_t heapid,
             void        *ptr);
```

### General Description

The \_\_ufree() function returns storage to the heap identified by the heapid. If the returned storage does not belong to the given heap, the result is unpredictable.

#### Parameter

#### Description

*heapid*

The identifier of the user-created heap to which the storage is to be returned.

*ptr*

A pointer to the storage to be returned to the heap.

### Returned Value

None

### Related Information

- “uheap.h” on page 52
- “\_\_ucreate() — Create a Heap Using User-Provided Storage” on page 1641
- “\_\_uheapreport() — Produce a Storage Report for a User-Created Heap” on page 1644
- “\_\_umalloc() — Allocate Storage from a User-Created Heap” on page 1646

## \_\_uheapreport() — Produce a Storage Report for a User-Created Heap

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	OS/390 V2R5

### Format

```
#include <uheap.h>
```

```
void __uheapreport(__uheapid_t heapid);
```

### General Description

The \_\_uheapreport() function generates a report of the storage used within the user-created heap identified by *heapid*. The report is directed to the ddname specified in the MSGFILE run-time option. The report format is similar to the heap pools portion of the storage report generated for the RPTSTG run-time option.

Statistics for the user-created heap will only be collected if the granularity field of the cellpool\_attrib\_table passed to \_\_ucreate() is non-zero and a valid value.

#### Parameter

#### Description

*heapid*

The identifier of the user-created heap for which a report is to be produced.

### Returned Value

None

### Related Information

- “uheap.h” on page 52
- “\_\_ucreate() — Create a Heap Using User-Provided Storage” on page 1641
- “\_\_umalloc() — Allocate Storage from a User-Created Heap” on page 1646
- “\_\_ufree() — Return Storage to a User-Created Heap” on page 1643

## ulimit() — Get/Set Process File Size Limits

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE
#include <ulimit.h>
```

```
long int ulimit(int cmd, ...);
```

### General Description

The `ulimit()` function provides control over the process file size limits. The *cmd* argument controls whether the file size limits are obtained or set. The *cmd* argument can be one of the following, defined in `<ulimit.h>`:

#### UL\_GETFSIZE

Return the soft (current) file size limit of the process. The limit returned is in units of 512-byte blocks. The return value is the integer portion of the soft file size limit divided by 512. Refer to the `setrlimit()` function, `RLIMIT_FSIZE` resource description for more detail.

#### UL\_SETFSIZE

Set the hard (maximum) and soft (current) file size limits for output operations of the process. The value of the second argument is used, and is treated as a long int. Refer to the `setrlimit()` function, `RLIMIT_FSIZE` resource description for more detail and restrictions on lowering and raising file size limits. The hard and soft file size limits are set to the specified value multiplied by 512. The new file size limit (in 512 byte increments) is returned.

### Returned Value

If successful, `ulimit()` returns the value of the requested limit.

If unsuccessful, `ulimit()` returns `-1`, and returns the error value in `errno`. The following are the possible values of `errno`:

`EINVAL`      The *cmd* argument is not valid.

`EPERM`      To increase the file size limit, superuser authority is required.

### Related Information

- “`ulimit.h`” on page 53
- “`getrlimit()` — Control Maximum Resource Consumption” on page 591
- “`setrlimit()` — Control Maximum Resource Consumption” on page 1264

## \_\_umalloc() — Allocate Storage from a User-Created Heap

### Standards

Standards / Extensions	C or C++	Dependencies
Language Environment	both	OS/390 V2R5

### Format

```
#include <uheap.h>
```

```
void *__umalloc(__uheapid_t heapid,
                size_t      size);
```

### General Description

The \_\_umalloc() function allocates storage from the heap identified by the heapid. \_\_umalloc() will search for an available cell within the cell pool that contains cells at least as large and closest in size to the requested size.

#### Parameter

#### Description

*heapid*

The identifier of the user-created heap from which the storage is to be allocated.

*size*

The amount of storage to be allocated.

### Returned Value

If successful, \_\_umalloc() returns a pointer to the reserved cell. The returned value is NULL if a cell of the required size is not available, if size was larger than the largest available cell size, or if size was specified as 0. If \_\_umalloc() returns a NULL because there is not enough storage or if the requested size was too large, it will also return an error value in errno. The following are the possible values of errno:

ENOMEM Insufficient memory is available

E2BIG Requested amount of storage is larger than the largest available cell size

### Related Information

- “uheap.h” on page 52
- “\_\_ucreate() — Create a Heap Using User-Provided Storage” on page 1641
- “\_\_ufree() — Return Storage to a User-Created Heap” on page 1643
- “\_\_uheapreport() — Produce a Storage Report for a User-Created Heap” on page 1644

umask() — Set and Retrieve File Creation Mask

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

Format

```
#define _POSIX_SOURCE
#include <sys/stat.h>

mode_t umask(mode_t newmask);
```

General Description

Changes the file creation mask of the process. *newmask* specifies new file-permission bits for the file creation mask of the process.

This mask restricts the setting of (or turns off) file-permission bits specified in the ‘mode’ argument used with all `open()`, `creat()`, `mkdir()`, and `mkfifo()` functions issued by the current process. File-permission bits set to 1 in the file creation mask are set to 0 in the file-permission bits of files that are created by the process.

For example, if a call to `open()` specifies a *mode* argument with file-permission bits, the file creation mask of the process affects the *mode* argument; bits that are 1 in the mask are set to 0 in the *mode* argument, and therefore in the mode of the created file.

Only the file-permission bits of the new mask are used. The meaning of other bits is implementation defined. For more information on these symbols, refer to “`chmod()` — Change the Mode of a File or Directory” on page 174.

The `_EDC_UMASK_DFLT` environment variable controls how the C runtime library sets the default umask. If OS/390 UNIX services are available, the run-time library establishes a default umask value of 022 octal, and queries the value of the `_EDC_UMASK_DFLT` environment variable. `_EDC_UMASK_DFLT` can have the following values:

- NO (case insensitive)  
The library should not change the umask.
- A valid octal value  
The library should use this as the default value for the umask.

Any other value for the environment variable causes the runtime library to use 022 octal as the umask value.

## Returned Value

umask() is always successful and returns the previous value of the file creation mask. There are no documented errno's for this function.

## Example CBC3BU01

```
/* CBC3BU01
   This example will work only from OS/390 C, not OS/390 C++.
*/
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

main() {
    int fd;

    if (umask(S_IRWXG) != 0)
        perror("umask() error");
    else if ((fd = creat("umask.file", S_IRWXU|S_IRWXG)) < 0)
        perror("creat() error");
    else {
        system("ls -l umask.file");
        close(fd);
        unlink("umask.file");
    }
}
```

## Output

```
-rwx-----  1 WELLIE  SYS1          0 Apr 19 14:50 umask.file
```

## Related Information

- “sys/stat.h” on page 48
- “chmod() — Change the Mode of a File or Directory” on page 174
- “creat() — Create a New File or Rewrite an Existing One” on page 233
- “mkdir() — Make a Directory” on page 817
- “mkfifo() — Make a FIFO Special File” on page 820
- “open() — Open a File” on page 872

## umount() — Remove a Virtual File System

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 4.3

### Format

```
#include <sys/stat.h>
```

```
int umount(const char *filesystem, mtm_t mtm);
```

### General Description

Removes a file system from the file hierarchy, or changes the mount mode of a mounted file system between read-only and read-write. The *filesystem* argument is a null-terminated string containing the file-system name. This is the same name that was specified when the file system was mounted.

In order to umount a file system, the caller must be an authorized program, or must be running for a user with appropriate privileges.

The *mtm* argument can be one of the following:

#### MTM\_UMOUNT

A normal unmount request. If the files in the named file system are not in use, the unmount is done. Otherwise, the request is rejected.

#### MTM\_DRAIN

An unmount drain request. The requestor is willing to wait for all uses of this file system to be ended normally before the file system is unmounted.

#### MTM\_IMMED

An immediate unmount request. The file system is unmounted immediately, forcing any users of files in the specified file system to fail. All data changes that were made up to the time of the request are saved. If there is a problem saving the data, the unmount request fails.

#### MTM\_FORCE

A forced unmount request. The file system is unmounted immediately, forcing any users of any files in the specified file system to fail. All data changes that were made up to the time of the request are saved. If there is a problem saving the data, the request continues, and the data may be lost. To prevent lost data, issue an immediate umount() request before issuing a forced umount() request.

#### MTM\_RESET

A reset unmount request. This allows a previous unmount drain request to be stopped.

#### MTM\_REMOUNT

A remount request. This changes the mount mode of a file system from read-only to read-write or from read-write to read-only. If neither MTM\_RDONLY nor MTM\_RDWR is specified, the mode is set to the

opposite of its current state. If a mode is specified, it must be the opposite of the current state.

### Returned Value

If successful, `umount()` returns the value 0. If unsuccessful, it returns the value `-1`, and sets `errno` to one of the following:

EBUSY	The file system is busy, for one of these reasons: <ul style="list-style-type: none"> <li>• A <code>umount()</code> (<code>MTM_UMOUNT</code>) was requested, and the file system still has open files or other file systems mounted under it.</li> <li>• A file system is currently mounted on the requested file system.</li> <li>• A RESET was requested, and the previous <code>umount()</code> request was either immediate or forced, instead of a drain request.</li> <li>• There is a <code>umount()</code> request already in progress for the specified file system.</li> <li>• A <code>umount()</code> drain request is being reset.</li> </ul>
EINTR	<code>umount()</code> was interrupted by a signal.
EINVAL	A parameter was incorrectly specified. Verify the spelling of the file-system name and the setting of <i>mtm</i> .
EIO	An I/O error occurred.
EPERM	Superuser authority is required to issue an unmount.

### Example CBC3BU02

```

/* CBC3BU02
   This example removes a file system from the hierarchy, using umount().
*/
#include <sys/stat.h>
#include <stdio.h>
#include <unistd.h>

main() {
    char HFS[]="POSIX.NEW.HFS";
    char filesystype[9]="HFS    ";

    puts("before umount()");
    system("df -Pk");
    if (umount(HFS, MTM_UMOUNT) != 0)
        perror("umount() error");
    else {
        puts("After umount()");
        system("df -Pk");
    }
}

```

### Output



before umount()					
Filesystem	1024-blocks	Used	Available	Capacity	Mounted on
POSIX.NEW.HFS	200	20	180	10%	/new_fs
POSIX.ROOT.FS	9600	8180	1420	85%	/
After umount()					
Filesystem	1024-blocks	Used	Available	Capacity	Mounted on
POSIX.ROOT.FS	9600	8180	1420	85%	/

Related Information

- “sys/stat.h” on page 48
- “mount() — Make a File System Available” on page 838

# uname() — Display Current Operating System Name

## Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

## Format

```
#define _POSIX_SOURCE
#include <sys/utsname.h>

int uname(struct utsname *name);
```

## General Description

Gets information identifying the operating system you are running on. The argument *name* points to a memory area where `uname()` can store a structure describing the operating system the process is running on.

| To provide an ASCII input/output format for applications using this function, define  
| feature test macro `__LIBASCII` as described on page 22.

The information about the operating system is returned in a `utsname` structure, which has the following elements:

- `char *sysname;` The name of the implementation of the operating system.
- `char *nodename;`  
The node name of this particular machine. The node name is set by the `SYSNAME` sysparm (specified at IPL), and usually differentiates machines running at a single location.
- `char *release;` The current release level of the implementation.
- `char *version;` The current version level of the release.
- `char *machine;` The name of the hardware type the system is running on.

Each of these elements is a normal C string, terminated with a NULL character.

Table 44 lists the operating system information returned by `uname()`.

Table 44. *uname()* Operating System Information

Operating System	Sysname	Release	Version
OS/390 UNIX MVS Rel. 3	MVS	2.2	5
OS/390 Rel. 1	MVS	100	1
OS/390 Rel. 2	OS/390	02.00	01

## Returned Value

If successful, `uname()` returns a nonnegative value. If unsuccessful, it returns the value `-1`, and it may set `errno` to indicate the reason for the failure, but no `errno` values are specified by the POSIX.1 standard.

## Example

### CBC3BU03

```

/* CBC3BU03
   This example gets information about the system you are running on.
*/
#define _POSIX_SOURCE
#include <sys/utsname.h>
#include <stdio.h>

main() {
    struct utsname uts;

    if (uname(&uts) < 0)
        perror("uname() error");
    else {
        printf("Sysname:  %s\n", uts.sysname);
        printf("Nodename: %s\n", uts.nodename);
        printf("Release:  %s\n", uts.release);
        printf("Version:  %s\n", uts.version);
        printf("Machine:  %s\n", uts.machine);
    }
}

```

## Output

```

Sysname:  OS/390
Nodename: SY1
Release:  02.00
Version:  01
Machine:  3090*

```

## Related Information

- “sys/utsname.h” on page 50

## unexpected() — Handle Exception Not Listed in Exception Specification

### Standards

Standards / Extensions	C or C++	Dependencies
	C++ only	

### Format

```
#include <unexpected.h>
```

```
void unexpected();
```

### General Description

Part of the OS/390 C++ error handling mechanism. It is called when a function throws an exception that is not listed in its exception specification. The `unexpected()` function calls the function most recently supplied as an argument to `set_unexpected()`. If `set_unexpected()` has not yet been called, then `unexpected()` calls `terminate()`.

In a multi-threaded environment, if a thread throws an exception that is not listed in its exception specification, then `unexpected()` is called. The default for `unexpected()` is to call `terminate()`, which defaults to calling `abort()`, which then causes a SIGABRT signal to be generated to the thread issuing the throw. If the SIGABRT signal is not caught, the process is terminated. You can replace the default `unexpected()` behavior for all threads in the process by using the `set_unexpected()` function. One possible use of `set_unexpected()` is to call a function which issues a `pthread_exit()`. If this is done, a throw of a condition by a thread that is not in the exception specification results in thread termination, but not process termination.

### Returned Value

Returns void.

Refer to the *OS/390 C/C++ Language Reference* for more information about OS/390 C++ exception handling, including the `unexpected()` function.

### Related Information

- “`terminate()` — Terminate After Failures in C++ Error Handling” on page 1560
- “`set_unexpected()` — Register a Function for `unexpected()`” on page 1281

## ungetc() — Push Character onto Input Stream

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2	both	

### Format

```
#include <stdio.h>
```

```
int ungetc(int c, FILE *stream);
```

### General Description

Pushes the character specified by the value of *c* converted to the unsigned char back onto the given input *stream*. The pushed back characters are returned by any subsequent read on the same stream in the reverse order of their pushing. That is, the last character pushed will be returned first.

Up to 4 characters can be pushed back to a given input stream. You can call `ungetc()` up to 4 times consecutively; this will result in 4 characters being pushed back in total.

The *stream* must be open for reading. A subsequent read operation on the *stream* starts with *c*. You cannot push EOF back on the stream using `ungetc()`. A successful call to the `ungetc()` function clears the EOF indicator for the stream.

Characters pushed back by `ungetc()`, and subsequently not read in, will be erased if a `fseek()`, `fsetpos()`, `rewind()`, or `fflush()` function is called before the character is read from the *stream*. After all the pushed back characters are read in, the file position indicator is the same as it was before the characters were pushed back.

Each character of pushback backs up the file position by one byte. This affects the value returned by `ftell()` or `fgetpos()`, the result of an `fseek()` using `SEEK_CUR`, or the result of an `fflush()`. For example, consider a file that contains: a b c d e f g h

After you have just read 'a', the current file position is at the start of 'b'. The following operations will all result in the file position being at the start of 'a', ready to read 'a' again.

```

/* 1 */      ungetc('a', fp);
              fflush(fp); /* flushes ungetc char and keeps position */

/* 2 */      ungetc('a', fp);
              pos = ftell(fp); /* points to first character */
              fseek(fp, pos, SEEK_SET);

/* 3 */      ungetc('a',fp);
              fseek(fp, 0, SEEK_CUR) /* starts at new file pos'n */

/* 4 */      ungetc('a', fp);
              fgetpos(fp, &fpos); /* gets position of first char */
              fsetpos(fp, &fpos);

```

You can use the environment variable `_EDC_COMPAT` to cause a OS/390 C/C++ application to ignore `ungetc()` characters for `fflush()`, `fgetpos()`, and `fseek()` using `SEEK_CUR`. For more details, see the *OS/390 C/C++ Programming Guide*.

The `ungetc()` function is not supported for files opened with `type=record`.

`ungetc()` has the same restriction as any read operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

## Returned Value

Returns the integer argument `c` converted to an unsigned char or EOF if `c` cannot be pushed back.

The `ungetc()` function is treated as a read operation. A flush or reposition is required after a call to `ungetc()` and before the next write operation.

## Example CBC3BU04

```

/* CBC3BU04
   In this example, the while statement reads decimal digits from an input
   data stream by using arithmetic statements to compose the numeric
   values of the numbers as it reads them.
   When a nondigit character appears before the EOF, ungetc() replaces it
   in the input stream so that later input functions can process it.
*/
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    FILE *stream;
    int ch;
    unsigned int result = 0;

    stream = fopen("myfile.dat","r+");
    while ((ch = getc(stream)) != EOF && isdigit(ch))
    {
        result = result * 10 + ch - '0';
    }
    printf("result is %i\n",result);
    if (ch != EOF)

```

```
{
    ungetc(ch,stream);          /* Put the nondigit character back */
    ch=getc(stream);
    printf("value put back was %c\n",ch);
}
```

### Related Information

- “stdio.h” on page 43
- “fflush() — Write Buffer to File” on page 385
- “fseek() — Change File Position” on page 474
- “getc() - getchar() — Read a Character” on page 499
- “putc() - putchar() — Write a Character” on page 1052

## ungetwc() — Push a Wide Character onto a Stream

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#include <wchar.h>
```

```
wint_t ungetwc(wint_t wc, FILE *stream);
```

### General Description

Pushes the wide character specified by *wc* back onto the input stream pointed to by *stream*. The pushed-back wide characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (with the stream pointed to by *stream*) to a file positioning function (*fseek()*, *fsetpos()*, or *rewind()*) discards any pushed-back wide characters for the stream. The external storage corresponding to the stream is unchanged. There is always at least one wide character of push back.

If the value of *wc* equals that of the macro *WEOF*, the operation fails and the input stream is unchanged.

A successful call to the *ungetwc()* function clears the EOF indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed-back wide characters is the same as it was before the wide characters were pushed back.

For a text stream, the file position indicator is backed up by one wide character. This affects *ftell()*, *fflush()*, *fseek()* via *SEEK\_CUR*, and *fgetpos()*. The environment variable, *\_EDC\_COMPAT* can be used to cause a pushed-back wide char to be ignored by *fflush()*, *fseek()* with *SEEK\_CUR*, and *fgetpos()*. For details, see the *OS/390 C/C++ Programming Guide*.

For a binary stream, the position indicator is unspecified until all characters are read or discarded, unless the last character is pushed back, in which case the file position indicator is backed up by one wide character. This affects *ftell()* and *fseek()* with *SEEK\_CUR*, *fgetpos()*, and *fflush()*. The environment variable *\_EDC\_COMPAT* can be used to cause the pushed-back wide character to be ignored by *fflush()*, *fgetpos()*, and *fseek()*.

*ungetwc()* is not supported for files opened with *type=record*.

*ungetwc()* has the same restriction as any read operation for a read immediately following a write, or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.



## Returned Value

Returns the wide character pushed back after conversion, or WEOF if the operation fails.

## Notes:

- For OS/390 C/C++ applications, only 1 wide character can be pushed back.
- The position on the stream after a successful `ungetwc()` is one wide character prior to the current position. See the *OS/390 C/C++ Programming Guide* for details on backing up a wide char.

## Example

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    FILE    *stream;
    wint_t   wc;
    unsigned int result = 0;
    :
    while ((wc = fgetwc(stream)) != WEOF && iswdigit(wc))
        result = result * 10 + wc - L'0';

    if (wc != WEOF)
        ungetwc(wc, stream);
        /* Put the nondigit wide character back */
}
```

## Related Information

- “wchar.h” on page 54
- “fflush() — Write Buffer to File” on page 385
- “fseek() — Change File Position” on page 474
- “fsetpos() — Set File Position” on page 477
- “fgetwc() — Get Next Wide Character” on page 394
- “fputwc() — Output a Wide-Character” on page 452

# unlink() — Remove a Directory Entry

## Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

## Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
int unlink(const char *pathname);
```

## General Description

Removes a directory entry. This unlink() deletes the link named by *pathname* and decrements the link count for the file itself.

*pathname* can refer to a pathname, a link, or a symbolic link. If the *pathname* refers to a symbolic link, unlink() removes the symbolic link but not any file or directory named by the contents of the symbolic link.

If the link count becomes 0 and no process currently has the file open, the file itself is deleted. The space occupied by the file is freed for new use, and the current contents of the file are lost. If one or more processes have the file open when the last link is removed, unlink() removes the link, but the file itself is not removed until the last process closes the file.

unlink() cannot be used to remove a directory; use rmdir() instead.

If unlink() succeeds, the change and modification times for the parent directory are updated. If the file's link count is not 0, the change time for the file is also updated. If unlink() fails, the link is not removed.

To provide an ASCII input/output format for applications using this function, define feature test macro \_\_LIBASCII as described on page 22.

## Returned Value

If successful, unlink() returns the value 0. If unsuccessful, it returns the value -1, and sets errno to one of the following:

- EACCES     The process did not have search permission for some component of *pathname*, or did not have write permission for the directory containing the link to be removed.
- EBUSY     *pathname* cannot be unlinked because it is currently being used by the system or some other process.
- ELOOP     A loop exists in symbolic links. This error is issued if more than POSIX\_SYMLLOOP symbolic links are detected in the resolution of *pathname*.

## ENAMETOOLONG

*pathname* is longer than PATH\_MAX characters, or some component of *pathname* is longer than NAME\_MAX characters while \_POSIX\_NO\_TRUNC is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH\_MAX. The PATH\_MAX and NAME\_MAX values can be determined using pathconf().

ENOENT *pathname* does not exist, or it is an empty string.

ENOTDIR Some component of the *pathname* prefix is not a directory.

EPERM *pathname* is a directory, and unlink() cannot be used on directories.

EROFS The link to be removed is on a read-only file system.

## Example CBC3BU06

```
/* CBC3BU06
   This example removes a directory entry, using unlink().
   */
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

main() {
    int fd;
    char fn[]="unlink.file";

    if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        close(fd);
        if (unlink(fn) != 0)
            perror("unlink() error");
    }
}
```

## Related Information

- “unistd.h” on page 53
- “remove() — Delete File” on page 1133
- “close() — Close a File” on page 191
- “link() — Create a Link to a File” on page 749
- “open() — Open a File” on page 872
- “rmdir() — Remove a Directory” on page 1147

## unlockpt() — Unlock a Pseudoterminal Master/Slave Pair

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>
int unlockpt(int fildes);
```

### General Description

The `unlockpt()` function unlocks the slave pseudoterminal device associated with the master to which *fildes* refers.

Portable applications must call `unlockpt()` before opening the slave side of a pseudoterminal device.

### Returned Value

Upon successful completion, `unlockpt()` returns 0. Otherwise, it returns -1 and sets `errno` to indicate the error.

The `unlockpt()` function will fail if:

- EACCESS** Either a `grantpt()` has not yet been issued, or an `unlockpt()` has already been issued. An `unlockpt()` must be issued after a `grantpt()`, and can only be issued once.
- EBADF** The *fildes* argument is not a file descriptor open for writing.
- EINVAL** The *fildes* argument is not associated with a master pseudoterminal device.

### Related Information

- “`grantpt()` — Grant Access to the Slave Pseudoterminal Device” on page 642
- “`open()` — Open a File” on page 872
- “`ptsname()` — Get Name of the Slave Pseudoterminal Device” on page 1051

## usleep() — Suspend Execution for an Interval

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON) MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
```

```
int usleep( useconds_t useconds);
```

### General Description

The `usleep()` function suspends thread execution for the number of microseconds specified by the *useconds* argument. Because of other activity, or because of the time spent processing the call, the actual suspension time may be longer than the amount of time specified.

The *useconds* argument must be less than 1,000,000. If the value of *useconds* is 0, then the call has no effect.

The `usleep()` function will not interfere with a previous setting of the real-time interval timer. If the thread has set this timer prior to calling `usleep()`, and if the time specified by *useconds* equals or exceeds the interval timer's prior setting, then the thread will be woken up when the previously set timer interval expires.

### Returned Value

On successful completion, `usleep()` returns 0. Otherwise, it returns -1 and sets `errno` to indicate the error.

Errno values defined for `usleep()`:

`EINVAL`      The *useconds* argument was greater than or equal to 1,000,000.

### Related Information

- “`unistd.h`” on page 53
- “`alarm()` — Set an Alarm” on page 102
- “`sigaction()` — Examine or Change a Signal Action” on page 1295
- “`setitimer()` — Set Value of an Interval Timer” on page 1232
- “`sleep()` — Suspend Execution of a Thread” on page 1364
- “`alarm()` — Set the Interval Timer” on page 1640

utime() — Set File Access and Modification Times

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

Format

```
#define _POSIX_SOURCE
#include <utime.h>

int utime(const char *pathname, const struct utimbuf *newtimes);
```

General Description

Sets the access and modification times of *pathname* to the values in the *utimbuf* structure. If *newtimes* is a NULL pointer, the access and modification times are set to the current time.

Normally, the effective user ID (UID) of the calling process must match the owner UID of the file, or the calling process must have appropriate privileges. However, if *newtimes* is a NULL pointer, the effective UID of the calling process must match the owner UID of the file, or the calling process must have write permission to the file or appropriate privileges.

The contents of a *utimbuf* structure are:

```
time_t actime
    The new access time (The time_t type gives the number of seconds
    since the epoch.)
time_t modtime
    The new modification time
```

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` as described on page 22.

Returned Value

If successful, *utime()* returns the value 0 and it updates the access and modification times of the file to those specified. If unsuccessful, *utime()* returns the value -1, it does not update file times, and it sets *errno* to one of the following:

- EACCES    The process does not have search permission on some component of the *pathname* prefix; or all of the following are true:
  - newtimes* is NULL.
  - The effective user ID of the process does not match the file's owner.
  - The process does not have write permission on the file.
  - The process does not have appropriate privileges.
- ELOOP    A loop exists in symbolic links. This error is issued if more than POSIX\_SYMLINK symbolic links (defined in the limits.h header file) are detected in the resolution of *pathname*.

## ENAMETOOLONG

*pathname* is longer than PATH\_MAX characters, or some component of *pathname* is longer than NAME\_MAX characters while \_POSIX\_NO\_TRUNC is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH\_MAX. The PATH\_MAX and NAME\_MAX values can be determined using pathconf().

**ENOENT** There is no file named *pathname*, or the *pathname* argument is an empty string.

**ENOTDIR** Some component of the *pathname* prefix is not a directory.

**EPERM** *newtimes* is not NULL, the effective user ID of the calling process does not match the owner of the file, and the calling process does not have appropriate privileges.

**EROFS** *pathname* is on a read-only file system.

## Example CBC3BU07

```
/* CBC3BU07 */
#define _POSIX_SOURCE
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>
#include <utime.h>

main() {
    int fd;
    char fn[]="utime.file";
    struct utimbuf ubuf;

    if ((fd = creat(fn, S_IWUSR)) < 0)
        perror("creat() error");
    else {
        close(fd);
        puts("before utime()");
        system("ls -l utime.file");
        ubuf.modtime = 0;
        time(&ubuf.actime);
        if (utime(fn, &ubuf) != 0)
            perror("utime() error");
        else {
            puts("after utime()");
            system("ls -l utime.file");
        }
        unlink(fn);
    }
}
```

## Output

```
before utime()
--w----- 1 WELLIE  SYS1          0 Apr 19 15:23 utime.file
after utime()
--w----- 1 WELLIE  SYS1          0 Dec 31 1969 utime.file
```

### Related Information

- “limits.h” on page 32
- “utime.h” on page 53



# utimes() — Set File Access and Modification Times

## Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

## Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/time.h>

int utimes(const char *path, const struct timeval times[2]);
```

## General Description

The `utimes()` function sets the access and modification times of the file pointed to by the *path* argument to the value of the *times* argument.

The *times* argument is an array of two `timeval` structures. The first array member represents the date and time of the last access, and the second member represents the date and time of the last modification. The times in the `timeval` structure are measured in seconds and microseconds since the Epoch, but the actual time stored with the file are rounded to the nearest second. The `timeval` members are:

tv\_sec        seconds since January 1, 1970  
tv\_usec       microseconds

If the *times* argument is a null pointer, the access and modification times of the file are set to the current time. The same process privilege requirements of the `utime()` function are required by `utimes()`. The last file status change, field `st_ctime` in a `stat()`, is updated with the current time.

To provide an ASCII input/output format for applications using this function, define feature test macro `__LIBASCII` described on page 22.

## Returned Value

If successful, 0 is returned.

If unsuccessful, `utimes()` returns -1, and returns the error value in `errno`.

- EACCES    The process does not have search permission on some component of the *path* prefix; or all of the following are true:
- *times* is NULL
  - The effective user ID of the process does not match the file's owner
  - The process does not have write permission on the file
  - The process does not have appropriate privileges
- ELOOP    A loop exists in symbolic links. This error is issued if more than **POSIX\_SYMLLOOP** symbolic links (defined in the `limits.h` header file) are detected in the resolution of *path*.

**ENAMETOOLONG**

*path* is longer than **PATH\_MAX** characters or some component of *path* is longer than **NAME\_MAX** characters while **\_POSIX\_NO\_TRUNC** is in effect. For symbolic links, the length of the pathname string substituted for a symbolic link exceeds **PATH\_MAX**. The **PATH\_MAX** and **NAME\_MAX** values can be determined using `pathconf()`.

**ENOTENT** There is no file named *path*, or the *path* argument is an empty string.

**ENOTDIR** Some component of the *path* prefix is not a directory.

**EPERM** *times* is not NULL, the effective user ID of the calling process does not match the owner of the file, and the calling process does not have appropriate privileges.

**EROFS** *path* is on a read-only file system.

**Related Information**

- “sys/time.h” on page 49
- “utime() — Set File Access and Modification Times” on page 1664

## \_\_utmpxname() — Change the utmpx Database Name

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <utmpx.h>
```

```
int __utmpxname(char *file);
```

### General Description

The `__utmpxname()` function changes the name of the utmpx database file for the current thread from default `/etc/utmpx` to the name specified by *file*. The `__utmpxname()` function does not open the file. It closes the old utmpx database file, if it is currently opened for the current thread, and saves the new utmpx database file name. If the file does not exist no indication is given.

Because the `__utmpxname()` function processes thread specific data the `__utmpxname()` function can be used safely from a multi-threaded application. If multiple threads in the same process open the database, then each thread opens the database with a different file descriptor. The thread's database file descriptor is closed when the calling thread terminates or the `endutxent()` function is called by the calling thread.

Programs must not reference the data passed back by `getutxline()`, `getutxid()`, `getutxent()`, or `pututxline()` after `__utmpxname()` has been called (the storage has been freed.) The `endutxent()` function resets the name of the utmpx data base back to the default value. If you must do additional utmpx operations on a non-standard utmpx data base after calling `endutxent()`, then call `__utmpxname()` again, to re-establish the non-standard name.

### Returned Value

If successful, `__utmpxname()` returns the value 0. If unsuccessful, it returns the value -1.

### Related Information

- “`getutxent()` — Read Next Entry in utmpx Database” on page 620
- “`getutxline()` — Search by Line utmpx Database” on page 624
- “`getutxid()` — Search by ID utmpx Database” on page 622
- “`pututxline()` — Write Entry to utmpx Database” on page 1061
- “`setutxent()` — Reset to Start of utmpx Database” on page 1282
- “`endutxent()` — Close the utmpx Database” on page 313

**va\_arg() - va\_end() - va\_start() — Access Function Arguments****Standards**

Standards / Extensions	C or C++	Dependencies
ISO C	both	

**Format**

```
#include <stdarg.h>
```

```
var_type va_arg(va_list arg_ptr, var_type);
void va_end(va_list arg_ptr);
void va_start(va_list arg_ptr, variable_name);
```

**\_XOPEN\_SOURCE**

```
#define _XOPEN_SOURCE
#include <varargs.h>
```

```
var_type va_arg(va_list arg_ptr, var_type);
void va_end(va_list arg_ptr);
void va_start(va_list arg_ptr);
```

**General Description**

The `va_arg()`, `va_end()`, and `va_start()` macros access the arguments to a function when it takes a fixed number of required arguments and a variable number of optional arguments. You declare required arguments as ordinary parameters to the function and access the arguments through the parameter names.

The `va_start()` macro initializes the `arg_ptr` pointer for subsequent calls to `va_arg()` and `va_end()`.

The argument `variable_name` is the identifier of the rightmost named parameter in the parameter list (preceding `,` ...). Use the `va_start()` macro before the `va_arg()` macro. Corresponding `va_start()` and `va_end()` macro calls must be in the same function. If `variable_name` is declared as a register, with a function or an array type, or with a type that is not compatible with the type that results after application of the default argument promotions, then the behavior is undefined.

The `va_arg()` macro retrieves a value of the given `var_type` from the location given by `arg_ptr` and increases `arg_ptr` to point to the next argument in the list. The `va_arg()` macro can retrieve arguments from the list any number of times within the function.

The macros also provide fixed-point decimal support under OS/390 C. The `sizeof(xx)` operator is used to determine the size and type casting that is used to generate the values. Therefore, a call, such as, `x = va_arg(ap, _Decimal(5,2));` is valid. The size of a fixed-point decimal number, however, cannot be made a variable. Therefore, a call, such as, `z = va_arg(ap, _Decimal(x,y))` where `x = 5` and `y = 2` is invalid.

The `va_end()` macro is needed by some systems to indicate the end of parameter scanning.

va\_start() and va\_arg() do not work with parameter lists of functions whose linkages were changed with the #pragma linkage directive.

stdarg.h and varargs.h are mutually exclusive. Whichever #include comes first, determines the form of macro that is visible.

The type definition for the va\_list type is normally "char \*va\_list[2]". Some applications (especially ported applications) require that the va\_list type be defined as "char \*va\_list". This alternate va\_list type is available if the user defines the feature test macro \_VARARG\_EXT\_ before the inclusion of any system header file. If the \_VARARG\_EXT\_ feature test macro is defined, va\_list will be typed as char \*va\_list, and the functions vprintf(), fprintf(), vsprintf(), and vswprintf() will use this alternate va\_list type.

## Returned Value

The va\_arg() macro returns the current argument. The va\_end() and va\_start() macros do not return a value.

## Example

### CBC3BV01

```
/* CBC3BV01
   This example passes a variable number of arguments to a function,
   stores each argument in an array, and prints each argument.
*/
#include <stdio.h>
#include <stdarg.h>

void vout(int max, ...);

int main(void)
{
    vout(3, "Sat", "Sun", "Mon");
    printf("\n");
    vout(5, "Mon", "Tues", "Wed", "Thurs", "Fri");
}

void vout(int max, ...)
{
    va_list arg_ptr;
    int args = 0;
    char *days[7];

    va_start(arg_ptr, max);
    while(args < max) {
        days[args] = va_arg(arg_ptr, char *);
        printf("Day: %s \n", days[args++]);
    }
    va_end(arg_ptr);
}
```

## Output

```
Day: Sat
Day: Sun
Day: Mon
```

```
Day: Mon
Day: Tues
Day: Wed
Day: Thurs
Day: Fri
```

```
/* This example uses a variable number of arguments for
   fixed-point decimal data types.
   The example works in OS/390 C only.
*/
#include <stdio.h>
#include <stdarg.h>
#include <decimal.h>

void vprnt(int, ...);

int main(void) {
    int i = 168;
    decimal(10,2) pd01 = 12345678.12d;
    decimal(20,5) pd02 = -987.65d;
    decimal(31,20) pd03 = 12345678901.12345678900987654321d;
    int j = 135;

    vprnt(0, i, pd01, pd02, pd03, j);

    return(0);
}

void vprnt(int whichcase, ...) {
    va_list arg_ptr;
    int m, n;
    decimal(10,2) va01;
    decimal(20,5) va02;
    decimal(31,20) va03;

    va_start(arg_ptr, whichcase);

    switch (whichcase) {
        case 0:
            m = va_arg(arg_ptr, int);
            va01 = va_arg(arg_ptr, decimal(10,2));
            va02 = va_arg(arg_ptr, decimal(20,5));
            va03 = va_arg(arg_ptr, decimal(31,20));
            n = va_arg(arg_ptr, int);
            printf("m      = %d\n", m);
            printf("va01 = %D(10,2)\n", va01);
            printf("va02 = %D(20,5)\n", va02);
            printf("va03 = %D(31,20)\n", va03);
            printf("n      = %d\n", n);
            break;
        default:
            printf("Illegal case number : %d\n", whichcase);
    }

    va_end(arg_ptr);
}
```

## Output

```

m      = 168
va01 = 12345678.12
va02 = -987.65000
va03 = 12345678901.12345678900987654321
n      = 135

```

### CBC3BV02

```

/* CBC3BV02
   These examples use the _XOPEN_SOURCE feature test macro.
   This example passes a variable number of arguments to a function,
   stores each argument in an array, and prints each argument.
*/
#define _XOPEN_SOURCE
#include <stdio.h>
#include <varargs.h>

void vout(va_alist)
va_dcl
{
    va_list arg_ptr;
    int args = 0;
    int max;
    char *days[7];

    va_start(arg_ptr);
    max = va_arg(arg_ptr, int);
    while(args < max) {
        days[args] = va_arg(arg_ptr, char *);
        printf("Day:      %s\n", days[args++]);
    }
    va_end(arg_ptr);
}

int main(void)
{
    vout(3, "Sat", "Sun", "Mon");
    printf("\n");
    vout(5, "Mon", "Tues", "Wed", "Thurs", "Fri");
}

/* This example uses a variable number of arguments for
   fixed-point decimal data types.
   The example works in OS/390 C only.
*/
#define _XOPEN_SOURCE
#include <stdio.h>
#include <varargs.h>
#include <decimal.h>

void vprnt(va_alist)
va_dcl
{
    va_list arg_ptr;
    int m, n, whichcase;
    decimal(10,2) va01;
    decimal(20,5) va02;
    decimal(31,20) va03;

    va_start(arg_ptr);

```

```
    whichcase = va_arg(arg_ptr, int);

    switch (whichcase) {
        case 0:
            m = va_arg(arg_ptr, int);
            va01 = va_arg(arg_ptr, decimal(10,2));
            va02 = va_arg(arg_ptr, decimal(20,5));
            va03 = va_arg(arg_ptr, decimal(31,20));
            n = va_arg(arg_ptr, int);
            printf("m      = %d\n", m);
            printf("va01 = %D(10,2)\n", va01);
            printf("va02 = %D(20,5)\n", va02);
            printf("va03 = %D(31,20)\n", va03);
            printf("n      = %d\n", n);
            break;
        default:
            printf("Illegal case number : %d\n", whichcase);
    }

    va_end(arg_ptr);
}

int main(void) {
    int i = 168;
    decimal(10,2) pd01 = 12345678.12d;
    decimal(20,5) pd02 = -987.65d;
    decimal(31,20) pd03 = 12345678901.12345678900987654321d;
    int j = 135;

    vprnt(0, i, pd01, pd02, pd03, j);

    return(0);
}
```

## Related Information

- “stdarg.h” on page 42
- “vfprintf() — Format and Print Data to Stream” on page 1679
- “vprintf() — Format and Print Data to stdout” on page 1681
- “vsprintf() — Format and Print Data to Buffer” on page 1683



## valloc() — Page-Aligned Memory Allocator

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>
```

```
void *valloc(size_t size);
```

### General Description

The `valloc()` function has the same effect as `malloc()`, except that the allocated memory will be aligned to a multiple of the value returned by `sysconf(_SC_PAGESIZE)`.

**Note:** When `free()` is used to release storage obtained by `valloc()`, the storage is not made available for reuse. The storage will not be freed until the enclave goes away.

### Special Behavior for C++

The C++ keywords `new` and `delete` are not interoperable with `valloc()`, `calloc()`, `free()`, `malloc()`, or `realloc()`.

### Returned Value

If successful, `valloc()` returns a pointer to the reserved storage. The storage space to which the returned value points is guaranteed to be aligned on a page boundary.

If unsuccessful, `valloc()` returns a `NULL` if there is not enough storage available, or if *size* is 0. If `valloc()` returns a `NULL` because there is not enough storage it will also return an error value in `errno`.

The following are the possible values of `errno`:

`ENOMEM`    Insufficient memory is available

### Related Information

- “`stdlib.h`” on page 45
- “`malloc()` — Reserve Storage Block” on page 786
- “`sysconf()` — Determine System Configuration Options” on page 1483

## vfork() — Create a New Process

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 4.3

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <unistd.h>
```

```
pid_t vfork(void);
```

**Note:** Although POSIX.1 does not require that the `<sys/types.h>` include file be included, XPG4 has it as an optional header. Therefore it is recommended that you include it for portability.

### General Description

The `vfork()` function creates a new process. The `vfork()` function has the same effect as `fork()`, except that the behavior is undefined, if the process created by `vfork()` attempts to call any other C/370 function before calling either `exec()` or `_exit()`. The new process (the *child process*) is an exact duplicate of the process that calls `vfork()` (the *parent process*), except for the following:

- The child process has a unique process ID (PID), which does not match any active process group ID.
- The child has a different parent process ID, that is, the process ID of the process that called `vfork()`.
- The child has its own copy of the parent's file descriptors. Each file descriptor in the child refers to the same open file description as the corresponding file descriptor in the parent.
- The child has its own copy of the parent's open directory streams. Each child's open directory stream may share directory stream positioning with the corresponding parent's directory stream.
- The following elements in the `tms` structure are set to 0 in the child:

```
tms_utime
tms_stime
tms_cutime
tms_cstime
```

For more information about these elements, see “`times()` — Get Process and Child Process Times” on page 1572.

- The child does not inherit any file locks previously set by the parent.
- The child process has no alarms set (similar to the results of a call to `alarm()` with an argument value of 0).
- The child has no pending signals.
- The child process may have its own copy of the parent's message catalogue descriptors.
- All `semadj` values are cleared.

- Interval timers are reset in the child process.

In all other respects, the child is identical to the parent. Because the child is a duplicate, it contains the same call to `vfork()` that was in the parent. Execution begins with this `vfork()` call, which returns a value of 0; the child then proceeds with normal execution.

The `vfork()` function is not supported from a multi-thread environment.

For more information on `vfork()` from an OS/390 perspective, refer to *OS/390 UNIX System Services Programming: Assembler Callable Services Reference*.

You can use OS/390 memory files from an OS/390 UNIX program. However, use of the `vfork()` function from the program removes access from a hiperspace memory file for the child process. Use of an `exec` function from the program clears a memory file when the process address space is cleared.

### Special Behavior for C

For POSIX resources, `vfork()` behaves as just described. But in general, MVS resources that existed in the parent do *not* exist in the child. This is true for open streams in MVS data sets and assembler-accessed OS/390 facilities, such as STIMERS. In addition, OS/390 allocations (through JCL, SVC99, or ALLOCATE) are not passed to the child process.

### Returned Value

If successful, `vfork()` returns 0 to the child process and the process ID of the newly created child to the parent process. If unsuccessful, `vfork()` fails to create a child process and returns -1 to the parent. `vfork()` then sets `errno` to one of the following:

EAGAIN	There are insufficient resources to create another process, or else the process has already reached the maximum number of processes you can run.
ELEMSGERR	LE message file not available.
ELENOFORK	Application contains a language that does not support <code>fork()</code> .
ENOMEM	The process requires more space than is available.

### Related Information

- “`sys/types.h`” on page 49
- “`unistd.h`” on page 53
- “`alarm()` — Set an Alarm” on page 102
- “`rexec()` — Execute Commands One at a Time on a Remote Host” on page 1143
- “`fcntl()` — Control Open File Descriptors” on page 350
- “`semop()` — Semaphore Operations” on page 1175
- “`signal()` — Handle Interrupts” on page 1330
- “`times()` — Get Process and Child Process Times” on page 1572
- “`getrlimit()` — Control Maximum Resource Consumption” on page 591
- “`nice()` — Change Priority of a Process” on page 864
- “`putenv()` — Change or Add an Environment Variable” on page 1054

- “setlocale() — Set Locale” on page 1241
- “shmat() — Shared Memory Attach Operation” on page 1285
- “putenv() — Change or Add an Environment Variable” on page 1054
- “sigaction() — Examine or Change a Signal Action” on page 1295
- “sigpending() — Examine Pending Signals” on page 1337
- “sigprocmask() — Examine or Change a Thread” on page 1339
- “stat() — Get File Information” on page 1404
- “system() — Execute a Command” on page 1488
- “times() — Get Process and Child Process Times” on page 1572
- “ulimit() — Get/Set Process File Size Limits” on page 1645

## vfprintf() — Format and Print Data to Stream

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4 XPG4.2	both	

### Format

```
#include <stdarg.h>
#include <stdio.h>
```

```
int vfprintf(FILE *stream, const char *format, va_list arg_ptr);
```

### General Description

The `vfprintf()` function is similar to `fprintf()`, except that *arg\_ptr* points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by `va_start()` for each call. In contrast, `fprintf()` can have a list of arguments, but the number of arguments in that list is fixed when you compile the program. For a specification of the *format* string, see “`fprintf()` - `printf()` - `sprintf()` — Format and Write Data” on page 436.

`vfprintf()` is not supported for files opened with `type=record`.

`vfprintf()` has the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

### Returned Value

If there is no error, `vfprintf()` returns the number of characters written to *stream*. If an error occurs, the function returns a negative value.

**Note:** In contrast to some UNIX-based implementations of the C language, the OS/390 C/C++ implementation of the `vfprintf()` family increments the pointer to the variable arguments list. To control whether the pointer to the argument is incremented, call the `va_end` macro after each call to `vsprintf()`.

### Example CBC3BV03

```
/* CBC3BV03
   This example prints out a variable number of strings to the file
   myfile.dat, using vfprintf().
*/
#include <stdarg.h>
#include <stdio.h>

void vout(FILE *stream, char *fmt, ...);
char fmt1 [] = "%s %s %s\n";

int main(void)
{
    FILE *stream;
    stream = fopen("myfile.dat", "w");
```

```
    vout(stream, fmt1, "Sat", "Sun", "Mon");
}

void vout(FILE *stream, char *fmt, ...)
{
    va_list arg_ptr;

    va_start(arg_ptr, fmt);
    vfprintf(stream, fmt, arg_ptr);
    va_end(arg_ptr);
}
```

### Output

Sat Sun Mon

### Related Information

- “stdarg.h” on page 42
- “stdio.h” on page 43
- “va\_arg() - va\_end() - va\_start() — Access Function Arguments” on page 1670
- “vprintf() — Format and Print Data to stdout” on page 1681
- “vsprintf() — Format and Print Data to Buffer” on page 1683

# vprintf() — Format and Print Data to stdout

## Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4.2	both	

## Format

```
#include <stdarg.h>
#include <stdio.h>

int vprintf(const char *format, va_list arg_ptr);
```

## General Description

The vprintf() function is similar to printf(), except that *arg\_ptr* points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by va\_start() for each call. In contrast, printf() can have a list of arguments, but the number of arguments in that list is fixed when you compile the program. For a specification of the *format* string, see “fprintf() - printf() - sprintf() — Format and Write Data” on page 436.

vprintf() is not supported for files opened with type=record.

vprintf() has the same restriction as any write operation for a read immediately following a write or a write immediately following a read. Between a write and a subsequent read, there must be an intervening flush or reposition. Between a read and a subsequent write, there must also be an intervening flush or reposition unless an EOF has been reached.

## Returned Value

If there is no error, vprintf() returns the number of characters written to stdout. If an error occurs, vprintf() returns a negative value.

**Note:** In contrast to some UNIX-based implementations of the C language, the OS/390 C/C++ implementation of the vprintf() family increments the pointer to the variable arguments list. To control whether the pointer to the argument is incremented, call the va\_end macro after each call to vprintf().

## Example CBC3BV04

```
/* CBC3BV04
   This example prints out a variable number of strings to stdout, using
   vprintf().
*/
#include <stdarg.h>
#include <stdio.h>

void vout(char *fmt, ...);
char fmt1 [] = "%s %s %s %s %s \n";

int main(void)
{
    FILE *stream;
    stream = fopen("myfile.dat", "w");
```

```
    vout(fmt1, "Mon", "Tues", "Wed", "Thurs", "Fri");
}

void vout(char *fmt, ...)
{
    va_list arg_ptr;

    va_start(arg_ptr, fmt);
    vprintf(fmt, arg_ptr);
    va_end(arg_ptr);
}
```

### Output

```
Mon  Tues  Wed   Thurs  Fri
```

### Related Information

- “stdarg.h” on page 42
- “stdio.h” on page 43
- “va\_arg() - va\_end() - va\_start() — Access Function Arguments” on page 1670
- “vfprintf() — Format and Print Data to Stream” on page 1679
- “vsprintf() — Format and Print Data to Buffer” on page 1683



## vsprintf() — Format and Print Data to Buffer

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C XPG4.2	both	

### Format

```
#include <stdarg.h>
#include <stdio.h>
```

```
int vsprintf(char *target-string, const char *format, va_list arg_ptr);
```

### General Description

The `vsprintf()` function is similar to `sprintf()`, except that *arg\_ptr* points to a list of arguments whose number can vary from call to call in the program. In contrast, `sprintf()` can have a list of arguments, but the number of arguments in that list is fixed when you compile the program. For a specification of the *format* string, see “`fprintf()` - `printf()` - `sprintf()` — Format and Write Data” on page 436.

### Returned Value

If there is no error, `vsprintf()` returns the number of characters written *target-string*. If an error occurs, `vsprintf()` returns a negative value.

**Note:** In contrast to some UNIX-based implementations of the C language, the OS/390 C/C++ implementation of the `vprintf()` family increments the pointer to the variable arguments list. To control whether the pointer to the argument is incremented, call the `va_end` macro after each call to `vsprintf()`.

### Example CBC3BV05

```
/* CBC3BV05
   This example assigns a variable number of strings to string and prints
   the resultant string, using vsprintf().
*/
#include <stdarg.h>
#include <stdio.h>

void vout(char *string, char *fmt, ...);
char fmt1 [] = "%s %s %s\n";

int main(void)
{
    char string[100];

    vout(string, fmt1, "Sat", "Sun", "Mon");
    printf("The string is: %s\n", string);
}

void vout(char *string, char *fmt, ...)
{
    va_list arg_ptr;

    va_start(arg_ptr, fmt);
    vsprintf(string, fmt, arg_ptr);
```

```
    va_end(arg_ptr);  
}
```

### Output

The string is: Sat Sun Mon

### Related Information

- “stdarg.h” on page 42
- “stdio.h” on page 43
- “va\_arg() - va\_end() - va\_start() — Access Function Arguments” on page 1670
- “vfprintf() — Format and Print Data to Stream” on page 1679
- “vprintf() — Format and Print Data to stdout” on page 1681

## vswprintf() — Write Wide Characters

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

### Format

#### Non-XP4

```
#include <stdarg.h>
#include <wchar.h>
```

```
int vswprintf(wchar_t *wcs, size_t n, const wchar_t *format, va_list arg);
```

#### XP4

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <stdarg.h>
#include <wchar.h>
```

```
int vswprintf(wchar_t *wcs, size_t n, const wchar_t *format, va_list arg);
```

### General Description

The `vswprintf()` function is equivalent to `vsprintf()`, except for the following:

- The argument *wcs* specifies an array of type `wchar_t`, rather than an array of type `char`, into which the generated output is to be written.
- The argument *format* specifies an array of type `wchar_t`, rather than an array of type `char`, which describes how subsequent arguments are converted for output.
- `%c` without an `l` prefix means an integer argument is to be converted to `wchar_t`, as if by calling `mbtowc()`, and then written.
- `%c` with `l` prefix means a `wint_t` is converted to `wchar_t` and then written.
- `%s` without an `l` prefix means a character array containing a multibyte character sequence is to be converted to an array of `wchar_t` and written. The conversion will take place as if `mbrtowc()` were called repeatedly.
- `%s` with `l` prefix means an array of `wchar_t` will be written. The array is written up to but not including the terminating null character, unless the precision specifies a shorter output.

A null wide character is written at the end of the wide characters written; the null wide character is not counted as part of the returned sum. If copying takes place between objects that overlap, the behavior is undefined.

### Special Behavior for XP4

If you define any feature test macro specifying XP4 behavior before the statement in your program source file to include the `wchar` header, then you must also define the `_MSE_PROTOS` feature test macro to make the declaration of the `vswprintf()` function in the `wchar` header available when you compile your program. Please see Table 3 on page 18 for a list of XP4 and other feature test macros.

## Returned Value

Returns the number of wide characters written in the array, not counting the terminating null wide character.

**Note:** In contrast to some UNIX-based implementations of the C language, the OS/390 C/C++ implementation of the `vprintf()` family increments the pointer to the variable arguments list. To control whether the pointer to the argument is incremented, call the `va_end` macro after each call to `vswprintf()`.

## Example CBC3BV06

```
/* CBC3BV06 */
#include <stdio.h>
#include <stdarg.h>
#include <wchar.h>

void vout(wchar_t *, size_t, wchar_t *, ...);

wchar_t *format3 = L"%ls %d %ls";
wchar_t *format5 = L"%ls %d %ls %d %ls";

int main(void)
{
    wchar_t wcsr[100];

    vout(wcsr, 100, format3, L"ONE", 2L, L"THREE");
    printf("%ls\n", wcsr);
    vout(wcsr, 100, format5, L"ONE", 2L, L"THREE", 4L, L"FIVE");
    printf("%ls\n", wcsr);
}

void vout(wchar_t *wcs, size_t n, wchar_t *fmt, ...)
{
    va_list arg_ptr;

    va_start(arg_ptr, fmt);
    vswprintf(wcs, n, fmt, arg_ptr);
    va_end(arg_ptr);
}
```

## Related Information

- “`stdarg.h`” on page 42
- “`wchar.h`” on page 54
- “`swprintf()` — Write Wide Characters to a Wide-Character Array” on page 1475
- “`vswprintf()` — Format and Print Data to Buffer” on page 1683

## wait() — Wait for a Child Process to End

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <sys/wait.h>
```

```
pid_t wait(int *status_ptr);
```

### General Description

Suspends the calling process until any one of its child processes ends. More precisely, `wait()` suspends the calling process until the system obtains status information on the ended child. If the system already has status information on a completed child process when `wait()` is called, `wait()` returns immediately. `wait()` is also ended if the calling process receives a signal whose action is either to execute a signal handler or to end the process.

The argument `status_ptr` points to a location where `wait()` can store a status value. This status value is zero if the child process explicitly returns a zero status. If it is not zero, it can be analyzed with the status analysis macros, described in “Status Analysis Macros,” below.

The `status_ptr` pointer may also be `NULL`, in which case `wait()` ignores the child's return status.

The following function calls are equivalent:

```
wait(status_ptr);
waitpid(-1,status_ptr,0);
wait3(status_ptr,0,NULL);
```

For more information, see “`waitpid()` — Wait for a Specific Child Process to End” on page 1692.

### Special Behavior for XPG4.2

If the calling process has `SA_NOCLDWAIT` set or has `SIGCHLD` set to `SIG_IGN`, and the process has no unwaited for children that were transformed into zombie processes, it will block until all of the children terminate, and `wait()` will fail and set `errno` to `ECHILD`.

### Status Analysis Macros

If the `status_ptr` argument is not `NULL`, `wait()` places the child's return status in `*status_ptr`. You can analyze this return status with the following macros, defined in the `sys/wait.h` header file:

**WIFEXITED(*\*status\_ptr*)**

This macro evaluates to a nonzero (true) value if the child process ended normally, that is, if it returned from `main()` or called one of the `exit()` or `_exit()` functions.

**WEXITSTATUS(*\*status\_ptr*)**

When `WIFEXITED()` is nonzero, `WEXITSTATUS()` evaluates to the low-order 8 bits of the child's return status passed on the `exit()` or `_exit()` function.

**WIFSIGNALED(*\*status\_ptr*)**

This macro evaluates to a nonzero (true) value if the child process ended because of a signal that was not caught.

**WIFSTOPPED(*\*status\_ptr*)**

This macro evaluates to a nonzero (true) value if the child process is currently stopped. This should only be used after a `waitpid()` with the `WUNTRACED` option.

**WSTOPSIG(*\*status\_ptr*)**

When `WIFSTOPPED()` is nonzero, `WSTOPSIG()` evaluates to the number of the signal that stopped the child.

**WTERMSIG(*\*status\_ptr*)**

When `WIFSIGNALED()` is nonzero, `WTERMSIG()` evaluates to the number of the signal that ended the child process.

**Returned Value**

If successful, `wait()` returns a value that is the process ID (PID) of the child whose status information has been obtained. If unsuccessful, it returns the value `-1` and sets `errno` to one of the following:

- ECHILD** The caller has no appropriate child processes, that is, it has no child processes whose status has not been obtained by previous calls to `wait()`, `waitid()`, `waitpid()`, or `wait3()`. `ECHILD` is also returned when the `SA_NOCLDWAIT` flag is set.
- EINTR** `wait()` was interrupted by a signal. The value of *\*status\_ptr* is undefined.

**Example  
CBC3BW01**

```
/* CBC3BW01
   This example suspends the calling process until any child processes ends.
*/
#define _POSIX_SOURCE
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

main() {
    pid_t pid;
    time_t t;
    int status;

    if ((pid = fork()) < 0)
        perror("fork() error");
    else if (pid == 0) {
```

```

    time(&t);
    printf("child (pid %d) started at %s", (int) getpid(), ctime(&t));
    sleep(5);
    time(&t);
    printf("child exiting at %s", ctime(&t));
    exit(42);
}
else {
    printf("parent has forked child with pid of %d\n", (int) pid);
    time(&t);
    printf("parent is starting wait at %s", ctime(&t));
    if ((pid = wait(&status)) == -1)
        perror("wait() error");
    else {
        time(&t);
        printf("parent is done waiting at %s", ctime(&t));
        printf("the pid of the process that ended was %d\n", (int) pid);
        if (WIFEXITED(status))
            printf("child exited with status of %d\n", WEXITSTATUS(status));
        else if (WIFSIGNALED(status))
            printf("child was terminated by signal %d\n",
                WTERMSIG(status));
        else if (WIFSTOPPED(status))
            printf("child was stopped by signal %d\n", WSTOPSIG(status));
        else puts("reason unknown for child termination");
    }
}
}

```

### Output

```

parent has forked child with pid of 65546
parent is starting wait at Fri Jun 16 10:53:03 1995
child (pid 65546) started at Fri Jun 16 10:53:04 1995
child exiting at Fri Jun 16 10:53:09 1995
parent is done waiting at Fri Jun 16 10:53:10 1995
the pid of the process that ended was 65546
child exited with status of 42

```

### Related Information

- “signal.h” on page 41
- “sys/types.h” on page 49
- “sys/wait.h” on page 51
- “exit() — End Program” on page 330
- “\_exit() — End a Process and Bypass the Cleanup” on page 332
- “fork() — Create a New Process” on page 422
- “pause() — Suspend a Process Pending a Signal” on page 899
- “waitpid() — Wait for a Specific Child Process to End” on page 1692
- “waitid() — Wait for Child Process to Change State” on page 1690
- “wait3() — Wait for Child Process to Change State” on page 1696

waitid() — Wait for Child Process to Change State

Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/wait.h>
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

General Description

The waitid() function suspends the calling process until one of its children changes state. It records the current state of a child in the structure pointed to by *infop*. If a child process changed state prior to the call, waitid() returns immediately.

The *idtype* and *id* arguments are used to specify which children waitid() will wait for.

| If *idtype* is P\_PID, waitid() will wait for the child with a process ID equal to (pid\_t)*id*.

| If *idtype* is P\_GID, waitid() will wait for any child with a process group ID equal to (pid\_t)*id*.

| If *idtype* is P\_ALL, waitid() will wait for any children and *id* is ignored.

The *options* argument is used to specify which state changes to wait for. It is formed by OR-ing together one or more of the following flags:

- WEXITED   Wait for processes that have exited.
- WSTOPPED   Status will be returned for any child that has stopped upon receipt of a signal.
- WCONTINUED   Status will be returned for any child that has stopped and has been continued.
- WNOHANG   Return immediately if there are no children to wait for.
- WNOWAIT   Keep the process whose status is returned in *infop* in a waitable state. This will not affect the state of the process; the process may be waited for again after this call completes.

The *infop* argument must point to a siginfo\_t structure. If waitid() returns because a child process was found that specified the conditions indicated by the arguments *idtype* and *options* then the structure pointed to by *infop* will be filled in by the system with the status of the process. The si\_signo member will always be equal to SIGCHLD.



## Returned Value

If waitid() returns due to the change of state of one of its children, 0 is returned. Otherwise, -1 is returned and errno is set to indicate the error.

The waitid() function will fail if:

- |        |   |
|--------|---|
| ECHILD | The calling process has no existing unwaited-for child processes.   |
| EINTR  | The waitid() function was interrupted due to the receipt of a signal by the calling process.                            |
| EINVAL | An invalid value was specified for <i>options</i> , or <i>idtype</i> and <i>id</i> specify an invalid set of processes. |

## Related Information

- “exec Functions” on page 322
- “exit() — End Program” on page 330
- “wait() — Wait for a Child Process to End” on page 1687

## waitpid() — Wait for a Specific Child Process to End

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <sys/wait.h>
```

```
pid_t waitpid(pid_t pid, int *status_ptr, int options);
```

### General Description

Suspends the calling process until a child process ends or is stopped. More precisely, waitpid() suspends the calling process until the system gets status information on the child. If the system already has status information on an appropriate child when waitpid() is called, waitpid() returns immediately. waitpid() is also ended if the calling process receives a signal whose action is either to execute a signal handler or to end the process.

**pid\_t *pid*** Specifies the child processes the caller wants to wait for:

- If *pid* is greater than zero, waitpid() waits for termination of the specific child whose process ID is equal to *pid*.
- If *pid* is equal to zero, waitpid() waits for termination of any child whose process group ID is equal to that of the caller.
- If *pid* is -1, waitpid() waits for any child process to end.
- If *pid* is less than -1, waitpid() waits for the termination of any child whose process group ID is equal to the absolute value of *pid*.

**int \**status\_ptr***

Points to a location where waitpid() can store a status value. This status value is zero if the child process explicitly returns a zero status. Otherwise, it is a value that can be analyzed with the status analysis macros described in “Status Analysis Macros”, below.

The *status\_ptr* pointer may also be NULL, in which case waitpid() ignores the child's return status.

**int *options***

Specifies additional information for waitpid(). The *options* value is constructed from the bitwise inclusive OR of zero or more of the following flags defined in the sys/wait.h header file:

WCONTINUED

**Special Behavior for XPG4.2:** Reports the status of any continued child processes as well as terminated ones. The WIFCONTINUED macro lets a process distinguish between a continued process and a terminated one.

WNOHANG Demands status information immediately. If status information is immediately available on an appropriate child process, waitpid() returns this information. Otherwise,

waitpid() returns immediately with an error code indicating that the information was not available. In other words, WNOHANG checks child processes without causing the caller to be suspended.

#### WUNTRACED

Reports on stopped child processes as well as terminated ones. The WIFSTOPPED macro lets a process distinguish between a stopped process and a terminated one.

### Special Behavior for XPG4.2

If the calling process has SA\_NOCLDWAIT set or has SIGCHLD set to SIG\_IGN, and the process has no unwaited for children that were transformed into zombie processes, it will block until all of the children terminate, and waitpid() will fail and set errno to ECHILD.

### Status Analysis Macros

If the *status\_ptr* argument is not NULL, waitpid() places the child's return status in *\*status\_ptr*. You can analyze this return status with the following macros, defined in the sys/wait.h header file:

#### WIFEXITED(*\*status\_ptr*)

This macro evaluates to a nonzero (true) value if the child process ended normally (that is, if it returned from main(), or else called the exit() or \_exit() function).

#### WEXITSTATUS(*\*status\_ptr*)

When WIFEXITED() is nonzero, WEXITSTATUS() evaluates to the low-order 8 bits of the status argument that the child passed to the exit() or \_exit() function, or the value the child process returned from main().

#### WIFSIGNALED(*\*status\_ptr*)

This macro evaluates to a nonzero (true) value if the child process ended because of a signal that was not caught.

#### WTERMSIG(*\*status\_ptr*)

When WIFSIGNALED() is nonzero, WTERMSIG() evaluates to the number of the signal that ended the child process.

#### WIFSTOPPED(*\*status\_ptr*)

This macro evaluates to a nonzero (true) value if the child process is currently stopped. This should only be used after a waitpid() with the WUNTRACED option.

#### WSTOPSIG(*\*status\_ptr*)

When WIFSTOPPED() is nonzero, WSTOPSIG() evaluates to the number of the signal that stopped the child.

#### WIFCONTINUED(*\*status\_ptr*)

**Special Behavior for XPG4.2:** This macro evaluates to a nonzero (true) value if the child process has continued from a job control stop. This should only be used after a waitpid() with the WCONTINUED option.

## Returned Value

If successful, `waitpid()` returns a value of the process (usually a child) whose status information has been obtained.

If `WNOHANG` was given, and if there is at least one process (usually a child) whose status information is not available, `waitpid()` returns zero.

If unsuccessful, `waitpid()` returns the value `-1` and sets `errno` to one of the following:

<code>ECHILD</code>	The process specified by <i>pid</i> does not exist or is not a child of the calling process, or the process group specified by <i>pid</i> does not exist or does not have any member process that is a child of the calling process.
<code>EINTR</code>	<code>waitpid()</code> was interrupted by a signal. The value of <i>*status_ptr</i> is undefined.
<code>EINVAL</code>	The value of <i>options</i> is incorrect.

## Example

### CBC3BW02

```

/* CBC3BW02
   The following function suspends the calling process using waitpid()
   until a child process ends.
*/
#define _POSIX_SOURCE
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

main() {
    pid_t pid;
    time_t t;
    int status;

    if ((pid = fork()) < 0)
        perror("fork() error");
    else if (pid == 0) {
        sleep(5);
        exit(1);
    }
    else do {
        if ((pid = waitpid(pid, &status, WNOHANG)) == -1)
            perror("wait() error");
        else if (pid == 0) {
            time(&t);
            printf("child is still running at %s", ctime(&t));
            sleep(1);
        }
        else {
            if (WIFEXITED(status))
                printf("child exited with status of %d\n", WEXITSTATUS(status));
            else puts("child did not exit successfully");
        }
    } while (pid == 0);
}

```

## Output

```
child is still running at Fri Jun 16 11:05:43 1995
child is still running at Fri Jun 16 11:05:44 1995
child is still running at Fri Jun 16 11:05:45 1995
child is still running at Fri Jun 16 11:05:46 1995
child is still running at Fri Jun 16 11:05:47 1995
child is still running at Fri Jun 16 11:05:48 1995
child is still running at Fri Jun 16 11:05:49 1995
child exited with status of 1
```

### Related Information

- “signal.h” on page 41
- “sys/types.h” on page 49
- “sys/wait.h” on page 51
- “exit() — End Program” on page 330
- “\_exit() — End a Process and Bypass the Cleanup” on page 332
- “fork() — Create a New Process” on page 422
- “pause() — Suspend a Process Pending a Signal” on page 899
- “wait() — Wait for a Child Process to End” on page 1687
- “waitid() — Wait for Child Process to Change State” on page 1690
- “wait3() — Wait for Child Process to Change State” on page 1696

## wait3() — Wait for Child Process to Change State

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.2.2

### Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/wait.h>
pid_t wait3 (int *stat_loc, int options, struct rusage *resource_usage);
```

### General Description

The wait3() function allows the calling process to obtain status information for specified child processes.

The following call:

```
wait3(stat_loc, options, resource_usage)
```

is equivalent to the call:

```
waitpid((pid_t)-1, stat_loc, options);
```

except that on successful completion, if the *resource\_usage* argument to wait3() is not a null pointer, the rusage structure that the third argument points to is filled in for the child process identified by the return value.

### Returned Value

See waitpid().

In addition to the error conditions specified on waitpid(), under the following conditions, wait3() may fail and set errno to:

**ECHILD**      The calling process has no existing unwaited-for child processes, or if the set of processes specified by the argument *pid* can never be in the states specified by the argument *options*.

### Related Information

- “exec Functions” on page 322
- “exit() — End Program” on page 330
- “fork() — Create a New Process” on page 422
- “pause() — Suspend a Process Pending a Signal” on page 899

## wrtomb() — Convert a Wide Character to a Multibyte Character

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

### Format

#### Non-XP4

```
#include <wchar.h>
```

```
int wrtomb(char *s, wchar_t wchar, mbstate_t *pss);
```

#### XP4

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>
```

```
int wrtomb(char *s, wchar_t wchar, mbstate_t *pss);
```

### General Description

If *s* is a null pointer, the `wrtomb()` function determines the number of bytes necessary to enter the initial shift state (zero if encodings are not state-dependent or if the initial conversion state is described). The resulting state described is the initial conversion state.

If *s* is not a null pointer, the `wrtomb()` function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by *wchar* (including any shift sequences), and stores the resulting bytes in the array whose first element is pointed to by *s*. At most, `MB_CUR_MAX` bytes are stored. If *wchar* is a null wide character, the resulting state described is the initial conversion state.

`wrtomb()` is a “restartable” version of `wctomb()`. That is, shift state information is passed as one of the arguments and is updated on return. With `wrtomb()`, you can switch from one multibyte string to another, provided that you have kept the shift-state information for each multibyte string.

The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you change the category, undefined results can occur.

### Special Behavior for XP4

If you define any feature test macro specifying XP4 behavior before the statement in your program source file to include the `wchar` header, then you must also define the `_MSE_PROTOS` feature test macro to make the declaration of the `wrtomb()` function in the `wchar` header available when you compile your program. Please see Table 3 on page 18 for a list of XP4 and other feature test macros.

## Returned Value

If *s* is a null pointer, the `wctomb()` function returns the number of bytes needed to enter the initial shift state. The value returned will not be greater than that of `MB_CUR_MAX`.

If *s* is not a null pointer, the `wctomb()` function returns the number of bytes stored in the array object (including any shift sequences) when *wchar* is a valid wide character. Otherwise, when *wchar* is not a valid wide character, an encoding error occurs, the value of the macro `EILSEQ` is stored in `errno` and the value `-1` is returned, but the conversion state remains unchanged.

## Example

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    char    *string;
    wchar_t wc;
    int     length;

    length = wctomb(string, wc, NULL);
}
```

## Related Information

- “`wchar.h`” on page 54
- “`mblen()` — Calculate Length of Multibyte Character” on page 791
- “`mbrlen()` — Calculate Length of Multibyte Character” on page 794
- “`mbrtowc()` — Convert a Multibyte Character to a Wide Character” on page 797
- “`mbsrtowcs()` — Convert a Multibyte String to a Wide-Character String” on page 802
- “`wcsrtombs()` — Convert Wide-Character String to Multibyte String” on page 1726
- “`wctomb()` — Convert Wide Character to Multibyte Character” on page 1751



## wcscat() — Append to Wide-Character String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h>  /* or #include <wctype.h> */
```

```
wchar_t *wcscat(wchar_t *string1, const wchar_t *string2);
```

### General Description

Appends a copy of the string pointed to by *string2* to the end of the string pointed to by *string1*.

The `wcscat()` function operates on null-terminated wide-character strings. The string arguments to this function must contain a wide null character marking the end of the string. Bounds checking is not performed.

The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you change the category, undefined results can occur.

This function was formerly declared in `wcstr.h`, which continues to be maintained for compatibility.

### Returned Value

Returns the value of *string1*.

### Example

#### CBC3BW04

```
/* CBC3BW04
   This example creates the wide character string "computer program" using
   wcscat().
*/
#include <stdio.h>
#include <wchar.h>

#define SIZE 40

int main(void)
{
    wchar_t buffer1[SIZE] = L"computer";
    wchar_t * string      = L" program";
    wchar_t * ptr;

    ptr = wcscat( buffer1, string );
    printf( "buffer1 = %ls\n", buffer1 );
}
```

### Output

```
buffer1 = computer program
```

**Related Information**

- “wchar.h” on page 54
- “strcat() — Concatenate Strings” on page 1414
- “wcschr() — Search for Wide-Character Substring” on page 1701
- “wcscmp() — Compare Wide-Character Strings” on page 1703
- “wcscpy() — Copy Wide-Character String” on page 1707
- “wcscspn() — Find Offset of First Wide-Character Match” on page 1709
- “wcslen() — Calculate Length of Wide-Character String” on page 1715
- “wcsncat() — Append to Wide-Character String” on page 1716

## wcschr() — Search for Wide-Character Substring

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment SAA XPG4 XPG4.2	both	

### Format

```
#include <wchar.h> /* or #include <wctype.h> */
```

```
wchar_t *wcschr(const wchar_t *string1, wchar_t character);
```

### Compiler Option

LANGVLV(EXTENDED), LANGVLV(SAA), or LANGVLV(SAA2)

### General Description

Searches *string* for the occurrence of *character*. The *character* may be a wide null character (`\0`). The wide null character at the end of *string* is included in the search. The `wcschr()` function operates on null-terminated wide-character strings. The argument to this function must contain a wide null character marking the end of the string.

The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you change the category, undefined results can occur.

This function was formerly declared in `wcstr.h`, which continues to be maintained for compatibility.

### Returned Value

Returns a pointer to the first occurrence of *character* in *string*. If the character is not found, a `NULL` pointer is returned.

### Example CBC3BW05

```
/* CBC3BW05
   This example finds the first occurrence of the character p in the wide
   character string "computer program" using wcschr().
*/
#include <stdio.h>
#include <wchar.h>

#define SIZE 40

int main(void)
{
    wchar_t buffer1[SIZE] = L"computer program";
    wchar_t * ptr;
    wint_t ch = L'p';

    ptr = wcschr( buffer1, ch );
    printf( "The first occurrence of %lc in '%ls' is '%ls'\n",

```

```
        ch, buffer1, ptr );  
    }
```

**Output**

The first occurrence of p in 'computer program' is 'puter program'

**Related Information**

- “wchar.h” on page 54
- “strchr() — Search for Character” on page 1416
- “wscat() — Append to Wide-Character String” on page 1699
- “wcscmp() — Compare Wide-Character Strings” on page 1703
- “wcsncpy() — Copy Wide-Character String” on page 1707
- “wcscspn() — Find Offset of First Wide-Character Match” on page 1709
- “wcslen() — Calculate Length of Wide-Character String” on page 1715
- “wcsncmp() — Compare Wide-Character Strings” on page 1718

## wcscmp() — Compare Wide-Character Strings

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h>  /* or #include <wctype.h>  */
```

```
int wcscmp(const wchar_t *string1, const wchar_t *string2);
```

### General Description

Compares two wide-character strings. The `wcscmp()` function operates on null-terminated wide-character strings. The string arguments to this function must contain a wide null character marking the end of the string.

The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you change the category, undefined results can occur.

This function was formerly declared in `wcstr.h`, which continues to be maintained for compatibility.

### Returned Value

Returns a value indicating the relationship between the two strings, as follows:

Value	Meaning
Less than 0	string pointed to by <i>string1</i> less than string pointed to by <i>string2</i>
0	string pointed to by <i>string1</i> identical to string pointed to by <i>string2</i>
Greater than 0	string pointed to by <i>string1</i> greater than string pointed to by <i>string2</i>

### Example

#### CBC3BW06

```
/* CBC3BW06
   This example compares the wide character string string1 to string2
   using wcscmp().
*/
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    int result;
    wchar_t string1[] = L"abcdef";
    wchar_t string2[] = L"abcdefg";

    result = wcscmp( string1, string2 );

    if ( result == 0 )
        printf( "\\\"%ls\\\" is identical to \\\"%ls\\\"\\n", string1, string2);
    else if ( result < 0 )
        printf( "\\\"%ls\\\" is less than \\\"%ls\\\"\\n", string1, string2 );
}
```

```
    else
        printf( "\"%ls\" is greater than \"%ls\"\\n", string1, string2);
}
```

**Output**

"abcdef" is less than "abcdefg"

**Related Information**

- “wchar.h” on page 54
- “strcmp() — Compare Strings” on page 1418
- “wscat() — Append to Wide-Character String” on page 1699
- “wchr() — Search for Wide-Character Substring” on page 1701
- “wscpy() — Copy Wide-Character String” on page 1707
- “wscspn() — Find Offset of First Wide-Character Match” on page 1709
- “wcslen() — Calculate Length of Wide-Character String” on page 1715
- “wcsncmp() — Compare Wide-Character Strings” on page 1718

## wscoll() — Language Collation String Comparison

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h>
```

```
int wscoll(const wchar_t *wcs1, const wchar_t *wcs2);
```

### General Description

Compares the wide-character string pointed to by `wcs1` to the wide-character string pointed to by `wcs2`, both interpreted as appropriate to the `LC_COLLATE` category of the current locale.

### Returned Value

Returns an integer greater than, equal to, or less than zero, according to whether the wide string pointed to by `wcs1` is greater than, equal to, or less than the wide-character string pointed to by `wcs2`, when both wide-character strings are interpreted as appropriate to the `LC_COLLATE` category of the current locale.

The `wscoll()` function differs from the `wscmp()` function. `wscoll()` function performs a comparison between two wide character strings based on language collation rules as controlled by the `LC_COLLATE` category. On the other hand, the `wscmp()` functions performs a wide-character code to wide-character code comparison.

The `wscoll()` function indicates error conditions by setting `errno`; however, there is no returned value to indicate an error. To check for errors, `errno` should be set to zero, and then checked upon return from `wscoll()`. If `errno` is nonzero, an error has occurred.

The `EILSEQ` error can be set to indicate that the `wcs1` or `wcs2` arguments contain characters outside the domain of the collating sequence.

**Note:** The ISO/C Multibyte Support Extensions do not indicate that the `wscoll()` function may return with an error.

### Example

#### CBC3BW07

```
/* CBC3BW07 */
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    int result;
    wchar_t *wcs1 = L"first_wide_string";
    wchar_t *wcs2 = L"second_wide_string";

    result = wscoll(wcs1, wcs2);
```

```
if ( result == 0)
    printf("\%ls\" is identical to \"%ls\"\\n", wcs1, wcs2);
else if ( result < 0)
    printf("\%ls\" is less than \"%ls\"\\n", wcs1, wcs2);
else
    printf("\%ls\" is greater than \"%ls\"\\n", wcs1, wcs2);
}
```

**Output**

"first\_wide\_string" is less than "second\_wide\_string"

**Related Information**

- “wchar.h” on page 54
- “strcoll() — Compare Strings” on page 1420
- “setlocale() — Set Locale” on page 1241



## wcscpy() — Copy Wide-Character String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h> /* or #include <wctype.h> */
```

```
wchar_t *wcscpy(wchar_t *string1, const wchar_t *string2);
```

### General Description

Copies the contents of *string2* (including the ending wide null character) into *string1*. The `wcscpy()` function operates on null-terminated wide-character strings. The string arguments to this function must contain a wide null character marking the end of the string. Bounds checking is not performed.

The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you change the category, undefined results can occur.

This function was formerly declared in `wcstr.h`, which continues to be maintained for compatibility.

### Returned Value

Returns the value of *string1*.

### Example CBC3BW08

```
/* CBC3BW08
   This example copies the contents of source to destination using wcscpy().
*/
#include <stdio.h>
#include <wchar.h>

#define SIZE    40

int main(void)
{
    wchar_t source[ SIZE ] = L"This is the source string";
    wchar_t destination[ SIZE ] = L"And this is the destination string";
    wchar_t * return_string;

    printf( "destination is originally = \"%ls\"\n", destination );
    return_string = wcscpy( destination, source );
    printf( "After wcscpy, destination becomes \"%ls\"\n", destination );
}
```

### Output

```
destination is originally = "And this is the destination string"
After wcscpy, destination becomes "This is the source string"
```

## **Related Information**

- “wchar.h” on page 54
- “strcpy() — Copy String” on page 1422
- “wscat() — Append to Wide-Character String” on page 1699
- “wcschr() — Search for Wide-Character Substring” on page 1701
- “wcscmp() — Compare Wide-Character Strings” on page 1703
- “wcscspn() — Find Offset of First Wide-Character Match” on page 1709
- “wcslen() — Calculate Length of Wide-Character String” on page 1715
- “wcsncpy() — Copy Wide-Character String” on page 1720

## wcscspn() — Find Offset of First Wide-Character Match

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h> /* or #include <wctype.h> */
```

```
size_t wcscspn(const wchar_t *string1, const wchar_t *string2);
```

### General Description

Determines the number of wide characters in the initial segment of the string pointed to by *string1* that do not appear in the string pointed to by *string2*. The `wcscspn()` function operates on null-terminated wide-character strings. The string arguments to these functions must contain a null wide character marking the end of the string.

The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you change the category, undefined results can occur.

This function was formerly declared in `wcstr.h`, which continues to be maintained for compatibility.

### Returned Value

Returns the number of wide characters in the segment.

### Example CBC3BW09

```
/* CBC3BW09
   This example uses wcscspn() to find the first occurrence of any of the
   characters a, x, l, or e in string.
*/
#include <stdio.h>
#include <wchar.h>

#define SIZE 40

int main(void)
{
    wchar_t string[ SIZE ] = L"This is the source string";
    wchar_t * substring = L"axle";

    printf( "The first %i characters in the string \"%ls\" are not in the "
           "string \"%ls\" \n", wcscspn( string, substring),
           string, substring );
}
```

### Output

The first 10 characters in the string "This is the source string" are not in the string "axle"

### **Related Information**

- “wchar.h” on page 54
- “strcspn() — Compare Strings” on page 1424
- “wscat() — Append to Wide-Character String” on page 1699
- “wcschr() — Search for Wide-Character Substring” on page 1701
- “wcscmp() — Compare Wide-Character Strings” on page 1703
- “wcscpy() — Copy Wide-Character String” on page 1707
- “wcslen() — Calculate Length of Wide-Character String” on page 1715

## wcsftime() — Format Date and Time

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

#### Non-XPG4

```
#include <wchar.h>
```

```
size_t wcsftime(wchar_t *wcs, size_t maxsize, const wchar_t *format,
                const struct tm *time_ptr)
```

#### XPG4

```
#define _XOPEN_SOURCE
#include <wchar.h>
```

```
size_t wcsftime(wchar_t *wcs, size_t maxsize, const char *format,
                const struct tm *time_ptr)
```

#### XPG4 and MSE

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>
```

```
size_t wcsftime(wchar_t *wcs, size_t maxsize, const wchar_t *format,
                const struct tm *time_ptr)
```

### General Description

Format date and time into a wide character string. The `wcsftime()` function is equivalent to the `strftime()` function, except that:

- The argument `wcs` specifies an array of a wide string into which the generated output is to be placed.
- The argument `maxsize` indicates a number of wide characters.
- The argument `*format` specifies an array of wide characters comprising the format string.
- The returned value indicates a number of wide characters.

### Special Behavior for XPG4

If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the `wchar` header, then the compiler assumes that your program is using the XPG4 variety of the `wcsftime()` function unless you also define the `_MSE_PROTOS` feature test macro. Please see Table 3 on page 18 for a list of XPG4 and other feature test macros.

The prototype for the XPG4 variety of the `wcsftime()` function is:

```
size_t wcsftime(wchar_t *wcs, size_t maxsize, const char *format,
               const struct tm *time_ptr)
```

The difference between this variety and the MSE variety of the `wcsftime()` function is that the third argument, `*format`, specifies an array of characters rather than an array of wide characters comprising the format string.

### Returned Value

If the total number of resulting wide characters including the terminating null wide character is not more than `maxsize`, the `wcsftime()` function returns the number of wide characters placed into the array pointed to by `wcs` not including the terminating null wide character. Otherwise, zero is returned and the contents of the array are indeterminate.

### Example CBC3BW10

```
/* CBC3BW10 */
#include <stdio.h>
#include <time.h>
#include <wchar.h>

int main(void)
{
    struct tm *timeptr;
    wchar_t dest[100];
    time_t temp;
    size_t rc;

    temp = time(NULL);
    timeptr = localtime(&temp);
    rc = wcsftime(dest, sizeof(dest)-1, L" Today is %A,"
                  L" %b %d.\n Time: %I:%M %p", timeptr);
    printf("%d characters placed in string to make:\n\n%S", rc, dest);
}
```

### Output

42 characters placed in string to make:

```
Today is Friday, Jun 16.
Time: 01:48 pm
```

### Related Information

- “`wchar.h`” on page 54
- “`strftime()` — Convert to Formatted Time” on page 1432

## wcsid() — Character Set ID for Wide Character

### Standards

Standards / Extensions	C or C++	Dependencies
C/370	both	

### Format

```
#include <stdlib.h>
```

```
int wcsid(const wchar_t c)
```

### External Entry Point

```
@@WCSID, __wcsid
```

### General Description

Determines the character set identifier for the specified wide character.

To avoid infringing on the user's name-space, this non-standard function has two names. One name is prefixed with two underscore characters, and one name is not. The name without the prefix underscore characters is exposed only when you use LANGLVL(EXTENDED).

To use this function, you must either invoke the function using its external entry point name (i.e., the name that begins with two underscore characters), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

### Returned Value

Returns the character set identifier for the wide character, or the value `-1` if the wide character is not valid.

### Example

#### CBC3BW11

```
/* CBC3BW11
   This example checks character set ID for wide character.
*/
#include "locale.h"
#include "stdio.h"
#include "stdlib.h"

main() {
    wchar_t wc = L'A';
    int rc;

    rc = wcsid(wc);
    printf("wide character '%C' is in character set id %i\n", wc, rc);
}
```

### **Related Information**

- “stdlib.h” on page 45



## wcslen() — Calculate Length of Wide-Character String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h> /* or #include <wctype.h> */
```

```
size_t wcslen(const wchar_t *string);
```

### General Description

Computes the number of wide characters in the string pointed to by *string*.

The behavior of this wide-character function is affected by the LC\_CTYPE category of the current locale. If you change the category, undefined results can occur.

### Returned Value

Returns the number of wide characters that precede the terminating wide null character.

### Example CBC3BW12

```
/* CBC3BW12
   This example computes the length of a wide-character string, using wcslen().
*/
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t * string = L"abcdef";

    printf( "Length of \"%ls\" is %i\n", string, wcslen( string ));
}
```

### Output

Length of "abcdef" is 6

### Related Information

- “wchar.h” on page 54
- “mblen() — Calculate Length of Multibyte Character” on page 791
- “strlen() — Determine String Length” on page 1436
- “wcsncat() — Append to Wide-Character String” on page 1716
- “wcsncmp() — Compare Wide-Character Strings” on page 1718
- “wcsncpy() — Copy Wide-Character String” on page 1720

## wcsncat() — Append to Wide-Character String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h> /* or #include <wctype.h> */
```

```
wchar_t *wcsncat(wchar_t *string1, const wchar_t *string2, size_t count);
```

### General Description

Appends up to *count* wide characters from *string2* to the end of *string1* and appends a null wide character to the result. The `wcsncat()` function operates on null-terminated wide-character strings. The string arguments to this function must contain a null wide character marking the end of the string.

The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you change the category, undefined results can occur.

This function was formerly declared in `wcstr.h`, which continues to be maintained for compatibility.

### Returned Value

Returns *string1*.

### Example

#### CBC3BW13

```
/* CBC3BW13
   This example demonstrates the difference between wcscat() and wcsncat().
   wcscat() appends the entire second string to the first whereas
   wcsncat() appends only the specified number of characters in the second
   string to the first.
*/
#include <stdio.h>
#include <wchar.h>
#include <string.h>

#define SIZE 40

int main(void)
{
    wchar_t buffer1[SIZE] = L"computer";
    wchar_t * ptr;

    /* Call wcscat with buffer1 and " program" */

    ptr = wcscat( buffer1, L" program" );
    printf( "wcscat : buffer1 = \"%ls\\n\"", buffer1 );

    /* Reset buffer1 to contain just the string "computer" again */

    memset( buffer1, L'\0', sizeof( buffer1 ) );
    ptr = wcscpy( buffer1, L"computer" );
```

```
/* Call wcsncat with buffer1 and " program" */  
ptr = wcsncat( buffer1, L" program", 3 );  
printf( "wcsncat: buffer1 = \"%ls\\n\"", buffer1 );  
}
```

### Output

```
wcscat : buffer1 = "computer program"  
wcsncat: buffer1 = "computer pr"
```

### Related Information

- “wchar.h” on page 54
- “strncat() — Concatenate Strings” on page 1438
- “wcsncmp() — Compare Wide-Character Strings” on page 1718
- “wcsncpy() — Copy Wide-Character String” on page 1720

## wcsncmp() — Compare Wide-Character Strings

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h> /* or #include <wctype.h> */
```

```
int wcsncmp(const wchar_t *string1, const wchar_t *string2, size_t count);
```

### General Description

Compares up to *count* wide characters in *string1* to *string2*. The `wcsncmp()` function operates on null-terminated wide-character strings. The string arguments to this function must contain a null wide character marking the end of the string.

The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you change the category, undefined results can occur.

This function was formerly declared in `wcstr.h`, which continues to be maintained for compatibility.

### Returned Value

Returns a value indicating the relationship between the two strings, as follows:

Value	Meaning
Less than 0	string pointed to by <i>string1</i> less than the string pointed to by <i>string2</i>
0	string pointed to by <i>string1</i> identical to string pointed to by <i>string2</i>
Greater than 0	string pointed to by <i>string1</i> greater than string pointed to by <i>string2</i>

### Example

#### CBC3BW14

```
/* CBC3BW14
   This example demonstrates the difference between wcscmp() and wcsncmp().
*/
#include <stdio.h>
#include <wchar.h>

#define SIZE 10

int main(void)
{
    int result;
    int index = 3;
    wchar_t buffer1[SIZE] = L"abcdefg";
    wchar_t buffer2[SIZE] = L"abcfg";
    void print_result( int, wchar_t *, wchar_t * );

    result = wcscmp( buffer1, buffer2 );
    printf( "Comparison of each character\n" );
    printf( " wcscmp: " );
```

```

print_result( result, buffer1, buffer2 );

result = wcsncmp( buffer1, buffer2, index);
printf( "\nComparison of only the first %i characters\n", index );
printf( "  wcsncmp: " );
print_result( result, buffer1, buffer2 );
}

void print_result( int res, wchar_t * p_buffer1, wchar_t * p_buffer2 )
{
    if ( res == 0 )
        printf( "\"%ls\" is identical to \"%ls\"\n", p_buffer1, p_buffer2);
    else if ( res < 0 )
        printf( "\"%ls\" is less than \"%ls\"\n", p_buffer1, p_buffer2 );
    else
        printf( "\"%ls\" is greater than \"%ls\"\n", p_buffer1, p_buffer2 );
}

```

### Output

Comparison of each character

wcsncmp: "abcdefg" is less than "abcfg"

Comparison of only the first 3 characters

wcsncmp: "abcdefg" is identical to "abcfg"

### Related Information

- “wchar.h” on page 54
- “strncmp() — Compare Strings” on page 1440
- “wcsncmp() — Compare Wide-Character Strings” on page 1703
- “wcsncat() — Append to Wide-Character String” on page 1716
- “wcsncpy() — Copy Wide-Character String” on page 1720

## wcsncpy() — Copy Wide-Character String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h> /* or #include <wctype.h> */
```

```
wchar_t *wcsncpy(wchar_t *string1, const wchar_t *string2, size_t count);
```

### General Description

Copies up to *count* wide characters from *string2* to *string1*. If *string2* is shorter than *count* characters, *string1* is padded out to *count* characters with null wide characters. The `wcsncpy()` function operates on null-terminated wide-character strings. The string arguments to this function must contain a null wide character marking the end of the string.

The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you change the category, undefined results can occur.

This function was formerly declared in `wcstr.h`, which continues to be maintained for compatibility.

### Returned Value

Returns *string1*.

### Example CBC3BW15

```
/* CBC3BW15
   This example demonstrates the difference between wcscopy() and wcsncpy().
*/
#include <stdio.h>
#include <wchar.h>

#define SIZE    40

int main(void)
{
    wchar_t source[ SIZE ] = L"123456789";
    wchar_t source1[ SIZE ] = L"123456789";
    wchar_t destination[ SIZE ] = L"abcdefg";
    wchar_t destination1[ SIZE ] = L"abcdefg";
    wchar_t * return_string;
    int index = 5;

    /* This is how wcscopy works */
    printf( "destination is originally = '%ls'\n", destination );
    return_string = wcscopy( destination, source );
    printf( "After wcscopy, destination becomes '%ls'\n\n", destination );

    /* This is how wcsncpy works */
    printf( "destination1 is originally = '%ls'\n", destination1 );
    return_string = wcsncpy( destination1, source1, index );
```

```
    printf( "After wcsncpy, destination1 becomes '%ls'\n", destination1 );  
}
```

**Output**

destination is originally = 'abcdefg'  
After wcsncpy, destination becomes '123456789'

destination1 is originally = 'abcdefg'  
After wcsncpy, destination1 becomes '12345fg'

**Related Information**

- “wchar.h” on page 54
- “strncpy() — Copy String” on page 1442
- “wcscpy() — Copy Wide-Character String” on page 1707
- “wcsncat() — Append to Wide-Character String” on page 1716
- “wcsncmp() — Compare Wide-Character Strings” on page 1718

## wcpbrk() — Locate First Wide Characters in String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h> /* or #include <wctr.h> */
```

```
wchar_t *wcpbrk(const wchar_t *string1, const wchar_t *string2);
```

### General Description

Locates the first occurrence in the string pointed to by *string1* of any character from the string pointed to by *string2*.

The behavior of this wide-character function is affected by the LC\_CTYPE category of the current locale. If you change the category, undefined results can occur.

This function was formerly declared in wctr.h, which continues to be maintained for compatibility.

### Returned Value

Returns a pointer to the character, or NULL if no wchar\_t from *string2* occurs in *string1*.

### Example

#### CBC3BW16

```
/* CBC3BW16
   This example returns a pointer to the first occurrence in the array
   string of either a or b, using wcpbrk().
*/
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t * result;
    wchar_t * string = L"The Blue Danube";
    wchar_t *chars = L"ab";

    result = wcpbrk( string, chars);
    printf("The first occurrence of any of the characters \"%ls\\
          \" in \" \"%ls\\\" is \"%ls\\\"\\n\",
          chars, string, result);
}
```

### Output

The first occurrence of any of the characters "ab" in "The Blue Danube" is "anube"



**Related Information**

- “wchar.h” on page 54
- “strpbrk() — Find Characters in String” on page 1444
- “wcschr() — Search for Wide-Character Substring” on page 1701
- “wcscmp() — Compare Wide-Character Strings” on page 1703
- “wcscspn() — Find Offset of First Wide-Character Match” on page 1709
- “wcsncmp() — Compare Wide-Character Strings” on page 1718
- “wcsrchr() — Locate Last Wide Character in String” on page 1724
- “wcswcs() — Locate Wide-Character Substring in Wide Character String” on page 1744

## wcsrchr() — Locate Last Wide Character in String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h>  /* or #include <wctype.h>  */
```

```
wchar_t *wcsrchr(const wchar_t *string, wchar_t character);
```

### General Description

Locates the last occurrence of *character* in the string pointed to by *string*. The terminating null wide character is considered to be part of the string.

The behavior of this wide character function is affected by the LC\_CTYPE category of the current locale. If you change the category, undefined results can occur.

This function was formerly declared in `wcstr.h`, which continues to be maintained for compatibility.

### Returned Value

Returns a pointer to the character, or a NULL pointer if *character* does not occur in the string.

### Example

#### CBC3BW17

```
/* CBC3BW17
   This example compares the use of wcschr() and wcsrchr(). It searches
   the string for the first and last occurrence of p in the wide character string.
*/
#include <stdio.h>
#include <wchar.h>

#define SIZE 40

int main(void)
{
    wchar_t buf[SIZE] = L"computer program";
    wchar_t * ptr;
    int ch = 'p';

    /* This illustrates wcschr */
    ptr = wcschr( buf, ch );
    printf( "The first occurrence of %c in '%ls' is '%ls'\n", ch, buf, ptr );

    /* This illustrates wcsrchr */
    ptr = wcsrchr( buf, ch );
    printf( "The last occurrence of %c in '%ls' is '%ls'\n", ch, buf, ptr );
}
```

### Output

The first occurrence of p in 'computer program' is 'puter program'  
The last occurrence of p in 'computer program' is 'program'

### Related Information

- “wchar.h” on page 54
- “strrchr() — Find Last Occurrence of Character in String” on page 1449
- “wcscmp() — Compare Wide-Character Strings” on page 1703
- “wcsncmp() — Find Offset of First Wide-Character Match” on page 1709
- “wcsncmp() — Compare Wide-Character Strings” on page 1718
- “wcschr() — Locate First Wide Characters in String” on page 1722
- “wcs wcs() — Locate Wide-Character Substring in Wide Character String” on page 1744

## wcsrtombs() — Convert Wide-Character String to Multibyte String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

### Format

#### Non-XP4

```
#include <wchar.h>
```

```
size_t wcsrtombs(char *dst, const wchar_t **src, size_t len, mbstate_t *ps);
```

#### XP4

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>
```

```
size_t wcsrtombs(char *dst, const wchar_t **src, size_t len, mbstate_t *ps);
```

### General Description

Converts a sequence of wide characters from the array indirectly pointed to by *src* into a sequence of corresponding multibyte characters that begin in the shift state described by *ps*, which, if *dst* is not a NULL pointer, are then stored into the array pointed to by *dst*. Conversion continues up to and including the terminating null wide character; the terminating null wide character (byte) shall be stored.

Conversion shall stop earlier in two cases:

- When a code is reached that does not correspond to a valid multibyte character.
- If *dst* is not a null pointer, conversion stops when the next multibyte element would exceed the limit of *len* total bytes to be stored into the array pointed to by *dst*.

Each conversion takes places as if by a call to the `wcrtomb()` function.

If *dst* is not null a pointer, the object pointed to by *src* shall be assigned either a NULL pointer (if conversion stopped due to reaching a terminating null wide character) or the address of the code just past the last wide character converted. If conversion stopped due to reaching a terminating null wide character, the resulting state described shall be the initial conversion state.

`wcsrtombs()` is a “restartable” version of `wcstombs()`. That is, shift state information is passed as on of the arguments, and gets updated on exit. With `wcsrtombs()`, you may switch from one multibyte string to another, provided that you have kept the shift state information.

The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you change the category, undefined results can occur.

### Special Behavior for XP4

If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the `wchar` header, then you must also define the `_MSE_PROTOS` feature test macro to make the declaration of the `wcsrtombs()` function in the `wchar` header available when you compile your program. Please see Table 3 on page 18 for a list of XPG4 and other feature test macros.

## Returned Value

If the string contains an invalid wide character, an encoding error occurs; The `wcsrtombs()` function stores the value of the macro `EILSEQ` in `errno` and returns `(size_t)-1`, but the conversion state shall be unchanged. Otherwise, it returns the number of bytes in the resulting multibyte character sequence, which is the same as the number of array elements modified when *dst* is not a null pointer.

## Example CBC3BW18

```
/* CBC3BW18 */
#include <stdio.h>
#include <string.h>
#include <wchar.h>

#define SIZE 20

int main(void)
{
    char    dest[SIZE];
    wchar_t *wcs = L"string";
    const wchar_t *ptr;
    size_t  count = SIZE;
    size_t  length;

    ptr = (wchar_t *) wcs;
    length = wcsrtombs(dest, &ptr, count, NULL);
    printf("%d characters were converted.\n", length);
    printf("The converted string is \"%s\"\n\n", dest);

    /* Reset the destination buffer */
    memset(dest, '\\0', sizeof(dest));

    /* Now convert only 3 characters */
    ptr = (wchar_t *) wcs;
    length = wcsrtombs(dest, &ptr, 3, NULL);
    printf("%d characters were converted.\n", length);
    printf("The converted string is \"%s\"\n\n", dest);
}
```

## Output

```
6 characters were converted.
The converted string is "string"
```

```
3 characters were converted.
The converted string is "str"
```

## Related Information

- “wchar.h” on page 54
- “mblen() — Calculate Length of Multibyte Character” on page 791
- “mbrlen() — Calculate Length of Multibyte Character” on page 794
- “mbrtowc() — Convert a Multibyte Character to a Wide Character” on page 797
- “mbsrtowcs() — Convert a Multibyte String to a Wide-Character String” on page 802
- “wctomb() — Convert a Wide Character to a Multibyte Character” on page 1697
- “wcstombs() — Convert Wide-Character String to Multibyte Character String” on page 1740

## wcssp() — Search for Wide Characters in a String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h> /* or #include <wctr.h> */
```

```
size_t wcssp(const wchar_t *string1, const wchar_t *string2);
```

### General Description

Computes the number of wide characters in the initial segment of the string pointed to by *string1*, which consists entirely of wide characters from the string pointed to by *string2*.

The behavior of this wide-character function is affected by the LC\_CTYPE category of the current locale. If you change the category, undefined results can occur.

This function was formerly declared in `wctr.h`, which continues to be maintained for compatibility.

### Returned Value

Returns the number of wide characters in the segment.

### Example

#### CBC3BW19

```
/* CBC3BW19
   This example finds the first occurrence in the array string of a
   character that is neither an a, b, nor c. Because the string in this
   example is cabbage, wcssp() returns 5, the index of the segment of
   cabbage before a character that is not an a, b, or c.
*/
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t * string = L"cabbage";
    wchar_t * source = L"abc";
    int index;

    index = wcssp( string, L"abc" );
    printf( "The first %d characters of \"%ls\" are found in \"%ls\"\\n",
           index, string, source );
}
```

### Output

The first 5 characters of "cabbage" are found in "abc"

### **Related Information**

- “wchar.h” on page 54
- “strspn() — Search String” on page 1451
- “wscat() — Append to Wide-Character String” on page 1699
- “wcschr() — Search for Wide-Character Substring” on page 1701
- “wcscmp() — Compare Wide-Character Strings” on page 1703
- “wcscspn() — Find Offset of First Wide-Character Match” on page 1709
- “wcsncmp() — Compare Wide-Character Strings” on page 1718



# wcsstr() — Locate a Wide Character Sequence

## Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

## Format

### Non-XP4

```
#include <wchar.h>

wchar_t *wcsstr(const wchar_t *wcs1, const wchar_t *wcs2);
```

### XP4

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

wchar_t *wcsstr(const wchar_t *wcs1, const wchar_t *wcs2);
```

## General Description

Locates the first occurrence in the wide-character string pointed to by `wcs1` of the sequence of wide characters (excluding the terminating null character) in the wide-character string pointed to by `wcs2`.

The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you change the category, undefined results can occur.

## Special Behavior for XP4

If you define any feature test macro specifying XP4 behavior before the statement in your program source file to include the `wchar` header, then you must also define the `_MSE_PROTOS` feature test macro to make the declaration of the `wcsstr()` function in the `wchar` header available when you compile your program. Please see Table 3 on page 18 for a list of XP4 and other feature test macros.

## Returned Value

Returns a pointer to the located wide string, or a null pointer if the wide-character string is not found. If `wcs2` points to a wide-character string with zero length, the function returns `wcs1`.

## Example

### CBC3BW20

```
/* CBC3BW20 */
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *wcs1 = L"needle in a haystack";
    wchar_t *wcs2 = L"hay";
    wchar_t *result;

    result = wcsstr(wcs1, wcs2);
```

```
    /* result = a pointer to "hatstack" */  
    printf("result: %S \n", result);  
}
```

**Related Information**

- “wchar.h” on page 54
- “strstr() — Locate Substring” on page 1453

## wcstod() — Convert Wide-Character String to a Double Floating-Point

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h>
```

```
double wcstod(const wchar_t *nptr, wchar_t **endptr);
```

### General Description

The `wcstod()` function converts a `wchar_t *` type floating-point number input string to a double value. The double value is hexadecimal floating-point or IEEE floating-point format depending on the floating-point mode of the thread invoking `wcstod()`. The `wcstod()` function uses `__isBFP()` to determine the floating-point format (hexadecimal floating-point or IEEE floating-point) of the invoking thread.

See the “*fscanf* Family of Formatted Input Functions” on page 467 for a description of special infinity and NaN sequences recognized by OS/390 formatted input functions, including `wcstod()` in IEEE floating-point mode.

Converts the initial portion of the wide-character string pointed to by *nptr* to double representation. First it decomposes the input string into three parts:

1. An initial, possibly empty, sequence of white space characters (as specified by the `iswspace()` function)
2. A subject sequence resembling a floating-point constant.
3. A final string of one or more unrecognized characters, including the terminating NULL character of the input string.

Then it attempts to convert the subject sequence to a floating-point number, and returns the result.

The expected form of the subject sequence is an optional plus or minus sign, then a non-empty sequence of digits optionally containing a decimal-point wide character, then an optional exponent part as defined in ISO/IEC 9899: subclause 6.1.3.1, but with no floating suffix. The subject sequence is defined as the longest initial subsequence of the input wide-character string, starting with the first non-white space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide-character string is empty or consists entirely of white space wide characters, or if the first non-white space wide character is other than a sign, a digit, or a decimal-point wide character.

If the subject sequence has the expected form, the sequence of wide characters starting with the first digit or the decimal-point wide character (whichever occurs first) is interpreted as a floating constant according to the rules of ISO/IEC 9899: subclause 6.1.3.1, except the decimal-point wide character is used in place of a period, and if neither an exponent part nor a decimal-point wide character appears, a decimal point is assumed to follow the last digit in the wide-character string. If the subject sequence begins with a minus sign, the value resulting from the conversion

is negated. A pointer to the final wide-character string is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

In a locale other than the C locale, additional implementation-defined subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

The behavior of this wide-character function is affected by the LC\_CTYPE category of the current locale. If you change the category, undefined results can occur.

## Returned Value

Returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, plus or minus HUGE\_VAL is returned—according to the sign of the value—and the value of the macro ERANGE is stored in *errno*. If the correct value would cause underflow, zero is returned and the value of the macro ERANGE is stored in *errno*.

## Example

### CBC3BW21

```
/* CBC3BW21 */
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *wcs;
    wchar_t *stopwcs;
    double d;

    wcs = L"3.1415926This stopped it";
    d = wcstod(wcs, &stopwcs);
    printf("wcs = %ls \n", wcs);
    printf("    wcstod = %f\n", d);
    printf("    Stopped scan at %ls \n", stopwcs);
}
```

## Related Information

- “wchar.h” on page 54
- “\_\_isBFP() — Determine Application Floating-Point Format” on page 705

## wcstok() — Break a Wide-Character String into Tokens

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

#### Non-XPG4

```
#include <wchar.h>
```

```
wchar_t *wcstok(wchar_t *wcs1, const wchar_t *wcs2, wchar_t **ptr);
```

#### XPG4

```
#define _XOPEN_SOURCE
#include <wchar.h>
```

```
wchar_t *wcstok(wchar_t *wcs1, const wchar_t *wcs2);
```

#### XPG4 and MSE

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>
```

```
wchar_t *wcstok(wchar_t *wcs1, const wchar_t *wcs2, wchar_t **ptr);
```

### General Description

A sequence of calls to the `wcstok()` function breaks the wide string pointed to by `wcs1` into a sequence of tokens, each of which is delimited by a wide character from the wide string pointed to by `wcs2`. The third argument points to a caller-provided wide-character pointer into which the `wcstok()` function stores information necessary for it to continue scanning the same string.

The first call in the sequence, `wcs1` shall point to a wide-character string, while in subsequent calls for the same wide string, `wcs1` shall be a null pointer. If `wcs1` is a null pointer, the value pointed to by *ptr* shall match that set by the previous call for the same wide-character string; otherwise its value is ignored. The separator wide-character string pointed to by `wcs2` may be different from call to call.

The first call in the sequence, searches the wide-character string pointed to by `wcs1` for the first wide character that is not contained in the current separator wide-character string pointed to by `wcs2`. If no such wide character is found, then there are no tokens in the wide-character string pointed to by `wcs1` and the `wcstok()` function returns a null pointer. If such a wide character is found, it is the start of the first token.

The `wcstok()` function then searches from there for a wide character that is contained in the current separator wide string. If no such wide character is found, the current token extends to the end of the wide-character string pointed to by `wcs1`, and subsequent searches for a token will return a null pointer. If such a wide char-

acter is found, it is overwritten by a null character, which terminates the current token.

In all cases, the `wcstok()` function stores sufficient information in the pointer *ptr* so that subsequent calls, with a null pointer as the value of the first argument and the unmodified pointer value as the third, will start searching just past the end of the previously returned token (if any).

The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you change the category, undefined results can occur.

### Special Behavior for XPG4

If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the `wchar` header, then the compiler assumes that your program is using the XPG4 variety of the `wcstok()` function, unless you also define the `_MSE_PROTOS` feature test macro. Please see Table 3 on page 18 for a list of XPG4 and other feature test macros.

The prototype for the XPG4 variety of the `wcstok()` function is:

```
wchar_t *wcstok(wchar_t *wcs1, const wchar_t *wcs2);
```

This variety of the `wcstok()` function is missing a third parameter to specify the address of restart information in your program storage. Instead, C/370 provides comparable restart information in runtime library storage. Please note that this library storage is provided on a per thread basis making the XPG4 `wcstok()` function thread-specific for a threaded application.

### Returned Value

Returns a pointer to the first wide character of a token, or a NULL pointer if there is no token.

### Example CBC3BW22

```
/* CBC3BW22 */
#include <wchar.h>
int main(void)
{
    static wchar_t str1[] = L"?a??b,,#c";
    static wchar_t str2[] = L"\t\t";
    wchar_t *t, *ptr1, *ptr2;

    t = wcstok(str1, L"?", &ptr1);    /* t points to the token L"a" */
    t = wcstok(NULL, L",", &ptr1);    /* t points to the token L"??b" */
    t = wcstok(str2, L"\t", &ptr2);    /* t is a null pointer */
    t = wcstok(NULL, L"#", &ptr1);    /* t points to the token L"c" */
    t = wcstok(NULL, L"?", &ptr1);    /* t is a null pointer */
}
```

**Related Information**

- “wchar.h” on page 54
- “strtok() — Tokenize String” on page 1458

## wcstol() — Convert a Wide-Character String to a Long Integer

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h>
long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);
```

### General Description

Converts the initial portion of the wide-character string pointed to by *nptr* to long int representation. First it decomposes the input wide-character string into three parts:

1. An initial, possibly empty, sequence of white-space wide characters (as specified by the `iswspace()` function).
2. A subject sequence resembling an integer determined by the value of `base`.
3. A final wide-character string of one or more unrecognized wide characters, including the terminating NULL character of the input wide-character string.

Then it attempts to convert the subject sequence to an integer, and returns the result.

If the value of `base` is zero, the expected form of the subject sequence is that of an integer constant as described in ISO/IEC 9899: subclause 6.1.3.2, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of `base` is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits from the portable character set representing an integer with the radix specified by `base`, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from `a` (or `A`) through `z` (or `Z`) are ascribed the values 10 to 35; only letters whose ascribed values are less than that of `base` are permitted. If the value of `base` is 16, the characters `0x` or `0X` may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input wide-character string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide-character string is empty or consists entirely of white space, or if the first non-white-space wide character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of `base` is zero, the sequence of wide characters starting with the first digit is interpreted as an integer constant according to the rules of ISO/IEC 9899: subclause 6.1.3.2. If the subject sequence has the expected form and the value of `base` is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final wide-character string is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

In a locale other than the C locale, additional implementation defined subject sequence forms may be accepted.



If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

The behavior of this wide-character function is affected by the LC\_CTYPE category of the current locale. If you change the category, undefined results can occur.

## Returned Value

Returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, LONG\_MAX or LONG\_MIN is returned (according to the sign of the value), and the value of the macro ERANGE is stored in *errno*.

## Example

### CBC3BW23

```
/* CBC3BW23 */
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *wcs;
    wchar_t *stopwcs;
    long    l;
    int     base;

    wcs = L"10110134932";
    printf("wcs = %ls \n", wcs);
    for (base=2; base<=8; base*=2) {
        l = wcstol(wcs, &stopwcs, base);
        printf("    wcstol = %ld\n", l);
        printf("    Stopped scan at %ls \n\n", stopwcs);
    }
}
```

## Related Information

- “wchar.h” on page 54
- “strtol() — Convert Character String to Long” on page 1460

## wcstombs() — Convert Wide-Character String to Multibyte Character String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
size_t wcstombs(char *dest, const wchar_t *string, size_t count);
```

### General Description

Converts the wide-character string pointed to by *string* into the multibyte array pointed to by *dest*. The converted string begins in the initial shift state. The conversion stops after *count* bytes in *dest* are filled up or a null wide character is encountered.

The behavior of this wide-character function is affected by the LC\_CTYPE category of the current locale. If you change the category, undefined results can occur.

### Returned Value

Returns the length in bytes of the multibyte character string, not including a terminating null wide character. The value  $(\text{size\_t})-1$  is returned if an invalid multibyte character is encountered.

If *count* is the returned value the array is not null terminated.

If *\*dest* is a null pointer, the number of characters required to convert the wide character string is returned.

If the area pointed to by *\*dest* is too small (as indicated by the value of *count*) to contain the wide character codes represented as multibyte characters, the number of bytes containing complete multibyte characters is returned.

**Note:** The `wcstombs()` does not generate redundant shift characters between the DBCS characters. When the `wctomb()` function is called for each character, redundant shift characters are generated.

### Example CBC3BW24

```
/* CBC3BW24
   In this example, a wide-character string is converted to a char string twice.
   The first call converts the entire string, while the second call
   only converts three characters. The results are printed each time.
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define SIZE 20
```

```

int main(void)
{
    char dest[SIZE];
    wchar_t * dptr = L"string";
    size_t count = SIZE;
    size_t length;

    length = wcstombs( dest, dptr, count );
    printf( "%d characters were converted.\n", length );
    printf( "The converted string is \"%s\"\n\n", dest );

    /* Reset the destination buffer */
    memset( dest, '\0', sizeof(dest));

    /* Now convert only 3 characters */
    length = wcstombs( dest, dptr, 3 );
    printf( "%d characters were converted.\n", length );
    printf( "The converted string is \"%s\"\n", dest );
}

```

### Output

6 characters were converted.  
The converted string is "string"

3 characters were converted.  
The converted string is "str"

### Related Information

- “stdlib.h” on page 45
- “mbstowcs() — Convert Multibyte Characters to Wide Characters” on page 804
- “wcslen() — Calculate Length of Wide-Character String” on page 1715
- “wcsrtombs() — Convert Wide-Character String to Multibyte String” on page 1726
- “wctomb() — Convert Wide Character to Multibyte Character” on page 1751

## wcstoul() — Convert a Wide-Character String to an Unsigned Long Integer

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h>
```

```
unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr, int base);
```

### General Description

Converts the initial portion of the wide-character string pointed to by *nptr* to unsigned long integer representation. First it decomposes the input wide-character string into three parts:

1. An initial, possibly empty, sequence of white-space wide characters (as specified by the `iswspace()` function).
2. A subject sequence resembling an unsigned integer represented in some radix determined by the value of `base`.
3. A final wide-character string of one or more unrecognized wide characters, including the terminating null character of the input wide-character string.

Then it attempts to convert the subject sequence to an unsigned integer, and returns the result.

If the value of `base` is zero, the expected form of the subject sequence is that of an integer constant as described in ISO/IEC 9899: subclause 6.1.3.2, optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of `base` is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits from the portable character set representing an integer with the radix specified by `base`, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from a (or A) through z (or Z) are ascribed the values 10 to 35; only letters whose ascribed values are less than that of `base` are permitted. If the value of `base` is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The *subject sequence* is defined as the longest initial sub-sequence of the input wide-character string, starting with the first non-white-space wide character, that is of the expected form. The subject sequence contains no wide characters if the input wide-character string is empty or consists entirely of white-space, or if the first non-white-space wide character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of `base` is zero, the sequence of wide characters starting with the first digit is interpreted as an integer constant according to the rules of ISO/IEC 9899: subclause 6.1.3.2. If the subject sequence has the expected form and the value of `base` is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conver-

sion is negated. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

In a locale other than the C locale, additional implementation-defined subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

The behavior of this wide-character function is affected by the LC\_CTYPE category of the current locale. If you change the category, undefined results can occur.

## Returned Value

Returns the converted value, if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, ULONG\_MAX is returned, and the value of the macro ERANGE is stored in *errno*.

## Example CBC3BW25

```
/* CBC3BW25 */
#include <stdio.h>
#include <wchar.h>

#define BASE 2

int main(void)
{
    wchar_t *wcs = L"1000e13 camels";
    wchar_t **endptr;
    unsigned long int answer;

    answer = wcstoul(wcs, endptr, BASE);
    printf("The input wide string used: %ls \n", wcs);
    printf("The unsigned long int produced: %lu\n", answer);
    printf("The substring of the input wide string that was not");
    printf(" converted to unsigned long: %ls \n", *endptr);
}
```

## Related Information

- “wchar.h” on page 54
- “strtoul() — Convert String to Unsigned Integer” on page 1462

## wcswcs() — Locate Wide-Character Substring in Wide Character String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h>
```

```
wchar_t *wcswcs(const wchar_t *string1, const wchar_t *string2);
```

### General Description

Locates the first occurrence in the string pointed to by *string1* of the sequence of wide characters (excluding the terminating wide null character) in the string pointed to by *string2*.

The behavior of this wide-character function is affected by the LC\_CTYPE category of the current locale. If you change the category, undefined results can occur.

This function was formerly declared in `wcstr.h`, which continues to be maintained for compatibility.

### Returned Value

Returns a pointer to the located string or NULL if the string is not found. If *string2* points to a string with zero length, the function returns *string1*.

### Example

#### CBC3BW26

```
/* CBC3BW26
   This example finds the first occurrence of the wide character string
   pr in buffer1, using wcswcs().
*/
#include <stdio.h>
#include <wchar.h>

#define SIZE 40

int main(void)
{
    wchar_t buffer1[SIZE] = L"computer program";
    wchar_t * ptr;
    wchar_t * wch = L"pr";

    ptr = wcswcs( buffer1, wch );
    printf( "The first occurrence of %ls in '%ls' is '%ls'\n",
           wch, buffer1, ptr );
}
```

### Output

The first occurrence of pr in 'computer program' is 'program'

**Related Information**

- “wchar.h” on page 54
- “strstr() — Locate Substring” on page 1453
- “wcschr() — Search for Wide-Character Substring” on page 1701
- “wcscmp() — Compare Wide-Character Strings” on page 1703
- “wcscspn() — Find Offset of First Wide-Character Match” on page 1709
- “wcpbrk() — Locate First Wide Characters in String” on page 1722
- “wcsrchr() — Locate Last Wide Character in String” on page 1724

## wcswidth() — Determine the Display Width of a Wide-Character String

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#include <wchar.h>
```

```
int wcswidth(const wchar_t *wcs, size_t n);
```

### General Description

Determines the number of printing positions that a graphic representation of *n* wide characters (or fewer than *n* wide characters, if a null wide character is encountered before *n* wide characters have been exhausted), in the wide-character string pointed to by *wcs*, occupies on a display device. The number of printing positions is independent of its location on the device.

### Returned Value

Returns either the value 0 (if *wcs* points to a null wide character), or returns the number of printing positions occupied by the wide-character string pointed to by *wcs*. The value -1 is returned if any wide character in the wide-character string pointed to by *wcs* is not a printing wide character.

The behavior of the `wcswidth()` function is affected by the `LC_CTYPE` category.

**Note:** Under OS/390 C/C++ applications, the width returned will be 1 for each single-byte character and 2 for each double-byte character.

### Example

#### CBC3BW27

```
/* CBC3BW27 */
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *wcs = L"ABC";

    printf("wcs has a width of: %d\n", wcswidth(wcs,3));
}
```

### Output

```
wcs has a width of: 3
```

### Related Information

- “wchar.h” on page 54



## wcsxfrm() — Transform a Wide-Character String

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h>
```

```
size_t wcsxfrm(wchar_t *wcs1, const wchar_t *wcs2, size_t n);
```

### General Description

Transforms the wide-character string pointed to by `wcs2` to values which represent character collating weights and places the resulting wide-character string into the array pointed to by `wcs1`. The transformation is such that if the `wscmp()` function is applied to two transformed wide-character strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the `wscoll()` function applied to the same two original wide-character strings. No more than `n` elements are placed into the resulting array pointed to by `wcs1`, including the terminating null wide character code. If `n` is zero, `wcs1` is permitted to be a null pointer. If copying takes place between objects that overlap, the behavior is undefined.

### Returned Value

Returns the length of the transformed wide-character string (not including the terminating null wide character code). If the value returned is `n` or more, the contents of the array pointed to by `wcs1` are indeterminate.

If `wcs1` is a null pointer, `wcsxfrm()` returns the number of elements required to contain the transformed wide string.

The transformed value of invalid wide character codes shall be either less than or greater than the transformed values of valid wide character codes depending on the option chosen for the particular locale definition. In this case `wcsxfrm()` returns `(size_t)-1`.

The `wcsxfrm()` function is controlled by the `LC_COLLATE` category.

The `EILSEQ` error may be set, indicating that the wide character string pointed to by `wcs2` contains wide character codes outside the domain of the collating sequence.

**Note:** The ISO/C Multibyte Support Extensions do not indicate that the `wcsxfrm()` function may return with an error.

### Example

#### CBC3BW28

```
/* CBC3BW28 */
#include <stdio.h>
#include <wchar.h>

int main(void)
{
```

```
wchar_t *wcs;
wchar_t  buffer[80];
int      length;

printf("Type in a string of characters.\n");
wcs = fgetws(buffer, 80, stdin);
length = wcsxfrm(NULL, wcs, 0);
printf("You would need a %d element array to hold the wide string", length);
printf("\n\n%s\n\ntransformed according", wcs);
printf(" to this program's locale.\n");
}
```

**Related Information**

- “wchar.h” on page 54
- “strxfrm() — Transform String” on page 1465

## wctob() — Convert Wide Character to Byte

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

### Format

#### Non-XPG4

```
#include <wchar.h>

int wctob(wint_t c);
```

#### XPG4

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

int wctob(wint_t c);
```

### General Description

Determines whether *c* corresponds to a member of the extended character set whose multibyte character corresponds to a single byte when in initial shift state.

The behavior of this wide-character function is affected by the LC\_CTYPE category of the current locale. If you change the category, undefined results can occur.

### Special Behavior for XPG4

If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the *wchar* header, then you must also define the *\_MSE\_PROTOS* feature test macro to make the declaration of the *wctob()* function in the *wchar* header available when you compile your program. Please see Table 3 on page 18 for a list of XPG4 and other feature test macros.

### Returned Value

Returns EOF if *c* does not correspond to a multibyte character with length one; otherwise it returns the single-byte representation.

### Example

#### CBC3BW29

```
/* CBC3BW29 */
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wint_t wc = L'A';

    if (wctob(wc) == wc)
        printf("wc is a valid single byte character\n");
    else
        printf("wc is not a valid single byte character\n");
}
```

**Output**

wc is a valid single byte character

**Related Information**

- “wchar.h” on page 54

## wctomb() — Convert Wide Character to Multibyte Character

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <stdlib.h>
```

```
int wctomb(char *string, wchar_t character);
```

### General Description

Converts the `wchar_t` value of *character* into a multibyte array pointed to by *string*. If the value of *character* is 0, the function is left in the initial shift state. At most, `wctomb()` stores `MB_CUR_MAX` characters in *string*.

The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you change the category, undefined results can occur.

### Returned Value

Returns the length in bytes of the multibyte character. The value `-1` is returned if *character* is not a valid multibyte character. If *string* is a NULL pointer, the `wctomb()` function returns nonzero if shift-dependent encoding is used, or 0 otherwise.

### Example CBC3BW30

```
/* CBC3BW30
   This example converts the wide character c to a character using wctomb().
*/
#include <stdio.h>
#include <stdlib.h>

#define SIZE 40

int main(void)
{
    static char  buffer[ SIZE ];
    wchar_t wch = L'c';
    int length;

    length = wctomb( buffer, wch );
    printf( "The number of bytes that comprise the multibyte "
           "character is %i\n", length );
    printf( "And the converted string is \"%s\"\n", buffer );
}
```

### Output

```
The number of bytes that comprise the multibyte character is 1
And the converted string is "c"
```

## Related Information

- “stdlib.h” on page 45
- “mbtowc() — Convert Multibyte Character to Wide Character” on page 806
- “wctomb() — Convert a Wide Character to a Multibyte Character” on page 1697
- “wcslen() — Calculate Length of Wide-Character String” on page 1715
- “wcstombs() — Convert Wide-Character String to Multibyte Character String” on page 1740

## wctype() — Obtain Handle for Character Property Classification

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment XPG4 XPG4.2	both	

### Format

```
#include <wchar.h>
```

```
wctype_t wctype(const char *property);
```

### General Description

The `wctype()` function is defined for valid property names as defined in the current locale. The *property* is a string identifying a generic character class for which code-page-specific type information is required. The function returns a value of type `wctype_t`, which can be used as the second argument to a call of `iswctype()`. The `wctype()` function determines values of `wctype_t` according to rules of the coded character set defined by character type information in the program's locale (category `LC_CTYPE`). Values returned by `wctype()` are valid until a call to `setlocale()` that modifies the category `LC_CTYPE`.

The behavior of this wide-character function is affected by the `LC_CTYPE` category of the current locale. If you change the category, undefined results can occur.

### Returned Value

Returns zero if the given property name is not valid for the current locale (category `LC_CTYPE`); otherwise it returns a value of type `wctype_t` that can be used in calls to `iswctype()`.

### Related Information

- “wctype.h” on page 56

## wcwidth() — Determine the Display Width of a Wide Character

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

#### Non-XPG4

```
#include <wchar.h>
```

```
int wcwidth(const wint_t wc);
```

#### XPG4

```
#define _XOPEN_SOURCE
#include <wchar.h>
```

```
int wcwidth(const wchar_t wc);
```

### General Description

Determines the number of printing positions that a graphic representation of *wc* occupies on a display device. Each of the printing wide characters occupies its own number of printing positions on a display device. The number is independent of its location on the device.

### Special Behavior for XPG4

If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the `wchar` header, then the compiler assumes that your program is using the XPG4 variety of the `wcwidth()` function. Please see Table 3 on page 18 for a list of XPG4 and other feature test macros.

The prototype for the XPG4 variety of the `wcwidth()` function is:

```
int wcwidth(const wchar_t wc);
```

The difference between this variety and the C/370, non-XPG4 variety of the `wcwidth()` function is that its parameter, *wc*, is a `wchar_t` rather than a `wint_t` type.

### Returned Value

Returns the value 0 (if *wc* is a null or non-spacing wide character), or returns the number of printing position occupied by *wc*, or returns the value -1 (if *wc* is not a printing wide character).

The behavior of the `wcwidth()` function is affected by the `LC_CTYPE` category.

### Notes:

1. Under OS/390 C/C++ applications, the width returned will be zero for a null or non-spacing character, 1 for a single-byte character, and 2 for a double-byte character.
2. A non-spacing character is a character belonging to the charclass named `_zlc` (zero length character class) in the `LC_CTYPE` category.



### Example

#### CBC3BW31

```
/* CBC3BW31 */
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wint_t wc = L'A';

    printf("wc has a width of: %d\n", wctype(wc));
}
```

### Output

wc has a width of: 1

### Related Information

- “wchar.h” on page 54

## w\_getmntent() — Get Information on Mounted File Systems

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 4.3

### Format

```
#include <sys/mntent.h>
```

```
int w_getmntent(char *buffer, int size);
```

### General Description

Gets information about all currently mounted file systems.

- buffer* A pointer to storage that is filled with the retrieved information. The returned information is mapped by the sys/mntent.h header file and contains multiple entries, one for each mounted file system.
- size* The length of the buffer. If *size* is zero, the total number of mount entries is returned. You can use this information in a subsequent call to obtain a buffer large enough to hold all the information about all the entries.

A header is placed at the beginning of the buffer, and you should zero it out before the first call to w\_getmntent(). In the header, the field *mnt\_size* returns the number of bytes of data put in the buffer. The field *mnt\_cur* contains positioning information that w\_getmntent() uses to store the information. If multiple calls are made, use the same buffer because the positioning information in *mnt\_cur* indicates where the function should continue with its list. The positioning information should not be changed between calls.

Three fields of interest returned in the buffer are:

*mnt\_fsname*

The file system name, which is up to 45 characters long and ends with a null. The process can use this field to obtain more information using w\_statfs().

**mnt\_fsname** corresponds to the *filesystem* argument for mount().

*mnt\_mountpoint*

The path name of the directory where the file system is mounted. This field ends with a null.

If the caller of w\_getmntent() lacks search authorization to one or more of the directories in the mount point path name, **mnt\_mountpoint** is returned empty. That is, **mnt\_pathlen** is zero and **mnt\_mountpoint** contains a null as the first character.

*mnt\_parm*

The file-system-specific parameter specified on the mount() function when the file system was mounted. This field ends with a null.

If no parameter was specified, **mnt\_parmlen** and **mnt\_parmoffset** are each zero. If a parameter was specified, its address is the sum of the address of **w\_mntent** and **mnt\_parmoffset**.

If all entries do not fit in the buffer supplied, multiple calls are required. If an entry together with its mount parameter will not fit in the buffer, the entry is returned without the mount parameter. In this case, **mnt\_parmlen** contains the length of the mount parameter, and **mnt\_parmoffset** is zero.

To assure that at least one entry, including the mount parameter, is returned, it is advisable to allocate space for at least two entries.

When the final entry has been placed in the buffer, `w_getmntent()` returns no entries.

## Returned Value

If successful, `w_getmntent()` returns the number of entries in the buffer. If unsuccessful, it returns the value `-1` and sets `errno` to one of the following:

**EINVAL**      A parameter was specified incorrectly.

**ERANGE**     The result is too large to fit in the available buffer space.

## Example CBC3BW32

```
/* CBC3BW32
   This example gets information about currently mounted systems, using
   wgetmnt().
*/
#include <sys/mntent.h>
#include <stdio.h>

main() {
    int entries, entry;
    struct {
        struct w_mnth header;
        struct w_mntent mount_table[10];
    } work_area;
    char *parm_ptr;

    memset(&work_area, 0x00, sizeof(work_area));
    do {
        if ((entries = w_getmntent((char *) &work_area,
                                   sizeof(work_area))) == -1)
            perror("w_getmntent() error");

        else for (entry=0; entry<entries; entry++) {
            printf("filesystem %s is mounted at %s\n",
                   work_area.mount_table[entry].mnt_fsname,
                   work_area.mount_table[entry].mnt_mountpoint);
            if (work_area.mount_table[entry].mnt_parmoffset != 0) {
                parm_ptr = ((char *)&(work_area.mount_table[entry])) +
                           work_area.mount_table[entry].mnt_parmoffset;
                printf(" mount parameter is %s",*parm_ptr);
            }
        }
    } while (entries > 0);
}
```

## Output

```
filesystem POSIX.NEW.HFS is mounted at /new_fs
filesystem POSIX.ROOT.FS is mounted at /
filesystem Memphis.data is mounted at /memphis
mount parameter is memphis1:/usr/remote/Memphis.data
```

**Related Information**

- “sys/mntent.h” on page 47
- “statvfs() — Get File System Information” on page 1408
- “w\_statfs() — Get the File System Status” on page 1789

## w\_getpsent() — Get Process Data

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 4.3

### Format

```
#include <sys/ps.h>
```

```
int w_getpsent(int token, W_PSPROC *buffptr, size_t length);
```

### General Description

Provides information about the status of any process that the calling process has access to.

*token*            A relative number that identifies the relative position of a process in the system. Zero represents the first process in the system. On the first call to w\_getpsent(), pass the token 0; the function then returns the token that identifies the next process to which the caller has access. Use that token on the next call.

*buffptr*           The address of the buffer where the data is to be stored.

*length*            The length of the buffer.

The data returned is described in the ps.h header file. See Table 45 for the format of the structure stored in the buffer.

Table 45 (Page 1 of 2). Variables Stored in Structure

Variable Type	Variable Name	General Description
unsigned int	ps_state	Process state
pid_t	ps_pid	Process ID
pid_t	ps_ppid	Parent ID
pid_t	ps_sid	Session ID (leader)
pid_t	ps_pgid	Process group ID
pid_t	ps_fgid	Foreground process group ID
uid_t	ps_euid	Effective user ID
uid_t	ps_ruid	Real user ID
uid_t	ps_suid	Saved set user ID
gid_t	ps_egid	Effective group ID
gid_t	ps_rgid	Real group ID
gid_t	ps_sgid	Saved set group ID
long	ps_size	Total size
time_t	ps_starttime	Starting time
clock_t	ps_usertime	User CPU time
clock_t	ps_sysstime	System CPU time
int	ps_conttylen	Length of ConTTY

Table 45 (Page 2 of 2). Variables Stored in Structure

Variable Type	Variable Name	General Description
char	*ps_conttyptr	Controlling terminal
int	ps_pathlen	Length of <i>arg0</i>
char	*ps_pathptr	File name
int	ps_cmdlen	Length of command
char	*ps_cmdptr	Command and arguments

## Returned Value

When successful, `w_getpsent()` returns the process token for the next process for which the caller has access. For the last active process to which the user has access, `w_getpsent()` returns zero, indicating there are no more processes to be accessed. If unsuccessful, `w_getpsent()` returns the value `-1` and sets `errno` to `EINVAL`, which indicates an incorrect process token.

## Example CBC3BW33

```

/* CBC3BW33
   This example provides status information, using wgetpsent().
*/
#include <stdio.h>
#include <sys/ps.h>
#include <sys/types.h>
#include <pwd.h>
#include <time.h>

main() {
    int token;
    W_PSPROC buf;
    struct passwd *pw;

    token = 0;

    memset(&buf, 0x00, sizeof(buf));
    buf.ps_conttyptr = (char *) malloc(buf.ps_conttylen = PS_CONTTYBLEN);
    buf.ps_pathptr = (char *) malloc(buf.ps_pathlen = PS_PATHBLEN);
    buf.ps_cmdptr = (char *) malloc(buf.ps_cmdlen = PS_CMDBLEN);
    if ((buf.ps_conttyptr == NULL) ||
        (buf.ps_pathptr == NULL) ||
        (buf.ps_cmdptr == NULL))
        perror("buffer allocation error");

    else do {
        if ((token = w_getpsent(token, &buf, sizeof(buf))) == -1)
            perror("w_getpsent() error");
        else if (token > 0)
            if ((pw = getpwuid(buf.ps_ruid)) == NULL)
                perror("getpwuid() error");
            else printf("token %d: pid %10d, user %8s, started %s", token,
                        (int) buf.ps_pid, pw->pw_name,
                        ctime(buf.ps_starttime));
    } while (token > 0);
}

```

## Output

```
token 2: pid      131074, user  MVSUSR1, started Fri Jun 16 08:09:17 1995
token 3: pid      65539, user  MVSUSR1, started Fri Jun 16 08:09:41 1995
token 6: pid     589830, user  MVSUSR1, started Fri Jun 16 10:29:17 1995
token 7: pid     851975, user  MVSUSR1, started Fri Jun 16 10:30:04 1995
```

**Related Information**

- “sys/ps.h” on page 48

## w\_ioctl(), \_\_w\_pioctrl() — Control of Devices

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 4.3

### Format

```
#define _OPEN_SYS
#include <termios.h>

int w_ioctl(int fildev,
            int cmd,
            int arglen,
            void *arg);

int __w_pioctrl(const char *pathname,
               int cmd,
               int arglen,
               void *arg);
```

### General Description

The `w_ioctl()` and `__w_pioctrl()` functions are general entry points for device-specific commands. The specific actions specified by `w_ioctl()` and `__w_pioctrl()` vary with the device, and they are defined by the device driver.

*fildev*        A descriptor for an open character special file (used by `w_ioctl()`).

*pathname*     The pathname of a file (used by `__w_pioctrl()`).

*cmd*           The command to be passed to the device driver, specified as an integer value.

*arglen*        The length of the argument passed to the device driver, specified as a value from 1 to 1024 bytes.

*arg*            The address of the buffer where the argument to be passed to the device driver is stored.

`w_ioctl()` and `__w_pioctrl()` pass the *cmd*, *arglen*, and *arg* arguments to the device driver to be interpreted and processed. When `w_ioctl()` and `__w_pioctrl()` complete successfully, the device driver returns *arglen* and *arg*, if appropriate.

### Returned Value

If successful, `w_ioctl()` returns zero. If unsuccessful, it returns the value `-1` and sets `errno` to one of the following:

**EINVAL**       An incorrect length was specified for *arglen*. The correct argument length range is 0 to 1024.

**ENODEV**       The device does not exist. The function is not supported by the device driver.

**ENOTTY**       An incorrect file descriptor was specified. *fildev* was not a character special file.



**ENAMETOOLONG**

The length of the *pathname* argument exceeds {PATH\_MAX}, or a *pathname* component is longer than {NAME\_MAX} and {\_POSIX\_NO\_TRUNC} is in effect for that file. For symbolic links, the length of the pathname string substituted for a symbolic link exceeds {PATH\_MAX}. {PATH\_MAX} and {NAME\_MAX} values can be determined by using pathconf() .

**ENOENT** Either there is no file named *pathname* or *pathname* is an empty string.

**ENOTDIR** A component of the *pathname* prefix is not a directory.

### Example

#### CBC3BW34

```
/* CBC3BW34
   This example shows a general entry point for device-specific commands.
   */
#include <termios.h>
#include <stdio.h>

main() {
    char buf[256];
    int ret;

    memset(buf, 0x00, sizeof(buf));
    if ((ret = w_ioctl(0, 1, sizeof(buf), buf)) != 0)
        perror("w_ioctl() error");
    else
        printf("w_iotctl() returned '%s'\n", buf);
}
```

**Output**

w\_ioctl() error: Invalid argument

**Note:** w\_ioctl() is dependent upon the file system device driver. There are no documented commands for the HFS file system.

**Related Information**

- “termios.h” on page 51
- “ioctl() — Control Device” on page 672
- “tcdrain() — Wait Until Output Has Been Transmitted” on page 1507
- “tcflow() — Suspend or Resume Data Flow on a Terminal” on page 1509
- “tcflush() — Flush Input or Output on a Terminal” on page 1512
- “tcgetattr() — Get the Attributes for a Terminal” on page 1515
- “tcgetpgrp() — Get the Foreground Process Group ID” on page 1520
- “tcsetattr() — Set the Attributes for a Terminal” on page 1531
- “tcsendbreak() — Send a Break Condition to a Terminal” on page 1529
- “tcsetpgrp() — Set the Foreground Process Group ID” on page 1546

## wmemchr — Locate Wide Character

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

### Format

#### Non-XP4

```
#include <wchar.h>
```

```
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
```

#### XP4

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>
```

```
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
```

### General Description

Locates the first occurrence of the wide character *c* in the initial *n* wide chars of the object pointed to by *s*.

If *n* has the value zero, *wmemchr* finds no occurrence of *c* and returns a null pointer.

### Special Behavior for XP4

If you define any feature test macro specifying XP4 behavior before the statement in your program source file to include the *wchar* header, then you must also define the *\_MSE\_PROTOS* feature test macro to make the declaration of the *wmemchr* function in the *wchar* header available when you compile your program. See Table 3 on page 18 for a list of XP4 and other feature test macros.

### Returned Value

The *wmemchr* function returns a pointer to the first occurrence of the wide character *c* in object *s*, or the null pointer if the wide character *c* does not occur within the first *n* wide characters of *s*.

### Example

```
#include <stdio>
#include <wchar>

main()
{
    wchar_t  *in = L"1234ABCD";
    wchar_t  *ptr;
    wchar_t  fnd = L'A';

    printf("\nEXPECTED: ABCD");
    ptr = wmemchr(in, L'A', 6);
    if (ptr == NULL)
        printf("\n** ERROR ** ptr is NULL, char L'A' not found\n");
    else
```

```
printf("\nRECEIVED: %ls \n",ptr);  
  
}
```

**Related Information**

- “wchar.h” on page 54
- “wmemcmp — Compare Wide Character” on page 1766
- “wmemcpy — Copy Wide Character” on page 1768
- “wmemmove — Move Wide Character” on page 1770
- “wmemset — Set Wide Character” on page 1772

## wmemcmp — Compare Wide Character

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

### Format

#### Non-XPG4

```
#include <wchar.h>
```

```
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

#### XPG4

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>
```

```
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

### General Description

Compares the first  $n$  wide chars of the object pointed to by  $s1$  to the first  $n$  wide chars of the object pointed to by  $s2$ .

If  $n$  has the value zero, `wmemcmp` returns zero.

### Special Behavior for XPG4

If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the `wchar` header, then you must also define the `_MSE_PROTOS` feature test macro to make the declaration of the `wmemcmp` function in the `wchar` header available when you compile your program. Please see Table 3 on page 18 for a list of XPG4 and other feature test macros.

### Returned Value

The `wmemcmp` function returns an integer that is less than zero if  $s1$  is less than  $s2$ ; equal to zero if  $s1$  is equal to  $s2$ ; and greater than zero if  $s1$  is greater than  $s2$ .

### Example

```
#include <wchar.h>
#include <stdio.h>

main()
{
    int      ptr;
    wchar_t  *in  = L"12345678";
    wchar_t  *out = L"12AAAAAB";

    printf("\nGREATER is the expected result");
    ptr = wmemcmp(in, out, 3);
    if (ptr == 0)
        printf("\nArrays are EQUAL %ls %ls \n", in, out);
    else
    {
        if (ptr > 0)
```

```
        printf("\nArray %ls GREATER than %ls \n",in, out);  
    else  
        printf("\nArray %ls LESS than %ls \n",in, out);  
    }  
}
```

**Related Information**

- “wchar.h” on page 54
- “wmemchr — Locate Wide Character” on page 1764
- “wmemcpy — Copy Wide Character” on page 1768
- “wmemmove — Move Wide Character” on page 1770
- “wmemset — Set Wide Character” on page 1772

## wmemcpy — Copy Wide Character

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

### Format

#### Non-XPG4

```
#include <wchar.h>
```

```
wchar_t *wmemcpy(wchar_t *s1, const wchar_t *s2, size_t n);
```

#### XPG4

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>
```

```
wchar_t *wmemcpy(wchar_t *s1, const wchar_t *s2, size_t n);
```

### General Description

Copies *n* wide chars of the object pointed to by *s2* to the object pointed to by *s1*.

Result of the copy is unpredictable if *s1* and *s2* overlap. If *n* has the value zero, *wmemcpy* copies zero wide characters.

### Special Behavior for XPG4

If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the *wchar* header, then you must also define the *\_MSE\_PROTOS* feature test macro to make the declaration of the *wmemcpy* function in the *wchar* header available when you compile your program. Please see Table 3 on page 18 for a list of XPG4 and other feature test macros.

### Returned Value

The *wmemcpy* function returns the value of *s1*.

### Example

```
#include <wchar.h>
#include <stdio.h>

main()
{
    wchar_t *in    = L"12345678";
    wchar_t *out   = L"ABCDEFGH";
    wchar_t *ptr;

    printf("-nExpected result: First 4 chars of in change");
    printf(" and are the same as first 4 chars of out");
    ptr = wmemcpy(in, out, 4);
    if (ptr == in)
        printf("-nArray in %ls array out %ls -n", in, out);
    else
    {
        printf("-n*** ERROR ***");
    }
}
```

```
        printf(" returned pointer wrong");  
    }  
}
```

**Related Information**

- “wchar.h” on page 54
- “wmemchr — Locate Wide Character” on page 1764
- “wmemcmp — Compare Wide Character” on page 1766
- “wmemmove — Move Wide Character” on page 1770
- “wmemset — Set Wide Character” on page 1772

## wmemmove — Move Wide Character

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

### Format

#### Non-XP4

```
#include <wchar.h>
```

```
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
```

#### XP4

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>
```

```
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
```

### General Description

Copies  $n$  wide chars of the object pointed to by  $s2$  to the object pointed to by  $s1$ . Copying takes place as if the  $n$  wide characters from  $s2$  are first copied into a temporary array of  $n$  wide characters that does not overlay the objects pointed to by  $s1$  and  $s2$ , and then copied from the temporary array into the object pointed to by  $s1$ .

If  $n$  has the value zero, `wmemmove` copies zero wide characters.

### Special Behavior for XP4

If you define any feature test macro specifying XP4 behavior before the statement in your program source file to include the `wchar` header, then you must also define the `_MSE_PROTOS` feature test macro to make the declaration of the `wmemmove` function in the `wchar` header available when you compile your program. Please see Table 3 on page 18 for a list of XP4 and other feature test macros.

### Returned Value

The `wmemmove` function returns the value of  $s1$ .

### Example

```
#include <wchar.h>
#include <stdio.h>

main()
{
    wchar_t *in   = L"12345678";
    wchar_t *out  = L"ABCDEFGH";

    wchar_t *ptr;

    printf("\nExpected result: First 4 chars of in and out the same");
    ptr = wmemmove(in, out, 4);
    if (ptr == in)
        printf("\nArray in %ls array out %ls \n", in, out);
```



```
else
{
    printf("\n*** ERROR ***");
    printf("    Returned pointer not correct.\n");
}
}
```

**Related Information**

- “wchar.h” on page 54
- “wmemchr — Locate Wide Character” on page 1764
- “wmemcmp — Compare Wide Character” on page 1766
- “wmemcpy — Copy Wide Character” on page 1768
- “wmemset — Set Wide Character” on page 1772

## wmemset — Set Wide Character

### Standards

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

### Format

#### Non-XP4

```
#include <wchar.h>
```

```
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

#### XP4

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>
```

```
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

### General Description

Copies the value of *c* into the first *n* wide chars of the object pointed to by *s*.

If *n* has the value zero, *wmemset* copies zero wide characters.

### Special Behavior for XP4

If you define any feature test macro specifying XP4 behavior before the statement in your program source file to include the *wchar* header, then you must also define the *\_MSE\_PROTOS* feature test macro to make the declaration of the *wmemset* function in the *wchar* header available when you compile your program. Please see Table 3 on page 18 for a list of XP4 and other feature test macros.

### Returned Value

The *wmemset* function returns the value of *s*.

### Example

```
#include <wchar.h>
#include <stdio.h>

void main()
{
    wchar_t *in = L"1234ABCD";
    wchar_t *ptr;

    printf("\nEXPECTED: AAAAAACD");
    ptr = wmemset(in, L'A', 6);
    if (ptr == in)
        printf("\nResults returned - %ls \n", ptr);
    else
    {
        printf("\n** ERROR ** wrong pointer returned\n");
    }
}
```

**Related Information**

- “wchar.h” on page 54
- “wmemchr — Locate Wide Character” on page 1764
- “wmemcmp — Compare Wide Character” on page 1766
- “wmemcpy — Copy Wide Character” on page 1768
- “wmemmove — Move Wide Character” on page 1770

## wordexp() — Perform Shell Word Expansions

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	POSIX(ON) MVS 4.3

### Format

```
#define _XOPEN_SOURCE
#include <wordexp.h>
```

```
int wordexp(const char *words, wordexp_t *pwordexp, int flags);
```

```
void wordfree(wordexp_t *pwordexp);
```

### General Description

The `wordexp()` function performs word expansions as described in the *X/Open CAE Specification, Commands and Utilities, Issue 4, Version 2 (XCU) Section 2.6, Word Expansions*, subject to quoting as in the **XCU** specification, **Section 2.2, Quoting**, and places the list of expanded words into the structure pointed to by `pwordexp`.

The `words` argument is a pointer to a string containing one or more words to be expanded. The expansions will be the same as would be performed by the shell described by the **XCU** specification if `words` were the part of a command line representing the arguments to a utility. Therefore, `words` must not contain an unquoted <newline> or any of the unquoted special shell characters:

```
|  &  ;  <  >
```

except in the context of command substitution as specified in the **XCU** specification, **Section 2.6.3, Command Substitution**. It also must not contain unquoted parentheses or braces, except in the context of command or variable substitution.

The structure type **wordexp\_t** is defined in the header `<wordexp.h>` and includes the following members:

Member Type	Member Name	Description
size_t	we_wordc	Count of words matched by <code>words</code> .
char **	we_wordv	Pointer to list of expanded words.
size_t	we_offs	Slots to reserve at the beginning of <code>pwordexp-&gt;we_wordv</code> .

The `wordexp()` function stores the number of generated words into `pwordexp->we_wordc` and a pointer to a list of pointers to words in `pwordexp->we_wordv`. Each individual field created during field splitting (see the **XCU** specification, **Section 2.6.5, Field Splitting**) or pathname expansion (see the **XCU** specification, **Section 2.6.6, Pathname Expansion**) is a separate word in the `pwordexp->we_wordv` list. The words are in order as described in the **XCU** specification, **Section 2.6, Word Expansions**. The first pointer after the last word pointer will be a null pointer. The expansion of special parameters described in the **XCU** specification, **Section 2.5.2, Special Parameters** is unspecified.

It is the caller's responsibility to allocate the storage pointed to by `pwordexp`. The `wordexp()` function allocates the other space as needed, including memory pointed

to by *pwordexp*->we\_wordv. The wordfree() function frees any memory associated with *pwordexp* from a previous call to wordexp().

The *flags* argument is used to control the behavior of wordexp(). The values of *flags* is the bitwise inclusive OR or zero or more of the following constants, which are defined in <wordexp.h>:

#### WRDE\_APPEND

Append words generated to the ones from a previous call to wordexp().

#### WRDE\_DOOFFS

Make use of *pwordexp*->we\_offs. If this flag is set, *pwordexp*->we\_offs is used to specify how many null pointers to add to the beginning of *pwordexp*->we\_wordv. In other words, *pwordexp*->we\_wordv will point to *pwordexp*->we\_offs null pointers followed by *pwordexp*->we\_wordc word pointers, followed by a null pointer.

#### WRDE\_NOCMD

Fail if command substitution, as specified in the **XCU** specification, **Section 2.6.3, Command Substitution**, is requested.

#### WRDE\_REUSE

The *pwordexp* argument was passed to a previous successful call to wordexp(), and has not been passed to wordfree(). The result will be the same as if the application had called wordfree() and then called wordexp() without WRDE\_REUSE.

#### WRDE\_SHOWERR

Do not redirect stderr to /dev/null.

#### WRDE\_UNDEF

Report error on an attempt to expand an undefined shell variable.

The WRDE\_APPEND flag can be used to append a new set of words to those generated by a previous call to wordexp(). The following rules apply when two or more calls to wordexp() are made with the same value of *pwordexp* and without intervening calls to wordfree():

1. The first such call must not set WRDE\_APPEND. All subsequent calls must set it.
2. All of the calls must set WRDE\_DOOFFS, or all must not set it.
3. After the second and each subsequent call, *pwordexp*->we\_wordv will point to a list containing the following:
  - a. zero or more null pointers, as specified by WRDE\_DOOFFS and *pwordexp*->we\_offs
  - b. pointers to the words that were in the *pwordexp*->we\_wordv list before the call, in the same order as before
  - c. pointers to the new words generated by the latest call, in the specified order
4. The count returned in *pwordexp*->we\_wordc will be the total number of words from all of the calls.
5. The application can change any of the fields after a call to wordexp(), but if it does, it must reset them to the original value before a subsequent call, using

the same *wordexp* value, to `wordfree()` or `wordexp()` with the `WRDE_APPEND` or `WRDE_REUSE` flag.

If *words* contains an unquoted:

`<newline> | & ; < > ( ) { }`

in an inappropriate context, `wordexp()` will fail, and the number of expanded words will be 0.

Unless `WRDE_SHOWERR` is set in *flags*, `wordexp()` will redirect `stderr` to `/dev/null` for any utilities executed as a result of command substitution while expanding *words*. If `WRDE_SHOWERR` is set, `wordexp()` may write messages to `stderr` if syntax errors are detected while expanding *words*.

If `WRDE_DOOFFS` is set, then *pwordexp*→*we\_offs* must have the same value for each `wordexp()` call and `wordfree()` call using a given *pwordexp*.

The following constants are defined in `<wordexp.h>` as error return values:

`WRDE_BADCHAR`

One of the unquoted characters:

`<newline> | & ; < > ( ) { }`

appears in *words* in an inappropriate context.

`WRDE_BADVAL`

Reference to undefined shell variable when `WRDE_UNDEF` is set in *flags*.

`WRDE_CMDSUB`

Command substitution requested when `WRDE_NOCMD` is set in *flags*.

`WRDE_NOSPACE`

Attempt to allocate memory failed.

`WRDE_SYNTAX`

Shell syntax error, such as unbalanced parentheses or unterminated string.

`WRDE_NOSYS`

(1) Posix shell not available, or (2) `wordexp()` invoked after `pthread_create()` has been used to create additional threads.

## Returned Value

If successful, `wordexp()` returns 0.

Otherwise, a nonzero value, as defined in `<wordexp.h>` and described above, is returned to indicate an error. If `wordexp()` returns the value `WRDE_NOSPACE`, then *pwordexp*→***we\_wordc***, and *pwordexp*→***we\_wordv*** will be updated to reflect any words that were successfully expanded. In other cases, they will not be modified.

## **Related Information**

- “wordexp.h” on page 56

## wordfree() — Perform Shell Word Expansions

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	

### Format

```
#define _XOPEN_SOURCE
#include <wordexp.h>
```

```
int wordexp(const char *words, wordexp_t *pwordexp, int flags);
```

```
void wordfree(wordexp_t *pwordexp);
```

### General Description

The `wordfree()` function frees any memory associated with *pwordexp* from a previous call to `wordexp()`. Please refer to the description of `wordexp()` for the rules governing use of `wordexp()` and `wordfree()`.

### Returned Value

The `wordfree()` function returns no value.

### Related Information

- “wordexp.h” on page 56



| **\_\_w\_pioc** —Control of Devices

| The information for this function is included in “w\_pioc(), \_\_w\_pioc() — Control of  
| Devices” on page 1762.

## write() — Write Data on a File or Socket

### Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	MVS 4.3

### Format

```
#define _POSIX_SOURCE
#include <unistd.h>
```

```
ssize_t write(int fs, const void *buf, size_t N);
```

### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <unistd.h>
```

```
ssize_t write(int fs, const void *buf, ssize_t N);
```

### Berkeley Sockets

```
#define _OE_SOCKETS
#include <unistd.h>
```

```
ssize_t write(int fs, const void *buf, ssize_t N);
```

### General Description

Writes *N* bytes from *buf* to the file or socket associated with *fs*. *N* should not be greater than INT\_MAX (defined in the limits.h header file). If *N* is zero, write() simply returns a value of zero without attempting any other action.

If *fs* refers to a regular file or any other type of file on which a process can seek, write() begins writing at the file offset associated with *fs*. A successful write() increments the file offset by the number of bytes written. If the incremented file offset is greater than the previous length of the file, the length of the file is set to the new file offset.

If *fs* refers to a file on which a process cannot seek, write() begins writing at the current position. There is no file offset associated with such a file.

If O\_APPEND (defined in the fcntl.h header file) is set for the file, write() sets the file offset to the end of the file before writing the output.

If there is not enough room to write the requested number of bytes (for example, because there is not enough room on the disk), write() outputs as many bytes as the remaining space can hold.

If write() is interrupted by a signal, the effect is one of the following:

- If write() has not written any data yet, it returns the value -1 and sets errno to EINTR.
- If write() has successfully written some data, it returns the number of bytes it wrote before it was interrupted.

Write operations on pipes or FIFO special files are handled in the same way a write operation on a regular file, with the following exceptions:

- A pipe has no associated file offset, so every write appends to the end of the pipe.
- If *N* is less than or equal to PIPE\_BUF, the output is not interleaved with data written by other processes that are writing to the same pipe. If *N* is greater than PIPE\_BUF bytes, the output can be interleaved with other data (regardless of the setting of O\_NONBLOCK, which is defined in the fcntl.h header file). A write to a pipe never returns with errno set to EINTR if it has transferred any data.
- If O\_NONBLOCK (defined in the fcntl.h header file) is not set, write() may block process execution until normal completion.
- If O\_NONBLOCK is set, write() does not block process execution. If *N* is less than or equal to PIPE\_BUF, write() succeeds completely and returns the value of *N*, or else it writes nothing, sets errno to EAGAIN, and returns the value -1. If *N* is greater than PIPE\_BUF, write() writes as many bytes as it can and returns this number as its result, or else it writes nothing, sets errno to EAGAIN, and returns the value -1.

With other files that support nonblocking writes and cannot accept data immediately, the effect is one of the following:

- If O\_NONBLOCK is not set, write() blocks until the data can be written.
- If O\_NONBLOCK is set, write() does not block the process. If some data can be written without blocking the process, write() writes what it can and returns the number of bytes written. Otherwise, it sets errno to EAGAIN and returns the value -1.

write() causes the signal SIGTTOU to be sent if all of these conditions are true:

- The process is attempting to write to its controlling terminal and TOSTOP is set as a terminal attribute.
- The process is running in a background process group and the SIGTTOU signal is not blocked or ignored.
- The process is not an orphan.

A successful write() updates the change and modification times for the file.

If *fs* refers to a socket, write() is equivalent to send() with no flags set.

## Behavior for Sockets

The write() call writes data from a buffer on a socket with descriptor *fs*. The socket must be a connected socket. This call writes up to *N* bytes of data.

### Parameter Description

<i>fs</i>	The file or socket descriptor.
<i>buf</i>	The pointer to the buffer holding the data to be written.
<i>N</i>	The length in bytes of the buffer pointed to by the <i>buf</i> parameter.

If there is not enough available buffer space to hold the socket data to be transmitted, and the socket is in blocking mode, write() blocks the caller until additional

buffer space becomes available. If the socket is in nonblocking mode, `write()` returns a `-1` and sets the error code to `EWOULDBLOCK`. See “`fcntl()` — Control Open File Descriptors” on page 350 or “`ioctl()` — Control Device” on page 672 for a description of how to set the nonblocking mode.

When the socket is not ready to accept data and the process is trying to write data to the socket:

- Unless `FNDELAY` or `O_NONBLOCK` is set, `write()` blocks until the socket is ready to accept data.
- If `FNDELAY` is set, `write()` returns a `0`.
- If `O_NONBLOCK` is set, `write()` does not block the process. If some data can be written without blocking the process, `write()` writes what it can and returns the number of bytes written. Otherwise, it sets the error code to `EAGAIN` and returns a `-1`.

For datagram sockets, this call sends the entire datagram, provided that the datagram fits into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application program wishes to send 1000 bytes, each call to this function can send 1 byte or 10 bytes or the entire 1000 bytes. Therefore, application programs using stream sockets should place this call in a loop, calling this function until all data has been sent.

### Special Behavior for C++ and Sockets

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

If successful, `write()` returns the number of bytes actually written, less than or equal to *N*. If unsuccessful, it returns the value `-1` and sets `errno` to one of the following:

A value of `0` or greater indicates the number of bytes sent. However, this does not assure that data delivery was complete. A connection can be dropped by a peer socket and a `SIGPIPE` signal generated at a later time if data delivery is not complete.

**EAGAIN** Resources temporarily unavailable. Subsequent requests may complete normally.

**EBADF** *fs* is not a valid file or socket descriptor.

**ECONNRESET**  
A connection was forcibly closed by a peer.

**EDESTADDRREQ**  
The socket is not connection-oriented and no peer address is set.

**EFAULT** Using the *buf* and *N* parameters would result in an attempt to access storage outside the caller's address space.

**EFBIG** Writing to the output file would exceed the maximum file size supported by the implementation.

**EINTR** `write()` was interrupted by a signal before it had written any output.

EINVAL	The request is invalid or not supported. The STREAM or multiplexer referenced by <i>fs</i> is linked (directly or indirectly) downstream from a multiplexer.
EIO	The process is in a background process group and is attempting to write to its controlling terminal, but TOSTOP (defined in the <code>termios.h</code> header file) is set, the process is neither ignoring nor blocking SIGTTOU signals, and the process group of the process is orphaned. An I/O error occurred.
EMSGSIZE	The message was too big to be sent as a single datagram.
ENOBUFS	Buffer space is not available to send the message.
ENOSPC	There is no available space left on the output device.
ENOTCONN	The socket is not connected.
ENXIO	A hang-up occurred on the STREAM being written to.
EPIPE	<code>write()</code> is trying to write to a pipe that is not open for reading by any other process. This error also generates a SIGPIPE signal. For a connected stream socket the connection to the peer socket has been lost.
ERANGE	The transfer request size was outside the range supported by the STREAMS file associated with <i>fs</i> .
EWouldBlock	The socket is in nonblocking mode and data is not available to write.

A write to a STREAMS file may fail if an error message has been received at the STREAM head. In this case, `errno` is set to the value included in the error message.

**Note:** OS/390 UNIX services do not supply any STREAMS devices or pseudodevices. It is impossible for `write()` to write any data on a STREAM. None of the STREAMS `errno`s will be visible to the invoker. See “`open()` — Open a File” on page 872 for more information.

### Example CBC3BW35

```

/* CBC3BW35
   This example writes a certain amount of bytes to a file, using write().
*/
#define _POSIX_SOURCE
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

#define mega_string_len 1000000

main() {
    char *mega_string;
    int fd, ret;
    char fn[]="write.file";

    if ((mega_string = (char*) malloc(mega_string_len)) == NULL)
        perror("malloc() error");
    else if ((fd = creat(fn, S_IWUSR)) < 0)

```

```

        perror("creat() error");
    else {
        memset(mega_string, '0', mega_string_len);
        if ((ret = write(fd, mega_string, mega_string_len)) == -1)
            perror("write() error");
        else printf("write() wrote %d bytes\n", ret);
        close(fd);
        unlink(fn);
    }
}

```

## Output

write() wrote 1000000 bytes

## Related Information

- “fcntl.h” on page 28
- “termios.h” on page 51
- “unistd.h” on page 53
- “connect() — Connect a Socket” on page 214
- “creat() — Create a New File or Rewrite an Existing One” on page 233
- “dup() — Duplicate an Open File Descriptor” on page 288
- “fcntl() — Control Open File Descriptors” on page 350
- “fwrite() — Write Items” on page 495
- “getsockopt() — Get the Options Associated with a Socket” on page 606
- “ioctl() — Control Device” on page 672
- “lseek() — Change the Offset of a File” on page 776
- “open() — Open a File” on page 872
- “pipe() — Create an Unnamed Pipe” on page 907
- “read() — Read From a File or Socket” on page 1080
- “readv() — Read Data on a File or Socket and Store in a Set of Buffers” on page 1093
- “recv() — Receive Data on a Socket” on page 1103
- “recvfrom() — Receive Messages on a Socket” on page 1106
- “recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers” on page 1110
- “select() — Monitor Activity on Files/Sockets and Message Queues” on page 1159
- “selectex() — Monitor Activity on Files/Sockets and Message Queues” on page 1166
- “send() — Send Data on a Socket” on page 1178
- “sendmsg() — Send Messages on a Socket” on page 1185
- “sendto() — Send Data on a Socket” on page 1190
- “setsockopt() — Set Options Associated with a Socket” on page 1270
- “socket() — Create a Socket” on page 1371
- “writev() — Write Data on a File or Socket Socket from an Array” on page 1785

## writev() — Write Data on a File or Socket Socket from an Array

### Standards

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	MVS 5.1

### Format

#### X/Open

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/uio.h>
```

```
ssize_t writev(int fs, const struct iovec *iov, int iovcnt);
```

#### Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/uio.h>
```

```
int writev(int fs, struct iovec *iov, int iovcnt);
```

### General Description

The `writev()` call writes data to a file or socket with descriptor `fs` from a set of buffers. The data is gathered from the buffers specified by `iov[0]...iov[iovcnt-1]`. When the descriptor refers to a socket, it must be a connected socket.

#### Parameter Description

<i>fs</i>	The file or socket descriptor.
<i>iov</i>	A pointer to an array of <b>iovec</b> buffers.
<i>iovcnt</i>	The number of buffers pointed to by the <i>iov</i> parameter.

The **iovec** structure is defined in **uio.h** and contains the following fields:

#### Element Description

<i>iov_base</i>	Pointer to the buffer.
<i>iov_len</i>	Length of the buffer.

This call writes the sum of the *iov\_len* bytes of data.

If there is not enough available buffer space to hold the socket data to be transmitted, and the socket is in blocking mode, `writev()` blocks the caller until additional buffer space becomes available. If the socket is in a nonblocking mode, `writev()` returns a `-1` and sets the error code to `EWOULDBLOCK`. See “`fcntl()` — Control Open File Descriptors” on page 350 or “`ioctl()` — Control Device” on page 672 for a description of how to set nonblocking mode.

When the socket is not ready to accept data and the process is trying to write data to the socket:

- Unless `FNDELAY` or `O_NONBLOCK` is set, `writev()` blocks until the socket is ready to accept data.
- If `FNDELAY` is set, `writev()` returns a `0`.

- If `O_NONBLOCK` is set, `writev()` does not block the process. If some data can be written without blocking the process, `writev()` writes what it can and returns the number of bytes written. Otherwise, it sets the error code to `EAGAIN` and returns a `-1`.

For datagram sockets, this call sends the entire datagram, provided that the datagram fits into the TCP/IP buffers. Stream sockets act like streams of information with no boundaries separating data. For example, if an application program wishes to send 1000 bytes, each call to this function can send 1 byte, or 10 bytes, or the entire 1000 bytes. Therefore, application programs using stream sockets should place this call in a loop, calling this function until all data has been sent.

### Special Behavior for C++

To use this function with C++, you must use the `_XOPEN_SOURCE_EXTENDED 1` feature test macro.

### Returned Value

If successful, the number of bytes written from the buffer is returned. The value `-1` indicates an error. The value of the error code indicates the specific error.

A value of 0 or greater indicates the number of bytes sent, however, this does not assure that data delivery was complete. A connection can be dropped by a peer socket and a `SIGPIPE` signal generated at a later time if data delivery is not complete.

### Error Code Description

<code>EAGAIN</code>	Resources temporarily unavailable. Subsequent requests may complete normally.
<code>EBADF</code>	<i>fs</i> is not a valid file or socket descriptor.
<code>ECONNRESET</code>	A connection was forcibly closed by a peer.
<code>EDESTADDRREQ</code>	The socket is not connection-oriented and no peer address is set.
<code>EFAULT</code>	Using the <i>iov</i> and <i>iovcnt</i> parameters would result in an attempt to access storage outside the caller's address space.
<code>EINTR</code>	A signal interrupted <code>writev()</code> before any data was transmitted.
<code>EINVAL</code>	An incorrect value for <i>iovcnt</i> was detected.
<code>EMSGSIZE</code>	The message was too big to be sent as a single datagram.
<code>ENOBUFS</code>	Buffer space is not available to send the message.
<code>ENOTCONN</code>	The socket is not connected.
<code>EPIPE</code>	For a connected stream socket the connection to the peer socket has been lost. A <code>SIGPIPE</code> signal is sent to the calling process.
<code>EPROTOTYPE</code>	The protocol is the wrong type for this socket. A <code>SIGPIPE</code> signal is sent to the calling process.
<code>EWouldBlock</code>	The socket is in nonblocking mode and data buffers are not available.



## Related Information

- “connect() — Connect a Socket” on page 214
- “fcntl() — Control Open File Descriptors” on page 350
- “getsockopt() — Get the Options Associated with a Socket” on page 606
- “ioctl() — Control Device” on page 672
- “read() — Read From a File or Socket” on page 1080
- “readv() — Read Data on a File or Socket and Store in a Set of Buffers” on page 1093
- “recv() — Receive Data on a Socket” on page 1103
- “recvfrom() — Receive Messages on a Socket” on page 1106
- “recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers” on page 1110
- “select() — Monitor Activity on Files/Sockets and Message Queues” on page 1159
- “selectex() — Monitor Activity on Files/Sockets and Message Queues” on page 1166
- “send() — Send Data on a Socket” on page 1178
- “sendmsg() — Send Messages on a Socket” on page 1185
- “sendto() — Send Data on a Socket” on page 1190
- “setsockopt() — Set Options Associated with a Socket” on page 1270
- “socket() — Create a Socket” on page 1371
- “write() — Write Data on a File or Socket” on page 1780

## \_\_wsinit() — Reinitialize Writeable Static

### Standards

Standards / Extensions	C or C++	Dependencies
C Library	both	

### Format

```
#include <unistd.h>
```

```
int __wsinit( void (*func_ptr)() );
```

### General Description

The `__wsinit()` function reinitializes the writeable static area of a module that was loaded by the `fetch()` library call. `func_ptr` must be a valid fetch pointer returned by `fetch()` or `fetchep()`. If the module contains C++, `__wsinit()` first runs any C++ static destructors, then `__wsinit()` runs the static constructors that are present in the load module.

Program variables with the static or extern storage class and writeable strings receive the initial value defined in the program, if any initial value was assigned. The C header files contain external variable declarations for those variables defined by the POSIX, XPG4 and XPG4.2 standards. If the fetched module contains these variables, `__wsinit()` reinitializes them as described in the *OS/390 C/C++ Programming Guide*.

### Returned Value

The `__wsinit()` function returns 0 if successful. Otherwise, it returns -1 and sets `errno` to indicate the error. The following are the possible values of `errno`:

`EINVAL` `func_ptr` is not a valid fetch pointer.

### Related Information

- “unistd.h” on page 53
- “fetch() — Get a Load Module” on page 369

## w\_statfs() — Get the File System Status

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 4.3

### Format

```
#include <sys/statfs.h>
```

```
int w_statfs(const char *filesystem, struct w_statfs *statbuf, size_t length);
```

### General Description

Gets status about a specific file system.

*filesystem* The name of the file system for which status is being retrieved. This file system name can be one of the following:

- The name specified in the FILESYSTEM parameter of the ROOT or MOUNT statements in the BPXPRMxx parmlib member.
- The name specified in a TSO/E MOUNT command.
- The name returned on a previous call to w\_getmntent().

*statbuf* A buffer that the status information is put into. The status information is mapped by the sys/statfs.h header file.

int statfs\_len

Length of *statfs*

int statfs\_blksize

Block size

unsigned int statfs\_total\_space

Total space in block size units

unsigned int statfs\_used\_space

Allocated space in block size units

unsigned int statfs\_free\_space

Space available to unprivileged users in block size units

*length*

The length of the buffer. The length of the buffer and the length of the structure are compared, and the shorter of the two is used to determine how much information to return in the buffer.

If the buffer length is zero, only the return value is returned. A process can use a length of zero to detect if a file system exists or not.

### Special Behavior for XPG4.2:

w\_statfs() is replaced by w\_statvfs().

## Returned Value

If successful, `w_statfs()` returns the length of the data in the buffer. If unsuccessful, it returns the value `-1` and sets `errno` to `EINVAL`, which indicates that a parameter was incorrectly specified

## Example

### CBC3BW36

```
/* CBC3BW36 */
#include <sys/statfs.h>
#include <stdio.h>

main() {
    char fs[]="POSIX.ROOT.FS";
    struct w_statfs buf;

    if (w_statfs(fs, &buf, sizeof(buf)) == -1)
        perror("w_statfs() error");
    else {
        printf("each block in %s is %d bytes big\n", fs,
            buf.statfs_blksize);
        printf("there are %d blocks in use out of a total of %d\n",
            buf.statfs_used_space, buf.statfs_total_space);
        printf("in bytes, that's %.0f bytes used out of a total of %.0f\n",
            ((double)buf.statfs_used_space * buf.statfs_blksize),
            ((double)buf.statfs_total_space * buf.statfs_blksize));
    }
}
```

## Output

```
each block in POSIX.ROOT.FS is 4096 bytes big
there are 2089 blocks in use out of a total of 2400
in bytes, that's 8556544 bytes used out of a total of 9830400
```

## Related Information

- “`sys/statfs.h`” on page 49
- “`mount()` — Make a File System Available” on page 838
- “`w_getmntent()` — Get Information on Mounted File Systems” on page 1756

## w\_statvfs() — Get the File System Status

### Standards

Standards / Extensions	C or C++	Dependencies
OS/390 UNIX	both	MVS 5.2.2

### Format

```
#define _OPEN_SOURCE 2
#include <sys/statvfs.h>
```

```
int w_statvfs(const char *filesystem, struct statvfs *buffer, size_t buflen);
```

### General Description

The `w_statvfs()` function gets status about a specific file system.

*filesystem* The name of the file system for which status is being retrieved. This file system name can be one of the following:

- The name specified in the FILESYSTEM parameter of the ROOT or MOUNT statements in the BPXPRMxx parmlib member.
- The name specified in a TSO/E MOUNT command.
- The name returned on a previous call to `w_getmntent()`.

*buffer* A buffer that the status information is put into. The information is returned in a `statvfs` structure, as defined in the `sys/statvfs.h` header file. The elements of this structure are described in “`statvfs()` — Get File System Information” on page 1408.

*buflen* The length of the buffer. The length of the buffer and the length of the structure are compared, and the shorter of the two is used to determine how much information to return in the buffer.

If the buffer length is zero, only the return value is returned. A process can use a length of zero to detect if a file system exists or not.

### Returned Value

If successful, `w_statvfs()` returns the length of the data in the buffer.

If unsuccessful, `w_statvfs()` returns `-1`, and returns the error value in `errno`. The following are the possible values of `errno`:

**EINVAL** A parameter was specified incorrectly. For example, the file system name (*filesystem*) was not found.

### Example

```
#define _OPEN_SOURCE 2
#include <sys/statvfs.h>
#include <stdio.h>

main() {
    char fs[]="POSIX.ROOT.FS";
    struct statvfs buf;

    if (w_statvfs(fs, &buf, sizeof(buf)) == -1)
        perror("w_statvfs() error");
    else {
```

```
    printf("each block in %s is %d bytes big\n", fs,
           buf.f_bsize);
    printf("there are %d blocks available out of a total of %d\n",
           buf.f_bavail, buf.f_blocks);
    printf("in bytes, that's %.0f bytes free out of a total of %.0f\n",
           ((double)buf.f_bavail * buf.f_bsize),
           ((double)buf.f_blocks * buf.f_bsize));
}
}
```

### Output

```
each block in POSIX.ROOT.FS is 4096 bytes big
there are 2089 blocks available out of a total of 2400
in bytes, that's 8556544 bytes free out of a total of 9830400
```

### Related Information

- “sys/statvfs.h” on page 49
- “fstatvfs() — Get File System Information” on page 481
- “mount() — Make a File System Available” on page 838
- “statvfs() — Get File System Information” on page 1408
- “w\_getmntent() — Get Information on Mounted File Systems” on page 1756

**y0() - y1() - yn() — Bessel Functions of the Second Kind**

Standards / Extensions	C or C++	Dependencies
SAA XPG4 XPG4.2	both	

**Format**

```
#include <math.h>
```

```
double y0(double x);
```

```
double y1(double x);
```

```
double yn(int n, double x);
```

**Compiler Option**

LANGLVL(SAA), LANGLVL(SAA2), or LANGLVL(EXTENDED)

**General Description**

Bessel functions are solutions to certain types of differential equations.

The y0(), y1(), and yn() functions are Bessel functions of the *second kind*, for orders 0, 1, and *n*, respectively. The argument *x* must be positive. The argument *n* should be greater than or equal to zero. If *n* is less than zero, there will be a negative exponent in the result.

**Returned Value**

For y0(), y1(), or yn(), if *x* is negative, the function sets errno to EDOM and returns the value -HUGEVAL. For y0(), y1(), or yn(), if *x* causes overflow, the function sets errno to ERANGE and returns the value -HUGEVAL.

The calculated value is returned otherwise.

**Example****CBC3BY01**

```
/* CBC3BY01
   This example computes y to be the order 0 Bessel function of the first
   kind for x and z to be the order 3 Bessel function of the second kind for x.
*/
#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, y, z;
    x = 4.27;

    y = y0(x);      /* y = -0.3660 is the order 0 bessel */
                   /* function of the first kind for x */
    z = yn(3,x);    /* z = -0.0875 is the order 3 bessel */
                   /* function of the second kind for x */
    printf("x = %f\n y = %f\n z = %f\n", x, y, z);
}
```

### Related Information

- “math.h” on page 35
- “erf() - erfc() — Calculate Error and Complementary Error Functions” on page 316
- “gamma() — Calculate Gamma Function” on page 497
- “j0() - j1() - jn() — Bessel Functions of the First Kind” on page 726



## Library Functions for the System Programming C Facilities

The library functions specific to the System Programming C environment are described in the *OS/390 C/C++ Programming Guide*. These system programming functions are as follows:

- `__xhotc()`
- `__xhotl()`
- `__xhott()`
- `__xhotu()`
- `__xregs()`
- `__xsacc()`
- `__xsrv()`
- `__xusr()`
- `__xusr2()`
- `__24malc()`
- `__4kmalc()`



## Appendix A. C/C++ Macros

You can use the macros listed here to write programs that use built-in services of the OS/390 C/C++ product. The general purpose macros, labelled *General*, are pre-defined macros that are either ANSI-standard macros or extensions to the Systems Application Architecture (SAA) definition. The internal-use-only macros, labelled *Internal*, are developed by IBM to provide you with additional OS/390 C/C++ functionality. Internal macros are defined in corresponding include files.

Table 46 (Page 1 of 2). C/C++ Macros: A - E

Macro	Include File	General or Internal
ABDAY_1	nl_langinfo.h	General
ABDAY_2	nl_langinfo.h	General
ABDAY_3	nl_langinfo.h	General
ABDAY_4	nl_langinfo.h	General
ABDAY_5	nl_langinfo.h	General
ABDAY_6	nl_langinfo.h	General
ABDAY_7	nl_langinfo.h	General
__abendcode	signal.h	General
ABMON_1	nl_langinfo.h	General
ABMON_10	nl_langinfo.h	General
ABMON_11	nl_langinfo.h	General
ABMON_12	nl_langinfo.h	General
ABMON_2	nl_langinfo.h	General
ABMON_3	nl_langinfo.h	General
ABMON_4	nl_langinfo.h	General
ABMON_5	nl_langinfo.h	General
ABMON_6	nl_langinfo.h	General
ABMON_7	nl_langinfo.h	General
ABMON_8	nl_langinfo.h	General
ABMON_9	nl_langinfo.h	General
__abnd_i	signal.h	General
__ABS(x)	math.h	Internal
abs(x)	stdlib.h	General
acos(x)	math.h	General
AM_STR	nl_langinfo.h	General
__amrc	stdio.h	General
__amrc_i	stdio.h	Internal
__APPEND	stdio.h	General
asin(x)	math.h	General
__assert	assert.h	Internal
assert(expr)	assert.h	General
assert(ignore)	assert.h	General
atan(x)	math.h	General
atan2(x,y)	math.h	General
__B	dynit.h	General
__B_	dynit.h	General
__BINARY	stdio.h	General
__BSAM_CLOSE	stdio.h	General
__BSAM_CLOSE_T	stdio.h	General
__BSAM_NOTE	stdio.h	General
__BSAM_OPEN	stdio.h	General
__BSAM_POINT	stdio.h	General
__BSAM_READ	stdio.h	General
__BSAM_WRITE	stdio.h	General
BUFSIZ	stdio.h	General
cdump(x)	ctest.h	General
__CELMSGF_WRITE	stdio.h	General
CHAR_BIT	limits.h	General

Table 46 (Page 1 of 2). C/C++ Macros: A - E

Macro	Include File	General or Internal
CHAR_MAX	limits.h	General
CHAR_MIN	limits.h	General
__cics	cics.h	Internal
__cics_CD	cics.h	Internal
CLK_TCK	time.h	General
CLOCKS_PER_SEC	time.h	General
__CLOSE	dynit.h	General
clrmemf(x)	stdio.h	General
__CMS_CLOSE	stdio.h	General
__CMS_OPEN	stdio.h	General
__CMS_READ	stdio.h	General
__CMS_STATE	stdio.h	General
__CMS_WRITE	stdio.h	General
__cntrl(c)	stdio.h	Internal
CODESET	nl_langinfo.h	General
__CONTIG	dynit.h	General
cos(x)	math.h	General
cosh(x)	math.h	General
CRNCYSTR	nl_langinfo.h	General
csnap(x)	ctest.h	General
ctdli	ims.h	General
ctest(x)	ctest.h	General
__ctest__	ctest.h	Internal
ctime(t)	time.h	General
ctrace(x)	ctest.h	General
__ctype	ctype.h	Internal
__ctype_i	ctype.h	Internal
__ctypesec	ctype.h	Internal
__ctypesec_i	ctype.h	Internal
__CURR	stdarg.h	Internal
__CURRENT	stdio.h	General
__CURRENT_LOWER	stdio.h	General
__cust_def	ctype.h	Internal
__CYL	dynit.h	General
__D_	dynit.h	General
D_FMT	nl_langinfo.h	General
D_T_FMT	nl_langinfo.h	General
DAY_1	nl_langinfo.h	General
DAY_2	nl_langinfo.h	General
DAY_3	nl_langinfo.h	General
DAY_4	nl_langinfo.h	General
DAY_5	nl_langinfo.h	General
DAY_6	nl_langinfo.h	General
DAY_7	nl_langinfo.h	General
DBL_DIG	float.h	General
__dbl_eps	float.h	Internal
DBL_EPSILON	float.h	General
__dbl_flts_i	float.h	Internal
DBL_MANT_DIG	float.h	General
DBL_MAX	float.h	General
DBL_MAX_EXP	float.h	General
DBL_MAX_10_EXP	float.h	General
DBL_MIN	float.h	General
DBL_MIN_EXP	float.h	General
DBL_MIN_10_EXP	float.h	General
__DEF_CLASS	dynit.h	General
difftime(t1, t0)	time.h	General
__DISK	dynit.h	General
__DISP_CATLG	dynit.h	General

Table 46 (Page 2 of 2). C/C++ Macros: A - E

Macro	Include File	General or Internal
__DISP_DELETE	dynit.h	General
__DISP_KEEP	dynit.h	General
__DISP_MOD	dynit.h	General
__DISP_NEW	dynit.h	General
__DISP_OLD	dynit.h	General
__DISP_SHR	dynit.h	General
__DISP_UNCATLG	dynit.h	General
DOMAIN	math.h	General
__DSORG_DA	dynit.h	General
__DSORG_DAU	dynit.h	General
__DSORG_GS	dynit.h	General
__DSORG_IS	dynit.h	General
__DSORG_ISU	dynit.h	General
__DSORG_PO	dynit.h	General
__DSORG_POU	dynit.h	General
__DSORG_PS	dynit.h	General
__DSORG_PSU	dynit.h	General
__DSORG_unknown	dynit.h	General
__DSORG_VSAM	dynit.h	General
__DUMMY	stdio.h	General
__DUMMY_DSN	dynit.h	General
dynalloc(x)	dynit.h	General
dynfree(x)	dynit.h	General
dyninit(__dynp)	dynit.h	General
__dynit	dynit.h	Internal
EACTIVE	mtf.h	General
EAUTOALC	mtf.h	General
EBADLNKG	mtf.h	General
EDOM	errno.h	General
EENTRY	mtf.h	General
EINACTIVE	mtf.h	General
EMODFIND	mtf.h	General
EMODFMT	mtf.h	General
EMODREAD	mtf.h	General
ENAME2LNG	mtf.h	General
ENOMEM	mtf.h	General
EOF	stdio.h	General
ERANGE	errno.h	General
erf(x)	math.h	General
erfc(x)	math.h	General
errno	errno.h	General
__errno_a	errno.h	Internal
__errno_i	errno.h	Internal
__errnoflg	errno.h	Internal
__errnoh	errno.h	Internal
__ESDS	stdio.h	General
__ESDS_PATH	stdio.h	General
ESUBCALL	mtf.h	General
ETASKABND	mtf.h	General
ETASKFAIL	mtf.h	General
ETASKID	mtf.h	General
ETASKNUM	mtf.h	General
EWRONGOS	mtf.h	General
EXIT_FAILURE	stdlib.h	General
EXIT_SUCCESS	stdlib.h	General
exp(x)	math.h	General

Table 47 (Page 1 of 3). C/C++ Macros: F - M

Macro	Include File	General or Internal
__F	dynit.h	General
__F_	dynit.h	General

Table 47 (Page 1 of 3). C/C++ Macros: F - M

Macro	Include File	General or Internal
fabs(x)	math.h	General
__FB	dynit.h	General
__FB_	dynit.h	General
__FBS	dynit.h	General
__FBS_	dynit.h	General
fdelrec(x)	stdio.h	General
fetch(x)	stdlib.h	General
fetchep(x)	stdlib.h	General
__FILE	assert.h	Internal
FILENAME_MAX	stdio.h	General
fldata(x,y,z)	stdio.h	General
__float	float.h	Internal
flocate(w,x,y,z)	stdio.h	General
FLT_DIG	float.h	General
__flt_eps	float.h	Internal
__flt_eps_i	float.h	Internal
FLT_EPSILON	float.h	General
FLT_MANT_DIG	float.h	General
FLT_MAX	float.h	General
FLT_MAX_EXP	float.h	General
FLT_MAX_10_EXP	float.h	General
FLT_MIN	float.h	General
FLT_MIN_EXP	float.h	General
FLT_MIN_10_EXP	float.h	General
FLT_RADIX	float.h	General
FLT_ROUNDS	float.h	General
FOPEN_MAX	stdio.h	General
fupdate(x,y,z)	stdio.h	General
gamma(x)	math.h	General
getc(p)	stdio.h	General
__getc(p)	stdio.h	Internal
getchar(c)	stdio.h	General
__gtab(x)	stdio.h	Internal
__gtca()	assert.h	Internal
__HFS	stdio.h	General
__HIPERSPACE	stdio.h	General
__HOLDQ	dynit.h	General
__HSP_CREATE	stdio.h	General
__HSP_DELETE	stdio.h	General
__HSP_EXTEND	stdio.h	General
__HSP_READ	stdio.h	General
__HSP_WRITE	stdio.h	General
HUGE	math.h	General
HUGE_VAL	math.h	General
iconv	iconv.h	General
__ims	ims.h	Internal
__imspcb_a	ims.h	Internal
INT_MAX	limits.h	General
INT_MIN	limits.h	General
__INTERCEPT_READ	stdio.h	General
__INTERCEPT_WRITE	stdio.h	General
__IO_DEVTYPE	stdio.h	General
__IO_INIT	stdio.h	General
__IO_RDJFCB	stdio.h	General
__IOFBF	stdio.h	General
__IOLBF	stdio.h	General
__ISALNUM	ctype. and wchar.h	Internal
isalnum(c)	ctype.h	General
__isalnum(c)	ctype.h	Internal
__ISALPHA	ctype. and wchar.h	Internal
isalpha(c)	ctype.h	General

Table 47 (Page 2 of 3). C/C++ Macros: F - M

Macro	Include File	General or Internal
__isalpha(c)	ctype.h	Internal
isblank	ctype.h	General
__ISBLANK	ctype. and wchar.h	Internal
iscics	cics.h	General
__ISCNTRL	ctype. and wchar.h	Internal
iscntrl(c)	ctype.h	General
__iscntrl(c)	ctype.h	Internal
__ISDIGIT	ctype. and wchar.h	Internal
isdigit(c)	ctype.h	General
__isdigit(c)	ctype.h	Internal
__ISGRAPH	ctype. and wchar.h	Internal
isgraph(c)	ctype.h	General
__isgraph(c)	ctype.h	Internal
islower(c)	ctype.h	General
__islower(c)	ctype.h	Internal
__ISPRINT	ctype. and wchar.h	Internal
isprint(c)	ctype.h	General
__isprint(c)	ctype.h	Internal
__ISPUNCT	ctype. and wchar.h	Internal
ispunct(c)	ctype.h	General
__ispunct(c)	ctype.h	Internal
__ISSPACE	ctype. and wchar.h	Internal
isspace(c)	ctype.h	General
__isspace(c)	ctype.h	Internal
__ISUPPER	ctype. and wchar.h	Internal
isupper(c)	ctype.h	General
__isupper(c)	ctype.h	Internal
iswalnum	wchar.h	General
iswalpha	wchar.h	General
iswcntrl	wchar.h	General
iswctype	wchar.h	General
iswdigit	wchar.h	General
iswgraph	wchar.h	General
iswlower	wchar.h	General
iswprint	wchar.h	General
iswpunct	wchar.h	General
iswspace	wchar.h	General
iswupper	wchar.h	General
iswxdigit	wchar.h	General
isxdigit(c)	ctype.h	General
__isxdigit(c)	ctype.h	Internal
__KEY_EQ	stdio.h	General
__KEY_EQ_BWD	stdio.h	General
__KEY_FIRST	stdio.h	General
__KEY_GE	stdio.h	General
__KEY_LAST	stdio.h	General
__KSDS	stdio.h	General
__KSDS_PATH	stdio.h	General
L_tmpnam	stdio.h	General
LC_ALL	locale.h	General
LC_C	locale.h	General
LC_C_FRANCE	locale.h	General
LC_C_GERMANY	locale.h	General
LC_C_ITALY	locale.h	General
LC_C_SPAIN	locale.h	General

Table 47 (Page 2 of 3). C/C++ Macros: F - M

Macro	Include File	General or Internal
LC_C_UK	locale.h	General
LC_C_USA	locale.h	General
LC_COLLATE	locale.h	General
LC_CTYPE	locale.h	General
LC_MONETARY	locale.h	General
LC_NUMERIC	locale.h	General
LC_SYNTAX	locale.h	General
LC_TIME	locale.h	General
LC_TOD	locale.h	General
LDBL_DIG	float.h	General
__ldbl_eps	float.h	Internal
LDBL_EPSILON	float.h	General
__ldbl_flts_i	float.h	Internal
LDBL_MANT_DIG	float.h	General
LDBL_MAX	float.h	General
LDBL_MAX_EXP	float.h	General
LDBL_MAX_10_EXP	float.h	General
LDBL_MIN	float.h	General
LDBL_MIN_EXP	float.h	General
LDBL_MIN_10_EXP	float.h	General
LEAWI_INCLUDED	leawi.h	Internal
__limits	limits.h	Internal
__locale	locale.h	Internal
log(x)	math.h	General
log10(x)	math.h	General
__LOWER	stdio.h	General
__M_	dynit.h	General
M_E	math.h	General
M_LN10	math.h	General
M_LN2E	math.h	General
M_LOG	math.h	General
M_LOG10E	math.h	General
M_PI	math.h	General
M_PI_2	math.h	General
M_PI_4	math.h	General
M_SQRT1_2	math.h	General
M_SQRT2	math.h	General
M_1_PI	math.h	General
M_2_PI	math.h	General
M_2_SQRTPI	math.h	General
M_2PI	math.h	General
__math	math.h	Internal
__max_flt	math.h	Internal
__max_flts_i	float.h	Internal
MAXTASK	mtf.h	General
MB_CUR_MAX	stdlib.h	General
MB_LEN_MAX	limits.h	General
memchr(x,y,z)	string.h	General
memcmp(x,y,z)	string.h	General
memcpy(x,y,z)	string.h	General
memset(x,y,z)	string.h	General
__min_flts	float.h	Internal
__min_flts_i	float.h	Internal
MON_1	nl_langinfo.h	General
MON_2	nl_langinfo.h	General
MON_3	nl_langinfo.h	General
MON_4	nl_langinfo.h	General
MON_5	nl_langinfo.h	General
MON_6	nl_langinfo.h	General
MON_7	nl_langinfo.h	General
MON_8	nl_langinfo.h	General
MON_9	nl_langinfo.h	General
MON_10	nl_langinfo.h	General

Table 47 (Page 3 of 3). C/C++ Macros: F - M

Macro	Include File	General or Internal
MON_11	nl_langinfo.h	General
MON_12	nl_langinfo.h	General
__MSGFILE	stdio.h	General
MTF_ALL	mtf.h	General
MTF_ANY	mtf.h	General
MTF_OK	mtf.h	General
__mtfh	mtf.h	Internal

Table 48 (Page 1 of 2). C/C++ Macros: N - Y

Macro	Include File	General or Internal
__NEXT	stdarg.h	Internal
__nextword(base)	stdarg.h	Internal
__NL_NUM_ITEMS	nl_langinfo.h	Internal
NOEXPR	nl_langinfo.h	General
__NOTVSAM	stdio.h	General
NULL	stddef.h	General
offsetof(x,y)	stddef.h	General
OMIT_FC	leawi.h	General
__osplist	stdlib.h	General
__OTHER	stdio.h	General
OVERFLOW	math.h	General
PCB_STRUCT(key_len)	ims.h	General
__pcblist	ims.h	General
__PCBLIST_INDEX	ims.h	Internal
__PERM	dynit.h	General
PLOSS	math.h	General
PM_STR	nl_langinfo.h	General
pow(x,y)	math.h	General
__PRINTER	dynit.h	General
__psizeof(type)	stdarg.h	Internal
putc(c, p)	stdio.h	General
__putc(c, p)	stdio.h	Internal
putchar(c)	stdio.h	General
RADIXCHAR	nl_langinfo.h	General
RAND_MAX	stdlib.h	General
__RBA_EQ	stdio.h	General
__RBA_EQ_BWD	stdio.h	General
__READ	stdio.h	General
__RECORD	stdio.h	General
REG_BADBR	regex.h	General
REG_BADPT	regex.h	General
REG_BADRPT	regex.h	General
REG_EBOL	regex.h	General
REG_EBRACE	regex.h	General
REG_EBRACK	regex.h	General
REG_ECHAR	regex.h	General
REG_ECOLLATE	regex.h	General
REG_ECTYPE	regex.h	General
REG_EEOL	regex.h	General
REG_EESCAPE	regex.h	General
REG_EPAREN	regex.h	General
REG_ERANGE	regex.h	General
REG_ESPACE	regex.h	General
REG_NOMATCH	regex.h	General
__RELEASE	dynit.h	General
release(x)	stdlib.h	General
__ROUND	dynit.h	General
__RRDS	stdio.h	General
__rsn_i	signal.h	Internal
__rsncode	signal.h	General
__R1	stdlib.h	General

Table 48 (Page 1 of 2). C/C++ Macros: N - Y

Macro	Include File	General or Internal
__S_	dynit.h	General
SCHAR_MAX	limits.h	General
SCHAR_MIN	limits.h	General
SEEK_CUR	stdio.h	General
SEEK_END	stdio.h	General
SEEK_SET	stdio.h	General
__setjmp	setjmp.h	Internal
setjmp(x)	setjmp.h	General
SHRT_MAX	limits.h	General
SHRT_MIN	limits.h	General
SIG_DFL	signal.h	General
SIG_ERR	signal.h	General
SIG_IGN	signal.h	General
SIG_PROMOTE	signal.h	General
SIGABND	signal.h	General
SIGABRT	signal.h	General
SIGFPE	signal.h	General
SIGILL	signal.h	General
SIGINT	signal.h	General
__signal	signal.h	Internal
SIGSEGV	signal.h	General
SIGTERM	signal.h	General
SIGUSR1	signal.h	General
SIGUSR2	signal.h	General
sin(x)	math.h	General
SING	math.h	General
sinh(x)	math.h	General
__size_t	time.h	Internal
__spc	spc.h	Internal
sqrt(x)	math.h	General
__stdarg	stdarg.h	Internal
__stddef	stddef.h	Internal
stderr	assert.h	General
__stderr_i	assert.h	Internal
stdin	stdio.h	General
__stdin_i	stdio.h	Internal
__stdio	stdio.h	Internal
__stdlib	stdlib.h	Internal
stdout	stdio.h	General
__stdout_i	stdio.h	Internal
strcat(x,y)	string.h	General
strchr(x,y)	string.h	General
strcmp(x,y)	string.h	General
strcpy(x,y)	string.h	General
__string	string.h	Internal
strlen(x)	string.h	General
strrchr(x,y)	string.h	General
__SVC99	stdio.h	Internal
svc99(x)	stdio.h	General
__SVC99_ALLOC	stdio.h	General
__SVC99_ALLOC_NEW	stdio.h	General
__sysplist	stdlib.h	General
__sysplist_i	stdlib.h	Internal
__SYSPLIST_INDEX	stdlib.h	Internal
T_FMT	nl_langinfo.h	General
T_FMT_AMPM	nl_langinfo.h	General
tan(x)	math.h	General
tanh(x)	math.h	General
__TAPE	stdio.h	General
__TDQ	stdio.h	General
__temp	ctype.h	Internal
__temp_a	stdio.h	Internal
__temp_i	ctype.h	Internal

Table 48 (Page 2 of 2). C/C++ Macros: N - Y

Macro	Include File	General or Internal
__TERM	dynit.h	General
__TERMINAL	stdio.h	General
__TEXT	stdio.h	General
__TGET_READ	stdio.h	General
THOUSEP	nl_langinfo.h	General
__time	time.h	Internal
tinit(x,y)	mtf.h	General
TLOSS	math.h	General
TMP_MAX	stdio.h	General
tolower(c)	ctype.h	General
__tolower(c)	ctype.h	Internal
__TOLOWER_INDEX	ctype.h	Internal
toupper(c)	ctype.h	General
__toupper(c)	ctype.h	Internal
__TOUPPER_INDEX	ctype.h	Internal
towlower	wchar.h	General
towupper	wchar.h	General
__TPUT_WRITE	stdio.h	General
__TRK	dynit.h	General
tsched	mtf.h	General
tsetsubt(x,y)	mtf.h	General
tsyncro(x)	mtf.h	General
tterm(x)	mtf.h	General
__U_	dynit.h	General
UCHAR_MAX	limits.h	General
UINT_MAX	limits.h	General
ULONG_MAX	limits.h	General
UNDERFLOW	math.h	General
__UNKN_ERROR	stdio.h	General
__UPDATE	stdio.h	General
USHRT_MAX	limits.h	General
__V	dynit.h	General
__V_	dynit.h	General
va_arg(ap, type)	stdarg.h	General
va_end(ap)	stdarg.h	General
va_list	stdarg.h	General
va_start(ap, arg)	stdarg.h	General
__valist	stdarg.h	Internal
__valist_	stdio.h	Internal
__VB	dynit.h	General
__VB_	dynit.h	General
__VBS	dynit.h	General
__VBS_	dynit.h	General
__VSAM_CLOSE	stdio.h	General
__VSAM_ENDREQ	stdio.h	General
__VSAM_ERASE	stdio.h	General
__VSAM_GENCB	stdio.h	General
__VSAM_GET	stdio.h	General
__VSAM_MODCB	stdio.h	General
__VSAM_OPEN_ESDS	stdio.h	General
__VSAM_OPEN_ESDS_PAS	stdio.h	General
__VSAM_OPEN_FAIL	stdio.h	General
__VSAM_OPEN_KSDS	stdio.h	General
__VSAM_OPEN_KSDS_PAS	stdio.h	General
__VSAM_OPEN_RRDS	stdio.h	General
__VSAM_POINT	stdio.h	General
__VSAM_PUT	stdio.h	General
__VSAM_SHOWCB	stdio.h	General
__VSAM_TESTCB	stdio.h	General
__wchar_t	stddef.h	Internal
__wcstr	wcstr.h	Internal
WEOF	wchar.h	General
__WRITE	stdio.h	General

Table 48 (Page 2 of 2). C/C++ Macros: N - Y

Macro	Include File	General or Internal
YESEXPR	nl_langinfo.h	General





## Appendix B. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing  
IBM Corporation  
North Castle Drive  
Armonk, NY 10504-1785  
USA

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation  
Licensing  
2-31 Roppongi 3-chome, Minato-ku  
Tokyo 106, Japan

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or

the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation  
Mail Station P300  
522 South Road  
Poughkeepsie, NY 12601-5400  
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

| All statements regarding IBM's future direction or intent  
| are subject to change without notice, and represent  
| goals and objectives only.

| This information contains examples of data and reports  
| used in daily business operations. To illustrate them as  
| completely as possible, the examples include the  
| names of individuals, companies, brands, and products.  
| All of these names are fictitious and any similarity to the  
| names and addresses used by an actual business  
| enterprise is entirely coincidental.

| COPYRIGHT LICENSE:

| This information contains sample application programs  
| in source language, which illustrates programming tech-  
| niques on various operating platforms. You may copy,  
| modify, and distribute these sample programs in any  
| form without payment to IBM, for the purposes of devel-  
| oping, using, marketing or distributing application pro-  
| grams conforming to the application programming

| interface for the operating platform for which the sample  
| programs are written. These examples have not been  
| thoroughly tested under all conditions. IBM, therefore,  
| cannot guarantee or imply reliability, serviceability, or  
| function of these programs. You may copy, modify, and  
| distribute these sample programs in any form without  
| payment to IBM for the purposes of developing, using,  
| marketing, or distributing application programs con-  
| forming to IBM's application programming interfaces.

| If you are viewing this information softcopy, the photo-  
| graphs and color illustrations may not appear.

---

## Programming Interface Information

This book documents intended Programming Interfaces  
that allow the customer to write programs to obtain the  
services of IBM OS/390 C/C++ and IBM Language Envi-  
ronment in OS/390.

---

## Standards

Extracts are reprinted from *IEEE Std 1003.1—1990, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from *IEEE P1003.1a Draft 6 July 1991, Draft Revision to Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*, copyright 1992 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from *IEEE Std 1003.2—1992, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shells and Utilities*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from *IEEE Std P1003.4a/D6—1992, IEEE Draft Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 2: Threads Extension [C language]*,

copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts from *ISO/IEC 9899:1990* have been reproduced with the permission of the International Organization for Standardization, ISO, and the International Electrotechnical Commission, IEC. The complete standard can be obtained from any ISO or IEC member or from the ISO or IEC Central Offices, Case Postal, 1211 Geneva 20, Switzerland. Copyright remains with ISO and IEC.

Portions of this book are extracted from *X/Open Specification, Programming Languages, Issue 3* copyright 1988, 1989, February 1992, by the X/Open Company Limited, with the permission of X/Open Company Limited. No further reproduction of this material is permitted without the written notice from the X/Open Company Ltd, UK.

Portions of this book are extracted from *X/Open CAE Specification, System Interfaces and Headers, Issue 4, Version 2* copyright September 1994, by the X/Open Company Limited, with the permission of X/Open Company Limited. No further reproduction of this material is permitted without the written notice from the X/Open Company Ltd, UK.

---

## Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AIX	BookManager
CICS	CICS/ESA
C/MVS	C++/MVS
C/370	DATABASE 2
DB2	DFSMS/MVS
ESA/390	GDDM
Hiperspace	IBM
IBMLink	IMS
IMS/ESA	Language Environment
MVS	MVS/ESA
Open Class	Open Edition
Operating System/2	Operating System/400
OS/2	OS/390
OS/400	QMF
RACF	RISC System/6000
SAA	SOM
SOMobjects	SP
System Object Model	Systems Application Architecture
System/370	S/370
S/390	VisualAge
VM/ESA	VSE/ESA
3090	

IEEE is a trademark in the United States and other countries of the Institute of Electrical and Electronics Engineers, Inc.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States.

Microsoft, Windows, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

Other company, product or service names may be trademarks or service marks of others.

ANSI	American National Standards Institute
ISO	International Organization for Standardization
OSF	Open Software Foundation Inc.
POSIX	Institute of Electrical and Electronic Engineers
X/Open	X/Open Company Ltd.
XPG3	X/Open Company Ltd.
XPG4	X/Open Company Ltd.
XTI	X/Open Company Ltd.

---

# Glossary

This glossary defines terms and abbreviations that are used in this book. Included are terms and definitions from the following sources:

- *American National Standard Dictionary for Information Systems*, ANSI/ISO X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI/ISO). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018. Such definitions are indicated by the symbol *ANSI/ISO* after the definition.
- *IBM Dictionary of Computing*, SC20-1699. These definitions are indicated by the trademark *IBM* after the definition.
- *X/Open CAE Specification. Commands and Utilities, Issue 4. July, 1992.* These definitions are indicated by the symbol *X/Open* after the definition.
- *ISO/IEC 9945-1:1990/IEEE POSIX 1003.1-1990* These definitions are indicated by the symbol *ISO.1* after the definition.
- *The Information Technology Vocabulary*, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol *ISO-JTC1* after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol *ISO Draft* after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

---

## A

**abstract code unit.** See *ACU*.

**abstraction (data).** A data type with a private representation and a public set of operations. The C++ language uses the concept of classes to implement data abstraction.

**access.** An attribute that determines whether or not a class member is accessible in an expression or declaration.

**access mode.** (1) A technique that is used to obtain a particular logical record from, or to place a particular logical record into, a file assigned to a mass storage device. *ANSI/ISO*. (2) The manner in which files are referred to by a computer. Access can be sequential (records are referred to one after another in the order in which they appear on the file), access can be random (the individual records can be referred to in a nonsequential manner), or access can be dynamic (records can be accessed sequentially or randomly, depending on the form of the input/output request). *IBM*. (3) A particular form of access permitted to a file. *X/Open*.

**ACU (abstract code unit).** A measurement used by the OS/390 C/C++ compiler for judging the size of a function. The number of ACUs that comprise a function is proportional to its size and complexity.

**addressing mode.** See *AMODE*.

**address space.** (1) The range of addresses available to a computer program. *ANSI/ISO*. (2) The complete range of addresses that are available to a programmer. See also *virtual address space*. (3) In the AIX operating system, the code, stack, and data that are accessible by a process. (4) The area of virtual storage available for a particular job. (5) The memory locations that can be referenced by a process. *X/Open*. *ISO.1*.

**alias.** (1) An alternate label; for example, a label and one or more aliases may be used to refer to the same data element or point in a computer program. *ANSI/ISO*. (2) An alternate name for a member of a partitioned data set. *IBM*. (3) In the AIX operating system, an alternative name used for a network. Synonymous with nickname. *IBM*.

**alias name.** (1) A word consisting solely of underscores, digits, and alphabets from the portable file name character set, and any of the following characters: ! % , @. Implementations may allow other characters within alias names as an extension. *X/Open*. (2) An alternate name. *IBM*. (3) A name that is defined in one network to represent a logical unit name in another interconnected network. The alias name does not have to be the same as the real name; if these names are not the same; translation is required. *IBM*.

**alignment.** The storing of data in relation to certain machine-dependent boundaries. *IBM*.

**American National Standards Institute.** See *ANSI/ISO*.

**AMODE (addressing mode).** In MVS, a program attribute that refers to the address length that a program is prepared to handle upon entry. In MVS, addresses may be 24 or 31 bits in length. *IBM.*

**ANSI/ISO (American National Standards Institute).** An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States. *ANSI/ISO.*

**API (application program interface).** A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program. *IBM.*

**application.** (1) The use to which an information processing system is put; for example, a payroll application, an airline reservation application, a network application. *IBM.* (2) A collection of software components used to perform specific types of user-oriented work on a computer. *IBM.*

**application program.** A program written for or by a user that applies to the user's work, such as a program that does inventory control or payroll. *IBM.*

**appropriate privileges.** An implementation-defined means of associating privileges with a process with regard to the function calls and function call options that need special privileges. There may be zero or more such means. *ISO.1.*

**archive libraries.** The archive library file, when created for application program object files, has a special symbol table for members that are object files.

**argument.** (1) A parameter passed between a calling program and a called program. *IBM.* (2) In a function call, an expression that represents a value that the calling function passes to the function specified in the call. Also called *parameter*. (3) In the shell, a parameter passed to a utility as the equivalent of a single string in the *argv* array created by one of the *exec* functions. An argument is one of the options, option-arguments, or operands following the command name. *X/Open.*

**array.** In programming languages, an aggregate that consists of data objects, with identical attributes, each of which may be uniquely referenced by subscripting. *IBM.*

**array element.** A data item in an array. *IBM.*

**ASCII (American National Standard Code for Information Interchange).** The standard code, using a coded character set consisting of 7-bit coded characters

(8 bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. *IBM.*

**Note:** IBM has defined an extension to ASCII code (characters 128-255).

**Assembler H.** An IBM licensed program. Translates symbolic assembler language into binary machine language.

**assembler language.** A source language that includes symbolic language statements in which there is a one-to-one correspondence with the instruction formats and data formats of the computer. *IBM.*

**automatic storage.** Storage that is allocated on entry to a routine or block and is freed on the subsequent return. Sometimes referred to as *stack storage* or *dynamic storage*.

## B

**background process.** (1) A process that does not require operator intervention but can be run by the computer while the workstation is used to do other work. *IBM.* (2) A mode of program execution in which the shell does not wait for program completion before prompting the user for another command. *IBM.* (3) A process that is a member of a background process group. *X/Open. ISO.1.*

**background process group.** Any process group, other than a foreground process group, that is a member of a session that has established a connection with a controlling terminal. *X/Open. ISO.1.*

**backslash.** The character \. This character is named <backslash> in the portable character set.

**based on.** The use of existing classes for implementing new classes.

**Berkeley Socket.** The socket prototype that represents a migration path for programs coded under the HOT1120 and HOT1130 products. It allows use of the BSD4.3 interface and function in the X/Open environment. Its purpose is to expedite the porting of existing BSD4.3 applications.

**binary stream.** (1) An ordered sequence of untranslated characters. (2) A sequence of characters that corresponds on a one-to-one basis with the characters in the file. No character translation is performed on binary streams. *IBM.*

**blank character.** (1) A graphic representation of the space character. *ANSI/ISO.* (2) A character that

represents an empty position in a graphic character string. *ISO Draft*. (3) One of the characters that belong to the *blank* character class as defined via the LC\_CTYPE category in the current locale. In the POSIX locale, a blank character is either a tab or a space character. *X/Open*.

**block.** (1) In programming languages, a compound statement that coincides with the scope of at least one of the declarations contained within it. A block may also specify storage allocation or segment programs for other purposes. *ISO-JTC1*. (2) A string of data elements recorded or transmitted as a unit. The elements may be characters, words or physical records. *ISO Draft*. (3) The unit of data transmitted to and from a device. Each block contains one record, part of a record, or several records.

**boundary alignment.** The position in main storage of a fixed-length field, such as a halfword or doubleword, on a byte-level boundary for that unit of information. *IBM*.

**brackets.** The characters [ (left bracket) and ] (right bracket), also known as *square brackets*. When used in the phrase “enclosed in (square) brackets” the symbol [ immediately precedes the object to be enclosed, and ] immediately follows it. When describing these characters in the portable character set, the names <left-bracket> and <right-bracket> are used. *X/Open*.

**breakpoint.** A point in a computer program where execution may be halted. A breakpoint is usually at the beginning of an instruction where halts, caused by external intervention, are convenient for resuming execution. *ISO Draft*.

**built-in.** (1) A function that the compiler will automatically inline instead of making the function call, unless the programmer specifies not to inline. (2) In programming languages, pertaining to a language object that is declared by the definition of the programming language; for example the built-in function SIN in PL/I, the predefined data type INTEGER in FORTRAN. *ISO-JTC1*. Synonymous with predefined. *IBM*.

## C

**C++ class library.** See *class library*.

**C++ library.** A system library that contains common C++ language subroutines for file access, memory allocation, and other functions.

**call.** To transfer control to a procedure, program, routine, or subroutine. *IBM*.

**callable services.** A set of services that can be invoked by a Language Environment-conforming high level language using the conventional Language

Environment-defined call interface, and usable by all programs sharing the Language Environment conventions.

Use of these services helps to decrease an application's dependence on the specific form and content of the services delivered by any single operating system.

**Callable Services Library (CSL).** A collection of assembler routines that you can use to:

- Call Shared File System (SFS) functions
- Interface with the VM command environment through a REXX EXEC
- Invoke the VM/CMS extract/replace facility, which allows application programs to obtain or modify selected system information without release or VM/CMS system dependencies
- Call program-to-program communication functions using the Common Programming Interface (CPI) for communications.

**call chain.** A trace of all active routines and subroutines.

**caller.** A routine that calls another routine.

**cancelability point.** A specific point within the current thread that is enabled to solicit cancel requests. This is accomplished using the pthread\_testintr() function.

**CASE (Computer-Aided Software Engineering).** A set of tools or programs to help develop complex applications. *IBM*.

**cast.** In the C and C++ languages, an expression that converts the type of the operand to a specified data type (the operator). *IBM*.

**cataloged procedures.** A set of control statements placed in a library and retrievable by name. *IBM*.

**character.** (1) A letter, digit, or other symbol that is used as part of the organization, control, or representation of data. A character is often in the form of a spatial arrangement of adjacent or connected strokes. *ANSI/ISO*. (2) A sequence of one or more bytes representing a single graphic symbol or control code. This term corresponds to the ISO C standard term *multibyte character* (multi-byte character), where a single-byte character is a special case of the multi-byte character. Unlike the usage in the ISO C standard, *character* here has no necessary relationship with storage space, and *byte* is used when storage space is discussed. *X/Open*. *ISO.1*.

**character array.** An array of type char. *X/Open*.

**character class.** A named set of characters sharing an attribute associated with the name of the class. The

classes and the characters that they contain are dependent on the value of the LC\_CTYPE category in the current locale. *X/Open*.

**character constant.** (1) A constant with a character value. *IBM*. (2) A string of any of the characters that can be represented, usually enclosed in apostrophes. *IBM*. (3) In some languages, a character enclosed in apostrophes. *IBM*.

**character set.** (1) A finite set of different characters that is complete for a given purpose; for example, the character set in ISO Standard 646, 7-bit Coded Character Set for Information Processing Interchange. *ISO Draft*. (2) All the valid characters for a programming language or for a computer system. *IBM*. (3) A group of characters used for a specific reason; for example, the set of characters a printer can print. *IBM*. (4) See also *portable character set*.

**character special file.** (1) A special file that provides access to an input or output device. The character interface is used for devices that do not use block I/O. *IBM*. (2) A file that refers to a device. One specific type of character special file is a terminal device file. *X/Open*. *ISO.1*.

**character string.** A contiguous sequence of characters terminated by and including the first null byte. *X/Open*.

**child.** A node that is subordinate to another node in a tree structure. Only the root node is not a child.

**CICS (Customer Information Control System).** Pertaining to an IBM licensed program that enables transactions entered at remote terminals to be processed concurrently by user-written application programs. It includes facilities for building, using, and maintaining databases. *IBM*.

**class.** (1) A C++ aggregate that may contain functions, types, and user-defined operators in addition to data. Classes may be defined hierarchically, allowing one class to be derived from another, and may restrict access to its members. (2) A user-defined data type. A class data type can contain both data representations (data members) and functions (member functions).

**class library.** A collection of C++ classes.

**C library.** A system library that contains common C language subroutines for file access, string operators, character operations, memory allocation, and other functions. *IBM*.

**CLIST.** A programming language that typically executes a list of TSO commands.

**COBOL (Common Business-Oriented Language).** A high-level language, based on English, that is primarily used for business applications.

**coded character set.** (1) A set of graphic characters and their code point assignments. The set may contain fewer characters than the total number of possible characters: some code points may be unassigned. *IBM*. (2) A coded set whose elements are single characters; for example, all characters of an alphabet. *ISO Draft*. (3) Loosely, a code. *ANSI/ISO*.

**code page.** (1) An assignment of graphic characters and control function meanings to all code points; for example, assignment of characters and meanings to 256 code points for an 8-bit code, assignment of characters and meanings to 128 code points for a 7-bit code. (2) A particular assignment of hexadecimal identifiers to graphic characters.

**code point.** (1) A 1-byte code representing one of 256 potential characters. (2) An identifier in an alert description that represents a short unit of text. The code point is replaced with the text by an alert display program.

**codeset.** Synonym for code element set. *IBM*.

**collating element.** The smallest entity used to determine the logical ordering of character or wide-character strings. A collating element consists of either a single character, or two or more characters collating as a single entity. The value of the LC\_COLLATE category in the current locale determines the current set of collating elements. *X/Open*.

**collating sequence.** (1) A specified arrangement used in sequencing. *ISO-JTC1*. *ANSI/ISO*. (2) An ordering assigned to a set of items, such that any two sets in that assigned order can be collated. *ANSI/ISO*. (3) The relative ordering of collating elements as determined by the setting of the LC\_COLLATE category in the current locale. The character order, as defined for the LC\_COLLATE category in the current locale, defines the relative order of all collating elements, such that each element occupies a unique position in the order. This is the order used in ranges of characters and collating elements in regular expressions and pattern matching. In addition, the definition of the collating weights of characters and collating elements uses collating elements to represent their respective positions within the collation sequence.

**collation.** The logical ordering of character or wide-character strings according to defined precedence rules. These rules identify a collation sequence between the collating elements, and such additional rules that can be used to order strings consisting of multiple collating elements. *X/Open*.



**collection.** (1) An abstract class without any ordering, element properties, or key properties. All abstract classes are derived from collection. (2) In a general sense, an implementation of an abstract data type for storing elements.

**Collection Class Library.** A set of classes that provide basic functions for collections, and can be used as base classes.

**command.** A request to perform an operation or run a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

**COMMAREA.** A communication area made available to applications running under CICS.

**common anchor area (CAA).** Dynamically acquired storage that represents a OS/390 C/C++ thread. Thread-related storage/resources are anchored off of the CAA. This area acts as a central communications area for the program, holding addresses of various storage and error-handling routines, and control blocks. The CAA is anchored by an address in register 12.

**Communications Server.** A RISC System/6000 connected to one or more OS/390 UNIX systems, allowing terminals on asynchronous ports of the RISC System/6000 to operate as if they were connected directly to the OS/390 UNIX system. Synonymous with OCS.

**compilation unit.** (1) A portion of a computer program sufficiently complete to be compiled correctly. *IBM.* (2) A single compiled file and all its associated include files. (3) An independently compilable sequence of high-level language statements. Each high-level language product has different rules for what makes up a compilation unit.

**Complex Mathematics library.** A C++ class library that provides the facilities to manipulate complex numbers and perform standard mathematical operations on them.

**condition.** (1) A relational expression that can be evaluated to a value of either true or false. *IBM.* (2) An exception that has been enabled, or recognized, by the Language Environment and thus is eligible to activate user and language condition handlers. Any alteration to the normal programmed flow of an application. Conditions can be detected by the hardware/operating system and result in an interrupt. They can also be detected by language-specific generated code or language library code.

**const.** (1) An attribute of a data object that declares the object cannot be changed. (2) A keyword that allows you to define a variable whose value does not change.

**constant.** (1) In programming languages, a language object that takes only one specific value. *ISO-JTC1.* (2) A data item with a value that does not change. *IBM.*

**constant expression.** An expression having a value that can be determined during compilation and that does not change during the running of the program. *IBM.*

**control character.** (1) A character whose occurrence in a particular context specifies a control function. *ISO Draft.* (2) Synonymous with nonprinting character. *IBM.* (3) A character, other than a graphic character, that affects the recording, processing, transmission, or interpretation of text. *X/Open.*

**controlling process.** The session leader that establishes the connection to the controlling terminal. If the terminal ceases to be a controlling terminal for this session, the session leader ceases to be the controlling process. *X/Open. ISO.1.*

**controlling terminal.** A terminal that is associated with a session. Each session may have at most one controlling terminal associated with it, and a controlling terminal is associated with exactly one session. Certain input sequences from the controlling terminal cause signals to be sent to all processes in the process group associated with the controlling terminal. *X/Open. ISO.1.*

**conversion.** (1) In programming languages, the transformation between values that represent the same data item but belong to different data types. Information may be lost because of conversion since accuracy of data representation varies among different data types. *ISO-JTC1.* (2) The process of changing from one method of data processing to another or from one data processing system to another. *IBM.* (3) The process of changing from one form of representation to another; for example to change from decimal representation to binary representation. *IBM.* (4) A change in the type of a value. For example, when you add values having different data types, the compiler converts both values to a common form before adding the values.

**conversion descriptor.** A per-process unique value used to identify an open codeset conversion. *X/Open.*

**coordinated universal time (UTC).** Equivalent to Greenwich Mean Time (GMT)

**Cross System Product.** See *CSP.*

**CSP (Cross System Product).** A set of licensed programs designed to permit the user to develop and run applications using independently defined maps (display and printer formats), data items (records, working storage, files, and single items), and processes (logic). The Cross System Product set consists of two parts: Cross System Product/Application Development

(CSP/AD) and Cross System Product/Application Execution (CSP/AE). *IBM*.

**current working directory.** (1) A directory, associated with a process, that is used in path-name resolution for path names that do not begin with a slash. *X/Open*. *ISO.1*. (2) In DOS, the directory that is searched when a file name is entered with no indication of the directory that lists the file name. DOS assumes that the current directory is the root directory unless a path to another directory is specified. *IBM*. (3) In the OS/2 operating system, the first directory in which the operating system looks for programs and files and stores temporary files and output. *IBM*. (4) In the AIX operating system, a directory that is active and that can be displayed. Relative path name resolution begins in the current directory. *IBM*.

**Customer Information Control System.** See *CICS*.

## D

**data abstraction.** A data type with a private representation and a public set of operations. The C++ language uses the concept of classes to implement data abstraction.

**DATABASE 2.** Pertaining to an IBM relational database.

**data division.** One of the four main parts of a COBOL program. The data division describes the files to be used in the program and the records contained within the files. It also describes any internal working storage records that are needed. *IBM*.

**data object.** (1) A storage area used to hold a value. (2) Anything that exists in storage and on which operations can be performed, such as files, programs, classes, or arrays. (3) In a program, an element of data structure, such as a file, array, or operand, that is needed for the execution of a program and that is named or otherwise specified by the allowable character set of the language in which a program is coded. *IBM*.

**data set.** Under MVS, a named collection of related data records that is stored and retrieved by an assigned name. Equivalent to a CMS *file*.

**data stream.** A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format. *IBM*.

**data structure.** The internal data representation of an implementation.

**data type.** The properties and internal representation that characterize data.

**DBCS (double-byte character set).** A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets.

Because each character requires 2 bytes, the typing, display, and printing of DBCS characters requires hardware and programs that support DBCS. *IBM*.

**ddname (data definition name).** (1) The logical name of a file within an application. The ddname provides the means for the logical file to be connected to the physical file. (2) The part of the data definition before the equal sign. It is the name used in a call to *fopen* or *freopen* to refer to the data definition stored in the environment.

**DD statement (data definition statement).** (1) In MVS, serves as the connection between the logical name of a file and the physical name of the file. (2) A job control statement that defines a file to the operating system, and is a request to the operating system for the allocation of input/output resources.

**declaration.** (1) In the C and C++ languages, a description that makes an external object or function available to a function or a block statement. *IBM*. (2) Establishes the names and characteristics of data objects and functions used in a program.

**default argument.** An argument that is declared with a default value in a function prototype or declaration. If a call to the function omits this argument, the default value is used. Arguments with default values must be the trailing arguments in a function prototype argument list.

**default class.** A class with preprogrammed definitions that can be used for simple implementations.

**default locale.** (1) The C locale, which is always used when no selection of locale is performed. (2) A system default locale, named by locale-related environmental variables.

**define directive.** A preprocessor statement that directs the preprocessor to replace an identifier or macro invocation with special code.

**definition.** (1) A data description that reserves storage and may provide an initial value. (2) A declaration that allocates storage, and may initialize a data object or specify the body of a function.

**degree.** The number of children of a node.

**delete.** (1) A C++ keyword that identifies a free storage deallocation operator. (2) A C++ operator used to destroy objects created by *new*.

**demangling.** The conversion of mangled names back to their original source code names. During C++ compilation, identifiers such as function and static class member names are mangled (encoded) with type and scoping information to ensure type-safe linkage. These mangled names appear in the object file and the final executable file. Demangling (decoding) converts these names back to their original names to make program debugging easier. See also *mangling*.

**descriptor.** PL/I control block that holds information such as string lengths, array subscript bounds, and area sizes, and is passed from one PL/I routine to another during run time.

**destructor.** A special member function that has the same name as its class, preceded by a tilde (~), and that "cleans up" after an object of that class, for example, freeing storage that was allocated when the object was created. A destructor has no arguments and no return type.

**device.** A computer peripheral or an object that appears to the application as such. *X/Open. ISO.1.*

**device ID.** A non-negative integer used to identify a device. *X/Open.*

**difference.** Given two sets A and B, the difference (A-B) is the set of all elements contained in A but not in B. For bags, there is an additional rule for duplicates: If bag P contains an element *m* times and bag Q contains the same element *n* times, then, if *m* > *n*, the difference contains that element *m-n* times. If *m* is equal to or less than *n*, the difference contains that element zero times.

**directory.** A type of file containing the names and controlling information for other files or other directories. *IBM.*

**display.** To direct the output to the user's terminal. If the output is not directed to the terminal, the results are undefined. *X/Open.*

**dot.** The file name consisting of a single dot character (.). *X/Open. ISO.1.*

**dot-dot.** The file name consisting solely of two dot characters (..). *X/Open. ISO.1.*

**double-byte character set.** See *DBCS*.

**dump.** To copy data in a readable format from main or auxiliary storage onto an external medium such as tape, diskette, or printer. *IBM.*

**dynamic.** Pertaining to an operation that occurs at the time it is needed rather than at a predetermined or fixed time. *IBM.*

**dynamic allocation.** Assignment of system resources to a program when the program is executed rather than when it is loaded into main storage. *IBM.*

**dynamic link library (DLL).** A file containing executable code and data bound to a program at load time or run time. The code and data in a dynamic link library can be shared by several applications simultaneously.

**dynamic storage.** Synonym for *automatic storage*.

## E

**EBCDIC (extended binary-coded decimal interchange code).** A coded character set of 256 8-bit characters. *IBM.*

**effective group ID.** An attribute of a process that is used in determining various permissions, including file access permissions. This value is subject to change during the process lifetime, as described in the *exec* family of functions and *setgid()*. *X/Open. ISO.1.*

**effective user ID.** (1) The user ID associated with the last authenticated user or the last *setuid()* program. It is equal to either the real or the saved user ID. (2) The current user ID, but not necessarily the user's login ID; for example, a user logged in under a login ID may change to another user's ID. The ID to which the user changes becomes the effective user ID until the user switches back to the original login ID. All discretionary access decisions are based on the effective user ID. *IBM.* (3) An attribute of a process that is used in determining various permissions, including file access permissions. This value is subject to change during the process lifetime, as described in *exec* and *setuid()*. *X/Open. ISO.1.*

**E-format.** Floating-point format, consisting of a number in scientific notation. *IBM.*

**element.** The component of an array, subrange, enumeration, or set.

**empty directory.** A directory that contains, at most, directory entries for dot and dot-dot. *X/Open. ISO.1.*

**empty string.** (1) A string whose first byte is a null byte. Synonymous with null string. *X/Open.* (2) A character array whose first element is a null character. *ISO.1.*

**enclave.** In the Language Environment for MVS and VM, an independent collection of routines, one of which is designated as the main routine. An enclave is roughly analogous to a program or run unit.

**entry point.** In assembler language, the address or label of the first instruction that is executed when a routine is entered for execution.

**epoch.** The time zero hours, zero minutes, zero seconds, on January 1, 1970 Coordinated Universal Time. *X/Open. ISO.1.*

**exception.** (1) Any user, logic, or system error detected by a function that does not itself deal with the error but passes the error on to a handling routine (also called throwing the exception). (2) In programming languages, an abnormal situation that may arise during execution, that may cause a deviation from the normal execution sequence, and for which facilities exist in a programming language to define, raise, recognize, ignore, and handle it; for example, (ON-) condition in PL/I, exception in ADA. *ISO-JTC1.*

**executable file.** A regular file acceptable as a new process image file by the equivalent of the *exec* family of functions, and thus usable as one form of a utility. The standard utilities described as compilers can produce executable files, but other unspecified methods of producing executable files may also be provided. The internal format of an executable file is unspecified, but a conforming application cannot assume an executable file is a text file. *X/Open.*

**extension.** (1) An element or function not included in the standard language. (2) File name extension.

## F

**feature test macro.** A macro (*#define*) used to determine whether a particular set of features will be included from a header. *X/Open. ISO.1.*

**fetch control block (FECB).** An executable dynamic stub which is created by a *fetch()* function call. The stub transfers control to the true entry point of the module specified in the *fetch* call. The stub also switches the writable static environment thereby giving each instance of the fetched routine its own global data.

**FIFO special file.** A type of file with the property that data written to such a file is read on a first-in-first-out basis. Other characteristics of FIFOs are described in *open()*, *read()*, *write()*, and *lseek()*. *X/Open. ISO.1.*

**file access permissions.** The standard file access control mechanism uses the file permission bits. The bits are set at the time of file creation by functions such as *open()*, *creat()*, *mkdir()*, and *mkfifo()* are changed by *chmod()*. The bits are read by *stat()* or *fstat()*. *X/Open.*

**file descriptor.** (1) A small positive integer that the system uses instead of the file name to identify an open file. *IBM.* (2) A per-process unique, non-negative integer used to identify an open file for the purpose of file access. *ISO.1.*

The value of a file descriptor is from zero to {*OPEN\_MAX*}—which is defined in *limits.h*. A process can have no more than {*OPEN\_MAX*} file descriptors open simultaneously. File descriptors may also be used to implement directory streams. *X/Open.*

**file mode.** An object containing the *file mode bits* and file type of a file, as described in *<sys/stat.h>*. *X/Open.*

**file mode bits.** A file's file permission bits, set-user-ID-on-execution bit (*S\_ISUID*) and set-group-ID-on-execution bit (*S\_ISGID*). *X/Open.*

**file offset.** The byte position in the file where the next I/O operation begins. Each open file description associated with a regular file, block special file, or directory has a file offset. A character special file that does not refer to a terminal device may have a file offset. There is no file offset specified for a pipe or FIFO. *X/Open. ISO.1.*

**file owner.** The user who has the highest level of access authority to a file, as defined by the file. *IBM.*

**file permission bits.** Information about a file that is used, along with other information, to determine if a process has read, write, or execute/search permission to a file. The bits are divided into three parts: owner, group, and other. Each part is used with the corresponding file class of process. These bits are contained in the file mode, as described in *<sys/stat.h>*. The detailed usage of the file permission bits is described in *file access permissions. X/Open. ISO.1.*

**file scope.** A name declared outside all blocks and classes has file scope and can be used after the point of declaration in a source file.

**first element.** The element visited first in an iteration over a collection. Each collection has its own definition for first element. For example, the first element of a sorted set is the element with the smallest value.

**flat collection.** A collection that has no hierarchical structure.

**foreground process.** (1) In the AIX operating system, a process that must run to completion before another command is issued to the shell. The foreground process is in the foreground process group, which is the group that receives the signals generated by a terminal. *IBM.* (2) A computer program that preempts the use of computer facilities while running. (3) A process that is a member of a foreground process group. *X/Open. ISO.1.*

**foreground process group.** (1) The group that receives the signals generated by a terminal. *IBM.* (2) A process group whose member processes have certain privileges, denied to processes in background process groups, when accessing their controlling ter-

minal. Each session that has established a connection with a controlling terminal has exactly one process group of the session as the foreground process group of that controlling terminal. *X/Open. ISO.1.*

**foreground process group ID.** The process group ID of the foreground process group. *X/Open. ISO.1.*

**for statement.** A looping statement that contains the word *for* followed by a list of expressions enclosed in parentheses (the condition) and a statement (the action). Each expression in the parenthesized list is separated by a semicolon. You can omit any of the expressions, but you cannot omit the semicolons.

**function.** A named group of statements that can be called and evaluated and can return a value to the calling statement. *IBM.*

**function call.** An expression that moves the path of execution from the current function to a specified function and evaluates to the return value provided by the called function. A function call contains the name of the function to which control moves and a parenthesized list of values. *IBM.*

## G

**GDDM (Graphical Data Display Manager).** Pertaining to an IBM licensed program that provides a group of routines that allows pictures to be defined and displayed procedurally through function routines that correspond to graphic primitives. *IBM.*

**global.** Pertaining to information available to more than one program or subroutine. *IBM.*

**global variable.** A symbol defined in one program module that is used in other independently compiled program modules.

**GMT (Greenwich Mean Time).** The solar time at the meridian of Greenwich, formerly used as the prime basis of standard time throughout the world. GMT has been superseded by coordinated universal time (UTC).

**Greenwich Mean Time.** See GMT.

**group ID.** (1) A non-negative integer that is used to identify a group of system users. Each system user is a member of at least one group. When the identity of a group is associated with a process, a group ID value is referred to as a real group ID, an effective group ID, one of the supplementary group IDs or a saved set-group-ID. *X/Open.* (2) A non-negative integer, which can be contained in an object of type *gid\_t*, that is used to identify a group of system users. *ISO.1.*

## H

**header file.** A text file that contains declarations used by a group of functions, programs, or users.

**hyperspace memory file.** A file used under MVS to deal with memory files as large as 2 gigabytes.

## I

**I18N.** Abbreviation for *internationalization*.

**identifier.** (1) One or more characters used to identify or name a data element and possibly to indicate certain properties of that data element. *ANSI/ISO.* (2) In programming languages, a token that names a data object such as a variable, an array, a record, a subprogram, or a function. *ANSI/ISO.* (3) A sequence of letters, digits, and underscores used to identify a data object or function. *IBM.*

**if statement.** A conditional statement that contains the keyword *if*, followed by an expression in parentheses (the condition), a statement (the action), and an optional *else* clause (the alternative action). *IBM.*

**ILC (interlanguage call).** A function call made by one language to a function coded in another language. Interlanguage calls are used to communicate between programs written in different languages.

**ILC (interlanguage communication).** The ability of routines written in different programming languages to communicate. ILC support enables the application writer to readily build applications from component routines written in a variety of languages.

**include directive.** A preprocessor directive that causes the preprocessor to replace the statement with the contents of a specified file.

**include file.** See *header file*.

**inheritance.** A technique that allows the use of an existing class as the base for creating other classes.

**initializer.** An expression used to initialize data objects. In the C++ language, there are three types of initializers:

1. An expression followed by an assignment operator is used to initialize fundamental data type objects or class objects that have copy constructors.
2. An expression enclosed in braces ( { } ) is used to initialize aggregates.
3. A parenthesized expression list is used to initialize base classes and members using constructors.

**input stream.** A sequence of control statements and data submitted to a system from an input unit. Synonymous with input job stream, job input stream. *IBM.*

**instance.** An object-oriented programming term synonymous with object. An instance is a particular instantiation of a data type. It is simply a region of storage that contains a value or group of values. For example, if a class `box` is previously defined, two instances of a class `box` could be instantiated with the declaration:

```
box box1, box2;
```

**instruction.** A program statement that specifies an operation to be performed by the computer, along with the values or locations of operands. This statement represents the programmer's request to the processor to perform a specific operation.

**integer constant.** A decimal, octal, or hexadecimal constant.

**interlanguage call.** See *ILC*. (1)

**internationalization.** The capability of a computer program to adapt to the requirements of different native languages, local customs, and coded character sets. *X/Open.*

Synonymous with *I18N*.

**interoperability.** Under MVS, resources existing in parent processes do not exist in a forked child process.

**interprocess communication.** (1) The exchange of information between processes or threads through semaphores, queues, and shared memory. (2) The process by which programs communicate data to each other and to synchronize their activities. Semaphores, signals, and internal message queues are common methods of inter-process communication.

**I/O Stream library.** A class library that provides the facilities to deal with many varieties of input and output.

**ISPF (Interactive System Productivity Facility).** An IBM licensed program that serves as a full-screen editor and dialogue manager. Used for writing application programs, it provides a means of generating standard screen panels and interactive dialogues between the application programmer and terminal user. (ISPF)

**iteration.** The process of repeatedly applying a function to a series of elements in a collection until some condition is satisfied.

## J

**job control.** A facility that allows users to selectively stop (suspend) the execution of a process and continue (resume) their execution at a later point.

The user typically employs this facility via the interactive interface jointly supplied by the terminal I/O driver and a command interpreter. *X/Open. ISO.1.*

## K

**key function.** (1) When used on a flat collection, a function that returns a reference to the key of an element. (2) In general, a function, called by a member function, that manipulates the keys of a class.

**key identifier.** The name associated with the key for a thread. Each thread in an application, where a key has been created, uses the key to set its own unique value in the thread storage area for the application.

**key set.** An unordered flat collection that uses keys and does not allow duplicate elements.

**keyword.** (1) A predefined word reserved for the C and C++ languages, that may not be used as an identifier. (2) A symbol that identifies a parameter in JCL.

**kind attribute.** An attribute for a mutex attribute object. This attribute's value determines whether the mutex can be locked once or more than once for a thread and whether state changes to the mutex will be reported to the debug interface.

## L

**label.** An identifier within or attached to a set of data elements. *ISO Draft.*

**Language Environment.** Abbreviated form of IBM Language Environment for MVS and VM. Pertaining to an IBM software product that provides a common runtime environment and runtime services to applications compiled by Language Environment-conforming compilers.

**leaves.** Nodes without children. Synonymous with terminals.

**library.** (1) A collection of functions, calls, subroutines, or other data. *IBM.* (2) A set of object modules that can be specified in a link command.

**line.** A sequence of zero or more non-new-line characters plus a terminating new-line character. *X/Open.*

**link.** To interconnect items of data or portions of one or more computer programs; for example, linking of

object programs by a linkage editor to produce an executable file.

**link pack area (LPA).** In MVS, an area of storage containing reenterable routines from system libraries. Their presence in main storage saves loading time.

**literal.** (1) In programming languages, a lexical unit that directly represents a value; for example, 14 represents the integer fourteen, "APRIL" represents the string of characters APRIL, 3.0005E2 represents the number 300.05. *ISO-JTC1.* (2) A symbol or a quantity in a source program that is itself data, rather than a reference to data. *IBM.* (3) A character string whose value is given by the characters themselves; for example, the numeric literal 7 has the value 7, and the character literal CHARACTERS has the value CHARACTERS. *IBM.*

**load module.** All or part of a computer program in a form suitable for loading into main storage for execution. A load module is usually the output of a linkage editor. *ISO Draft.*

**local.** (1) In programming languages, pertaining to the relationship between a language object and a block such that the language object has a scope contained in that block. *ISO-JTC1.* (2) Pertaining to that which is defined and used only in one subdivision of a computer program. *ANSI/ISO.*

**locale.** The definition of the subset of a user's environment that depends on language and cultural conventions. *X/Open.*

**lvalue.** An expression that represents a data object that can be both examined and altered.

## M

**macro.** An identifier followed by arguments (may be a parenthesized list of arguments) that the preprocessor replaces with the replacement code located in a preprocessor #define directive.

**macro instruction.** Synonym for *macro*.

**mangling.** The encoding during compilation of identifiers such as function and variable names to include type and scope information. The prelinker uses these mangled names to ensure type-safe linkage. See also *demangling*.

**mask.** A pattern of characters that controls the keeping, deleting, or testing of portions of another pattern of characters. *ISO-JTC1. ANSI/ISO.*

**member.** A data object or function in a structure, union, or class. Members can also be classes, enumerations, bit fields, and type names.

**member function.** (1) An operator or function that is declared as a member of a class. A member function has access to the private and protected data members and member functions of objects of its class. Member functions are also called methods. (2) A function that performs operations on a class.

**method.** In the C++ language, a synonym for member function.

**migrate.** To move to a changed operating environment, usually to a new release or version of a system. *IBM.*

**mode.** A collection of attributes that specifies a file's type and its access permissions. *X/Open. ISO.1.*

**module.** A program unit that usually performs a particular function or related functions, and that is distinct and identifiable with respect to compiling, combining with other units, and loading.

**multibyte character.** A mixture of single-byte characters from a single-byte character set and double-byte characters from a double-byte character set.

**multitasking.** A mode of operation that allows concurrent performance, or interleaved execution of two or more tasks. *ISO-JTC1. ANSI/ISO.*

**mutex.** A flag used by a semaphore to protect shared resources. The mutex is locked and unlocked by threads in a program. A mutex can only be locked by one thread at a time and can only be unlocked by the same thread that locked it. The current owner of a mutex is the thread that it is currently locked by. An unlocked mutex has no current owner.

**mutex attribute object.** Allows the user to manage the characteristics of mutexes in their application by defining a set of values to be used for the mutex during its creation. A mutex attribute object allows the user to create many mutexes with the same set of characteristics without redefining the same set of characteristics for each mutex created.

**mutex object.** Used to identify a mutex.

## N

**name.** In the C++ language, a name is commonly referred to as an identifier. However, syntactically, a name can be an identifier, operator function name, conversion function name, destructor name or qualified name.

**named pipe.** A FIFO file. Named pipes allow transfer of data between processes in a FIFO manner and synchronization of process execution. Allows processes to

communicate even though they do not know what processes are on the other end of the pipe.

**name space.** A category used to group similar types of identifiers.

**node.** In a tree structure, a point at which subordinate items of data originate. *ANSI/ISO.*

**NULL.** In the C and C++ languages, a pointer that does not point to a data object. *IBM.*

**null character (NUL).** The ASCII or EBCDIC character '\0' with the hex value 00, all bits turned off. It is used to represent the absence of a printed or displayed character. This character is named <NUL> in the portable character set.

**null pointer.** The value that is obtained by converting the number 0 into a pointer; for example, (void \*) 0. The C and C++ languages guarantee that this value will not match that of any legitimate pointer, so it is used by many functions that return pointers to indicate an error. *X/Open.*

**null string.** (1) A string whose first byte is a null byte. Synonymous with *empty string*. *X/Open.* (2) A character array whose first element is a null character. *ISO.1.*

**null value.** A parameter position for which no value is specified. *IBM.*

**null wide-character code.** A wide-character code with all bits set to zero. *X/Open.*

**number sign.** The character #, also known as *pound sign* and *hash sign*. This character is named <number-sign> in the portable character set.

## O

**object.** (1) A region of storage. An object is created when a variable is defined or new is invoked. An object is destroyed when it goes out of scope. (See also *instance*.) (2) In object-oriented design or programming, an abstraction consisting of data and the operations associated with that data. See also *class*. *IBM.* (3) An instance of a class.

**object module.** (1) All or part of an object program sufficiently complete for linking. Assemblers and compilers usually produce object modules. *ISO Draft.* (2) A set of instructions in machine language produced by a compiler from a source program. *IBM.*

**object-oriented programming.** A programming approach based on the concepts of data abstraction and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates not

on how something is accomplished, but on what data objects comprise the problem and how they are manipulated.

**OS/390 UNIX System Services(OS/390 UNIX).** The set of functions provided by the Shell and Utilities, kernel, debugger, file system, C/C++ Run-Time Library, Language Environment, and other elements of the OS/390 operating system that allow users to write and run application programs that conform to UNIX standards.

**open file.** A file that is currently associated with a file descriptor. *X/Open. ISO.1.*

**open file description.** A record of how a process or a group of processes are accessing a file. Each file descriptor refers to exactly one open file description, but an open file description can be referred to by more than one file descriptor. A file offset, file status, and file access modes are attributes of an open file description. *X/Open. ISO.1.*

**Open Socket.** The API type of socket for the HOT1120 and HOT1130 OS/390 UNIX products, which use a BSD4.3 interface. In OS/390 UNIX Release 3, this interface is available with the C/C++ MVS Language Environment, Release 1.5; see **Berkeley Socket**. This API will be deleted from any replacement of the HOT1130 OS/390 UNIX product.

**operand.** An entity on which an operation is performed. *ISO-JTC1. ANSI/ISO.*

**operating system (OS).** Software that controls functions such as resource allocation, scheduling, input/output control, and data management.

**operator precedence.** In programming languages, an order relation defining the sequence of the application of operators within an expression. *ISO-JTC1.*

**orientation of a stream.** After application of an input or output function to a stream, it becomes either byte-oriented or wide-oriented. A byte-oriented stream is a stream that had a byte input or output function applied to it when it had no orientation. A wide-oriented stream is a stream that had a wide character input or output function applied to it when it had no orientation. A stream has no orientation when it has been associated with an external file but has not had any operations performed on it.

**OCS.** Outboard Communications Server. See *Communications Server*.

**overflow.** (1) A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage. (2) That portion of an operation that exceeds the capacity of the intended unit of storage. *IBM.*



**overloading.** An object-oriented programming technique that allows you to redefine functions and most standard C++ operators when the functions and operators are used with class types.

## P

**pack.** To store data in a compact form in such a way that the original form can be recovered.

**parameter.** (1) In the C and C++ languages, an object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier following the macro name in a function-like macro definition. *X/Open*. (2) Data passed between programs or procedures. *IBM*.

**parent process.** (1) The program that originates the creation of other processes by means of `spawn` or `exec` function calls. See also *child process*. (2) A process that creates other processes.

**parent process ID.** (1) An attribute of a new process identifying the parent of the process. The parent process ID of a process is the process ID of its creator, for the lifetime of the creator. After the creator's lifetime has ended, the parent process ID is the process ID of an implementation-dependent system process. *X/Open*. (2) An attribute of a new process after it is created by a currently active process. *ISO.1*.

**partitioned data set (PDS).** A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data. *IBM*.

**partitioned data set extended (PDSE).** See *PDSE*.

**path name.** (1) A string that is used to identify a file. A path name consists of, at most, {`PATH_MAX`} bytes, including the terminating null character. It has an optional beginning slash, followed by zero or more file names separated by slashes. If the path name refers to a directory, it may also have one or more trailing slashes. Multiple successive slashes are considered to be the same as one slash. A path name that begins with two successive slashes may be interpreted in an implementation-dependent manner, although more than two leading slashes will be treated as a single slash. The interpretation of the path name is described in *pathname resolution*. *ISO.1*. (2) A file name specifying all directories leading to the file.

**path name resolution.** Path name resolution is performed for a process to resolve a path name to a particular file in a file hierarchy. There may be multiple path names that resolve to the same file. *X/Open*.

**path prefix.** A path name, with an optional ending slash, that refers to a directory. *ISO.1*.

**pattern.** A sequence of characters used either with regular expression notation or for path name expansion, as a means of selecting various characters strings or path names, respectively. The syntaxes of the two patterns are similar, but not identical. *X/Open*.

**PDSE.** Partitioned data set extended, similar to *partitioned data set*, but with extended capabilities.

**period.** The character (`.`). The term *period* is contrasted against *dot*, which is used to describe a specific directory entry. This character is named `<period>` in the portable character set.

**permissions.** Codes that determine how a file can be used by any users who work on the system. See also *file access permissions*. *IBM*.

**pipe.** (1) An object accessed by one of the pair of file descriptors created by the `pipe()` function. Once created, the file descriptors can be used to manipulate it, and it behaves identically to a FIFO special file when accessed in this way. It has no name in the file hierarchy. *X/Open*. *ISO.1*. (2) To direct data so that the output from one process becomes the input to another process. The standard output of one command can be connected to the standard input of another with the pipe operator (`|`). Two commands connected in this way constitute a pipeline. *IBM*.

**pointer.** In the C and C++ languages, a variable that holds the address of a data object or a function. *IBM*.

**portable character set.** The set of characters specified in POSIX 1003.2, section 2.4:

<code>&lt;NUL&gt;</code>	
<code>&lt;alert&gt;</code>	
<code>&lt;backspace&gt;</code>	
<code>&lt;tab&gt;</code>	
<code>&lt;newline&gt;</code>	
<code>&lt;vertical-tab&gt;</code>	
<code>&lt;form-feed&gt;</code>	
<code>&lt;carriage-return&gt;</code>	
<code>&lt;space&gt;</code>	
<code>&lt;exclamation-mark&gt;</code>	<code>!</code>
<code>&lt;quotation-mark&gt;</code>	<code>"</code>
<code>&lt;number-sign&gt;</code>	<code>#</code>
<code>&lt;dollar-sign&gt;</code>	<code>\$</code>
<code>&lt;percent-sign&gt;</code>	<code>%</code>
<code>&lt;ampersand&gt;</code>	<code>&amp;</code>
<code>&lt;apostrophe&gt;</code>	<code>'</code>
<code>&lt;left-parenthesis&gt;</code>	<code>(</code>
<code>&lt;right-parenthesis&gt;</code>	<code>)</code>
<code>&lt;asterisk&gt;</code>	<code>*</code>
<code>&lt;plus-sign&gt;</code>	<code>+</code>
<code>&lt;comma&gt;</code>	<code>,</code>
<code>&lt;hyphen&gt;</code>	<code>-</code>
<code>&lt;hyphen-minus&gt;</code>	<code>-</code>
<code>&lt;period&gt;</code>	<code>.</code>
<code>&lt;slash&gt;</code>	<code>/</code>
<code>&lt;zero&gt;</code>	<code>0</code>

<one>	1
<two>	2
<three>	3
<four>	4
<five>	5
<six>	6
<seven>	7
<eight>	8
<nine>	9
<colon>	:
<semicolon>	;
<less-than-sign>	<
<equals-sign>	=
<greater-than-sign>	>
<question-mark>	?
<commercial-at>	@
<A>	A
<B>	B
<C>	C
<D>	D
<E>	E
<F>	F
<G>	G
<H>	H
<I>	I
<J>	J
<K>	K
<L>	L
<M>	M
<N>	N
<O>	O
<P>	P
<Q>	Q
<R>	R
<S>	S
<T>	T
<U>	U
<V>	V
<W>	W
<X>	X
<Y>	Y
<Z>	Z
<left-square-bracket>	[
<backslash>	\
<reverse-solidus>	\
<right-square-bracket>	]
<circumflex>	^
<circumflex-accent>	^
<underscore>	_
<low-line>	~
<grave-accent>	`
<a>	a
<b>	b
<c>	c
<d>	d
<e>	e
<f>	f
<g>	g
<h>	h
<i>	i
<j>	j
<k>	k
<l>	l
<m>	m

<n>	n
<o>	o
<p>	p
<q>	q
<r>	r
<s>	s
<t>	t
<u>	u
<v>	v
<w>	w
<x>	x
<y>	y
<z>	z
<left-brace>	{
<left-curly-bracket>	{
<vertical-line>	
<right-brace>	}
<right-curly-bracket>	}
<tilde>	~

**portability.** The ability of a programming language to compile successfully on different operating systems without requiring changes to the source code.

**positional parameter.** A parameter that must appear in a specified location relative to other positional parameters. *IBM.*

**precedence.** The priority system for grouping different types of operators with their operands.

**predefined macros.** Frequently used routines provided by an application or language for the programmer.

**prelinker.** A utility provided with Language Environment that can be used for application programs that are re-entrant, have external symbol names that are longer than what the linkage editor supports, or require DLL support. The prelinker must be run with all C++ applications. The prelinker is invoked before the linkage editor.

**preprocessor.** A phase of the compiler that examines the source program for preprocessor statements that are then executed, resulting in the alteration of the source program.

**printable character.** One of the characters included in the print character classification of the LC\_CTYPE category in the current locale. *X/Open.*

**private.** Pertaining to a class member that is only accessible to member functions and friends of that class.

**privilege.** See *file access permissions*.

**process.** (1) An instance of an executing application and the resources it uses. (2) An address space and single thread of control that executes within that address space, and its required system resources. A process is created by another process issuing the fork() function. The process that issues the fork() function is

known as the parent process, and the new process created by the `fork()` function is known as the child process. *X/Open. ISO.1.*

**process group.** A collection of processes that permits the signaling of related processes. Each process in the system is a member of a process group that is identified by the process group ID. A newly created process joins the process group of its creator. *IBM. X/Open. ISO.1.*

**process group ID.** The unique identifier representing a process group during its lifetime. A process group ID is a positive integer. (Under ISO only, it is a positive integer *that can be contained in a `pid_t`.*) A process group ID will not be reused by the system until the process group lifetime ends. *X/Open. ISO.1.*

**process group leader.** A process whose process ID is the same as its process group ID. *X/Open. ISO.1.*

**process ID.** The unique identifier representing a process. A process ID is a positive integer. (Under ISO only, it is a positive integer *that can be contained in a `pid_t`.*) A process ID will not be reused by the system until the process lifetime ends. In addition, if there exists a process group whose process group ID is equal to that process ID, the process ID will not be reused by the system until the process group lifetime ends. A process that is not a system process will not have a process ID of 1. *X/Open. ISO.1.*

**protected.** Pertaining to a class member that is only accessible to member functions and friends of that class, or to member functions and friends of classes derived from that class.

**pseudoterminal.** A special file in the `/dev` directory that effectively functions as a keyboard and a display device.

## Q

**qualified name.** Used to qualify a nonclass type name such as a member by its class name.

**Query Management Facility (QMF).** Pertaining to a query and report writing facility that enables a variety of tasks such as data entry, query building, administration, and report analysis.

**queue.** A sequence with restricted access in which elements can only be added at the back end (or bottom) and removed from the front end (or top). A queue is characterized by first-in, first-out behavior and chronological order.

## R

**radix character.** The character that separates the integer part of a number from the fractional part. *X/Open.*

**read-only file system.** A file system that has implementation-dependent characteristics restricting modifications. *X/Open. ISO.1.*

**real group ID.** The attribute of a process that, at the time of process creating, identifies the group of the user who created the process. This value is subject to change during the process lifetime, as describe in `setgid()`. *X/Open. ISO.1.*

**real user ID.** The attribute of a process that, at the time of process creation, identifies the user who created the process. This value is subject to change during the process lifetime, as described in `setuid()`. *X/Open. ISO.1.*

**reason code.** A code that identifies the reason for a detected error. *IBM.*

**redirection.** In the shell, a method of associating files with the input or output of commands. *X/Open.*

**reentrant.** The attribute of a program or routine that allows the same copy of a program or routine to be used concurrently by two or more tasks.

**refresh.** To ensure that the information on the user's terminal screen is up-to-date. *X/Open.*

**regular expression.** (1) A mechanism to select specific strings from a set of character strings. (2) A set of characters, meta-characters, and operators that define a string or group of strings in a search pattern. (3) A string containing wildcard characters and operations that define a set of one or more possible strings.

**regular file.** A file that is a randomly accessible sequence of bytes, with no further structure imposed by the system. *X/Open. ISO.1.*

**relation.** An unordered flat collection class that uses keys, allows for duplicate elements, and has element equality.

**relative path name.** The name of a directory or file expressed as a sequence of directories followed by a file name, beginning from the current directory. See *path name resolution. IBM.*

**root.** (1) A node that has no parent. All other nodes of a tree are descendants of the root. (2) In the AIX operating system, the user name for the system user with the most authority. *IBM.* (3) In the OS/2 operating system, the base directory.

**runtime library.** A compiled collection of functions whose members can be referred to by an application program during runtime execution. Typically used to refer to a dynamic library that is provided in object code, such that references to the library are resolved during the linking step. The runtime library itself is not statically bound into the application modules.

## S

**saved set-group-ID.** An attribute of a process that allows some flexibility in the assignment of the effective group ID attribute, as described in the `exec` family of functions and `setgid()`. *X/Open. ISO.1.*

**saved set-user-ID.** An attribute of a process that allows some flexibility in the assignment of the effective user ID attribute, as described in `exec` and `setuid()`. *X/Open. ISO.1.*

**scope.** (1) That part of a source program in which a variable is visible. (2) That part of a source program in which an object is defined and recognized.

**seconds since the epoch.** A value to be interpreted as the number of seconds between a specified time and the epoch. A Coordinated Universal Time name (specified in terms of seconds (*tm\_sec*), minutes (*tm\_min*), hours (*tm\_hour*), days since January 1 of the year (*tm\_yday*), and calendar year minus 1900 (*tm\_year*)) is related to a time represented as seconds since the epoch, according to the expression below.

If the year < 1970 or the value is negative, the relationship is undefined. If the year is equal to or greater than 1970 CE and the value is non-negative, the value is related to a Coordinated Universal Time name according to the expression:

```
tm_sec +  
tm_min * 60 +  
tm_hour * 3600 +  
tm_yday * 86400 +  
(tm_year-70) * 31536000 +  
((tm_year-69) / 4) * 86400
```

*X/Open. ISO.1.*

**sequence.** A sequentially ordered flat collection.

**sequential data set.** A data set whose records are organized on the basis of their successive physical positions, such as on magnetic tape. *IBM.*

**session.** A collection of process groups established for job control purposes. Each process group is a member of a session. A process is considered to be a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership; see `setsid()`. There can be multiple process groups in the same session. *X/Open. ISO.1.*

**session leader.** A process that has created a session; see `setsid()`. *X/Open. ISO.1.*

**shell.** A program that interprets sequences of text input as commands. It may operate on an input stream or it may interactively prompt and read commands from a terminal. *X/Open.*

This feature is provided as part of OS/390 UNIX Shell and Utilities feature licensed program.

**signal.** (1) A condition that may or may not be reported during program execution. For example, `SIGFPE` is the signal used to represent erroneous arithmetic operations such as a division by zero. (2) A mechanism by which a process may be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term *signal* is also used to refer to the event itself. *X/Open. ISO.1.* (3) In AIX operating system operations, a method of interprocess communication that simulates software interrupts. *IBM.*

**signal handler.** A function to be called when the signal is reported.

**slash.** The character `/`, also known as *solidus*. This character is named `<slash>` in the portable character set.

**S-name.** An external non-C++ name in an object module produced by compiling with the `NOLONGNAME` option. Such a name is up to 8 characters long and single case.

**source file.** A file that contains source statements for such items as high-level language programs and data description specifications. *IBM.*

**source program.** A set of instructions written in a programming language that must be translated to machine language before the program can be run. *IBM.*

**space character.** The character defined in the portable character set as `<space>`. The space character is a member of the space character class of the current locale, but represents the single character, and not all of the possible members of the class. *X/Open.*

**specifiers.** Used in declarations to indicate storage class, fundamental data type and other properties of the object or function being declared.

**stack storage.** Synonym for *automatic storage*.

**standard error.** An output stream usually intended to be used for diagnostic messages. *X/Open.*

**standard input.** (1) An input stream usually intended to be used for primary data input. *X/Open.* (2) The

primary source of data entered into a command. Standard input comes from the keyboard unless redirection or piping is used, in which case standard input can be from a file or the output from another command. *IBM.*

**standard output.** (1) An output stream usually intended to be used for primary data output. *X/Open.*  
(2) In the AIX operating system, the primary destination of data coming from a command. Standard output goes to the display unless redirection or piping is used, in which case standard output can go to a file or to another command. *IBM.*

**statement.** An instruction that ends with the character ; (semicolon) or several instructions that are surrounded by the characters { and }.

**static.** A keyword used for defining the scope and linkage of variables and functions. For internal variables, the variable has block scope and retains its value between function calls. For external values, the variable has file scope and retains its value within the source file. For class variables, the variable is shared by all objects of the class and retains its value within the entire program.

**stream.** (1) A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format. (2) A file access object that allows access to an ordered sequence of characters, as described by the ISO C standard. Such objects can be created by the `fopen` and `fdopen` functions, and are associated with a file descriptor. A stream provides the additional services of user-selectable buffering and formatted input and output. *X/Open.*

**stream buffer.** A stream buffer is a buffer between the ultimate consumer, ultimate producer, and the I/O Stream Library functions that format data. It is implemented in the I/O Stream Library by the `streambuf` class and the classes derived from `streambuf`.

**string.** A contiguous sequence of bytes terminated by and including the first null byte. *X/Open.*

**string constant.** Zero or more characters enclosed in double quotation marks.

**struct.** An aggregate of elements, having arbitrary types.

**structure.** A construct (a class data type) that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types. A structure can be used in all places a class is used. The initial projection is public.

**stub routine.** Within runtime libraries, contains the minimum lines of code required to locate a given routine at run time.

**subscript.** One or more expressions, each enclosed in brackets, that follow an array name. A subscript refers to an element in an array.

**supplementary group ID.** An attribute of a process used in determining the file access permissions. A process has up to `{NGROUPS_MAX}`—which is defined in `limits.h`—supplementary group IDs of a process are set up to the supplementary group IDs of the parent process when the process is created. Whether a process' effective group ID is included in or omitted from its list of supplementary group IDs is unspecified. *X/Open. ISO.1.*

**support.** In system development, to provide the necessary resources for the correct operation of a functional unit. *IBM.*

## T

**tab character.** A character that in the output stream indicates that printing or displaying should start at the next horizontal tabulation position on the current line. The tab is the character designated by '\t' in the C language. If the current position is at or past the last defined horizontal tabulation position, the behavior is unspecified. It is unspecified whether the character is the exact sequence transmitted to an output device by the system to accomplish the tabulation. *X/Open.*

This character is named `<tab>` in the portable character set.

**task.** (1) In a multiprogramming or multiprocessing environment, one or more sequences of instructions treated by a control program as an element of work to be accomplished by a computer. *ISO-JTC1. ANSI/ISO.*  
(2) A routine that is used to simulate the operation of programs. Tasks are said to be *nonpreemptive* because only a single task is executing at any one time. Tasks are said to be *lightweight* because less time and space are required to create a task than a true operating system process.

**task library.** A class library that provides the facilities to write programs that are made up of tasks.

**template.** A family of classes or functions with variable types.

**terminals.** Synonym for *leaves*.

**text file.** A file that contains characters organized into one or more lines. The lines must not contain NUL characters and none can exceed `{LINE_MAX}`—which is defined in `limits.h`—bytes in length, including the new-

line character. The term *text file* does not prevent the inclusion of control or other non-printable characters (other than NUL). *X/Open*.

**this.** A C++ keyword that identifies a special type of pointer in a member function, that references the class object with which the member function was invoked.

**thread.** The smallest unit of operation to be performed within a process. *IBM*.

**tilde.** The character ~. This character is named <tilde> in the portable character set.

**token.** The smallest independent unit of meaning of a program as defined either by a parser or a lexical analyzer. A token can contain data, a language keyword, an identifier, or other parts of language syntax. *IBM*.

**tokenization.** The process of parsing input into tokens.

**traceback.** A section of a dump that provides information about the stack frame, the program unit address, the entry point of the routine, the statement number, and the status of the routines on the call-chain at the time the traceback was produced.

**trap.** An unprogrammed conditional jump to a specified address that is automatically activated by hardware. A recording is made of the location from which the jump occurred. *ISO-JTC1*.

**tree.** A hierarchical collection of nodes that can have an arbitrary number of references to other nodes. A unique path connects every two nodes.

**truncate.** To shorten a value to a specified length.

**type.** The description of the data and the operations that can be performed on or by the data. See also *data type*.

**type conversion.** Synonym for *boundary alignment*.

**type specifier.** Used to indicate the data type of an object or function being declared.

## U

**unblock a thread.** A blocked thread is a thread that is waiting on a specific condition to continue execution. A thread can be unblocked by signalling the condition blocking it.

**undefined behavior.** Referring to a program or function that may produce erroneous results without warning because of its use of an indeterminate value, or because of erroneous program constructs or erroneous data.

**underflow.** (1) A condition that occurs when the result of an operation is less than the smallest possible nonzero number. (2) Synonym for arithmetic underflow, monadic operation. *IBM*.

**union.** (1) In the C or C++ language, a variable that can hold any one of several data types, but only one data type at a time. *IBM*. (2) For bags, there is an additional rule for duplicates: If bag P contains an element *m* times and bag Q contains the same element *n* times, then the union of P and Q contains that element *m+n* times.

**unspecified behavior.** Referring to a program or function that may produce erroneous results without warning because of erroneous program constructs or erroneous data.

**user ID.** A non-negative integer that is used to identify a system user. (Under ISO only, a nonnegative integer, which can be contained in an object of type *uid\_t*.) When the identity of a user is associated with a process, a user ID value is referred to as a real user ID, an effective user ID, or (under ISO only, and there optionally) a saved set-user ID. *X/Open*. *ISO.1*.

**user name.** A string that is used to identify a user. *ISO.1*.

**user prefix.** In an MVS environment, the user prefix is typically the user's logon user identification.

## V

**variable.** In programming languages, a language object that may take different values, one at a time. The values of a variable are usually restricted to a certain data type. *ISO-JTC1*.

**visible.** Visibility of identifiers is based on scoping rules and is independent of *access*.

**VSAM (Virtual Storage Access Method).** An IBM licensed program that controls communication and the flow of data in an SNA network. It provides single-domain, multiple-domain, and interconnected network capability. *IBM*.

## W

**while statement.** A looping statement that contains the keyword *while* followed by an expression in parentheses (the condition) and a statement (the action). *IBM*.

**white space.** (1) Space characters, tab characters, form-feed characters, and new-line characters. (2) A sequence of one or more characters that belong to the space character class as defined via the *LC\_CTYPE*

category in the current locale. In the POSIX locale, white space consists of one or more blank characters (space and tab characters), new-line characters, carriage-return characters, form-feed characters, and vertical-tab characters. *X/Open*.

**wide character.** A character whose range of values can represent distinct codes for all members of the largest extended character set specified among the supporting locales.

**wide-character code.** An integral value corresponding to a single graphic symbol or control code. *X/Open*.

**wide-character string.** A contiguous sequence of wide-character codes terminated by and including the first null wide-character code. *X/Open*.

**wide-oriented stream.** See *orientation of a stream*.

**working directory.** Synonym for *current working directory*.

**wrapping of output.** The automatic disposition of a line of output onto two or more lines, necessitated by the limitation of the width of the device or file to which output is directed.

**write.** (1) To output characters to a file, such as standard output or standard error. Unless otherwise stated, standard output is the default output destination for all uses of the term *write*. *X/Open*. (2) To make a permanent or transient recording of data in a storage device or on a data medium. *ISO-JTC1. ANSI/ISO*.

## X

**X/Open Socket.** The application program interface (API) type of socket as defined by X/Open in XPG4.2.





---

# Index

## Special Characters

\_\_ussos.h header file 50

## Numerics

24malc() 1795

4kmalc() 1795

## A

a64l() library function 124

abend 71

    user 71

abnormal program termination 71

abort() library function 71

*See also* abend

abort() library function 71

abs() library function 73

absolute value 73

    decimal data type 268

    floating-point data type 338

    integer argument 73

    long integer 733

acc parameter for fopen() 419

accept\_and\_recv() library function 78

accept() library function 75

access mode, fopen() 417

access() library function 82

acos() library function 85

acosh() library function 87

address

    socket peer 1253

address, host 531

advance() library function 88

AF\_INET domain

    example 131

    servers 215

    socket descriptor created in 128

AF\_UNIX domain

    example 132

    servers 215

    socket descriptor created in 129

aio.h header file 23

alarm() library function 102

\_ALL\_SOURCE feature test macro 19

alloca() library function 105

\_ALL\_SOURCE\_NO\_THREADS feature test macro 20

\_ALTER\_RESOURCE symbolic constant 170

AMODE

    switching 371

\_\_amrc structure

    macros 44

arccosine library function 85

arcsine library function 108

arctangent library function 113

arguments

    accessing 1670

arpa/inet.h header file 23

asctime() library function 106

asin() library function 108

asinh() library function 110

assert.h header file 23

assert() macro 111

assertion diagnostic 111

assigning buffers 1208

asynchronous

    signal catching 1302

    signal, wait for an 1356

at (@) character, mapping to 61

atan() library function 113

atan2() library function 113

atanh() library function 115

atexit() library function 116

\_\_atof() library function 119

\_\_atof\_l() library function 120

atof() library function 121

atoi() library function 122

atol() library function 123

ATTACH 1490

attributes, terminal 1515

audit flags 165, 341

    change by file descriptor 341

    change by path 165

## B

basename() library function 125

baud rate

    input

        determining 156

        termios 161

    output

        determining 159

        termios 163

bcmp() library function 126

bcopy() library function 127

bessel library functions

    first kind 726

    second kind 1793

binary

    files 418

    search 137

bind() library function 128

- blank character attribute 706
- blank character, wide 718
- blksize parameter 419
- BPX\_ACCT\_DATA environment variable 323
- \_BPX\_JOBNAME environment variable 323
- break condition 1529
- brk() library function 133
- broadcast, unblock a thread 943
- bsd\_signal() library function 135
- bsearch() library function 137
- buffers 1208
  - assigning 1208
  - BUFSIZ macro 43
  - comparing 811
  - copying 813, 814
  - data stored in 1093
  - flushing 385
  - format and print data 1683
  - receive data and store in 1103
  - receive messages and store in 1106, 1110
  - searching 809
  - setting characters 816
- BUFSIZ macro 43
- built-in library functions 64
  - list of 64
- bytesseek parameter in fopen() 419
- bzero() library function 140

## C

### C

- macros 1795
- calloc() library function 141
- cancel a thread 936
- cancelability
  - point, establishing 1047
  - PTHREAD\_INTR\_ASYNCHRONOUS type 936
  - PTHREAD\_INTR\_CONTROLLED type 936
  - PTHREAD\_INTR\_DISABLE type 936
  - PTHREAD\_INTR\_ENABLE type 936
- canonical input processing 1537
- catclose() library function 143
- category argument of setlocale() 1241
- catgets() library function 144
- catopen() library function 146
- cbrt() library function 148
- cclass() library function 149
- cds() library function 151
- cdump() library function 152
- ceil() library function 153
- ceiling of value, determining 153
- \_\_certificate() library function 154
- cfgetispeed() library function 156
- cfgetospeed() library function 159
- cfsetispeed() library function 161
- cfsetospeed() library function 163
- character
  - classification with ctype.h 24
  - classifying 694
  - converting case 1592, 1604
  - finding in a string 1444
  - multibyte
    - conversion using mbrtowc() 797
    - conversion using mbstowcs() 804
    - conversion using mbtowc() 806
    - length 791, 794
  - property 720
  - property classification 1753
  - reading with fgetc() 388
  - reading with getc() and getchar() 499
  - reading with getwc() 627
  - reading with getwchar() 629
  - setting 816
  - testing 694
  - ungetting 1655, 1658
  - writing with fputc() 448
  - writing with fputwc() 452
  - writing with putc() and putchar() 1052
- \_\_ characters, mapping from \_\_ 61
- \_\_ characters, mapping to \_\_ 61
- chaudit() library function 165
- chdir() library function 167
- \_\_check\_resource\_auth\_np() library function 169
- CheckSchEnv() library function 172
- CheckWLMSchEnv library function 172
- child process 422
- chmod() library function 174
- chown() library function 177
- chpriority() library function 180
- chroot() library function 182
- CICS (Customer Information Control System)
  - cics.h header file 23
  - verify running 708
- classifying characters 694
- cleanup thread handler 939, 941
- clear error indicators 187
- clearenv() library function 184
- clearerr() library function 187
- client
  - incoming requests, preparing server for 752
- clock ticking in times() library function 1572
- clock() library function 189
- CLOCKS\_PER\_SEC 189
- close() library function 191
- closedir() library function 194
- closelog() library function 196
- closing
  - files 348
  - streams 348
- clrmmemf() library function 197
  - associated macros 44

- \_\_cnvblk() library function 199
- code set conversion utilities
  - iconv
    - usage 654
  - iconv.h header file 30
- coded character set
  - definition 1810
- collate.h header file 23
- collating elements 204
  - See also locale
  - get next 554
  - maximum 788
  - multicharacter 710
  - wide character 632
- collequiv() library function 200
- collorder() library function 202
- collrange() library function 204
- colltostr() library function 206
- commands, invoking from library function 1488
- compare
  - buffers 811
  - strings 1418, 1420, 1424, 1440
- comparison functions, cs() 239
- compile() library function 208
- concatenation
  - strings 1414, 1438
- concurrent access 353
- condition
  - attribute object
    - destroy a 955
    - initialize a 959
  - variable
    - wait on a 952
    - wait on for a limited time 950
- configuration
  - system 1483
- configuration variable 896
- confstr() library function 212
- connect() library function 214
- connection
  - duplex, shutting down 1293
- connection between sockets 214
- connection request 75
- ConnectServer() library function 219
- ConnectWorkMgr() library function 221
- \_\_console() library function 223
- ContinueWLMWorkUnit library function 225
- ContinueWorkUnit() library function 225
- control block information 402
- \_\_CONTROL\_RESOURCE symbolic constant 170
- controlling terminal, path name 246
- conversions
  - character
    - multibyte to wide with mbrtowc() 797
    - multibyte to wide with mbstowcs() 804
    - multibyte to wide with mbtowc() 806
    - string to double 1456

- conversions (*continued*)
  - character (*continued*)
    - to lowercase 1592, 1604
    - to uppercase 1592, 1604
  - character string to double 1456
  - code set 654
  - date and time 1445
  - floating-point numbers to integers and fractions 837
  - specifier
    - argument in fscanf() and scanf() library function 466
    - fscanf() and scanf() library functions 465
    - used by strptime() 1432
    - used by strtptime() 1445
  - string to unsigned integer 1462
  - string, multibyte to wide 802
  - strings to integer values 122
  - time structure to string 106
  - time to character string 250
  - wide character to multibyte 1697, 1764, 1766, 1768, 1770, 1772
- \_\_convert\_id\_np() library function 226
- copying
  - bytes 813, 814
  - strings 1422, 1442
- cos() library function 229
- cosh() library function 231
- cosine
  - calculating 229
  - hyperbolic, calculating 231
- \_CPCN\_NAMES symbolic constant 1517
- \_CPCN\_TABLES symbolic constant 1517
- cpio.h header file 23
- creat() library function 233
- CreateWorkUnit() library function 236
- creating
  - socket 1371
  - socket pair 1375
  - symbolic link to path name 1479
  - symbolic links, external 336
  - temporary file 1582, 1584
  - thread key identifiers 981
  - threads 963
- crypt() library function 238
- \_CS\_PATH symbolic constant 212
- \_CS\_SHELL symbolic constant 212
- cs() library function 239
- csid() library function 240
- csnap() library function 242
- csp.h header file 24
- \_\_csplist macro 24, 243
- ctdli() library function 244
- ctermid() library function 246
- ctest.h header file 24
- ctest() library function 248

- ctime() library function 250
- ctrace() library function 252
- ctype.h header file 24
- \_\_CURRENT clrmemf() argument 197
- current file position, changing 474
- current host address 539
- \_\_CURRENT\_LOWER clrmemf() argument 197
- \_\_CURRENT\_LOWER macro 44
- \_\_CURRENT macro 44
- current terminal
  - \_\_getlogin1() 552
  - getlogin() 550
- cuserid() library function 254

## D

- data
  - buffers, stored in 1093
  - receiving 1103
  - sending on socket 1190
  - store in buffers 1103
- data items
  - reading 456
  - writing 495
- data set
  - host information 1224
  - network information 1251
  - network services, opening 1267
  - protocol, opening 1259
- data types
  - fixed-point decimal 24
  - floating-point 28, 462
  - limits 32
- database
  - group 523, 525
  - user 587, 589
- datagram
  - sending on socket 1178
- date conversion 1445
- date format 1711
- DBCS (Double-Byte Character Support)
  - shift state information 477
- dbm\_clearerr() library function 255
- dbm\_close() library function 256
- dbm\_delete() library function 257
- dbm\_error() library function 258
- dbm\_fetch() library function 259
- dbm\_firstkey() library function 260
- DBM\_INSERT symbolic constant 266
- dbm\_nextkey() library function 262
- dbm\_open() library function 264
- DBM\_REPLACE symbolic constant 266
- dbm\_store() library function 266
- deadlocks 354
- Debug Tool library function 248
- debugging
  - ctest.h 24
    - with \_\_errno2(), reason codes 319
- decabs() library function 268
- decchk() library function 269
- decfix() library function 271
- decimal
  - data type
    - absolute value 268
    - preferred sign 271
    - valid types 269
  - fixed-point operations in decimal.h 24
- decimal host address
  - from network number 666
- decimal.h header file 24
- delete
  - mutex object 986
  - VSAM records 359
- delete a mutex object 986
- DeleteWLMWorkUnit library function 272
- DeleteWorkUnit() library function 272
- deleting VSAM records 359
- descriptor, socket 128
- destroying
  - condition variable attribute objects 955
  - condition variables 945
  - mutex attribute objects 996
  - thread attributes object 918
- destructor routine 981
- detach a thread 965
- detachstate attribute
  - getting 920
  - setting 929
- device ID
  - lstat() 778
  - stat() 1404
- diagnostic error messages
  - specifying 111
- difftime() library function 273
- directories (HFS)
  - closing 194
  - entry removal 1660
  - mode
    - changing 344
  - opening 877, 880
  - reading with readdir() 1086, 1089
  - removing 1147
  - renaming 1136
  - repositioning 1141
  - rewinding 1141
  - working 508
- directory mode, changing 174
- directory operations 24
- dirent.h header file 24
- dirname() library function 275

- div\_t structure 277
- div() library function 277
- division 277
  - integral 740
- \_\_dlght() function 67
- dll.h header file 25
- dllfree() library function 278
- dllload() library function 280
- dllqueryfn() library function 282
- dllqueryvar() library function 284
- DLLs (Dynamic Link Libraries)
  - explicit use 278, 280
  - freeing 278
  - loading 280
  - obtaining function pointers 282
  - obtaining variable pointers 284
- domain
  - servers in the AF\_INET 215
  - servers in the AF\_UNIX 215
- drand48() library function 286
- dtconv structure 34
- dtconv structure, address 754
- dumps, formatted 152
- dup() library function 288
- dup2() library function 290
- duplex connection 1293
- dynalloc() library function 292
- dynamic
  - allocations in dynit.h 25
  - data set allocation 292
  - data set deallocation 299
  - function call 369, 382
- dynfree() library function 299
- dyninit() library function 301
- dynit.h header file 25
- \_\_dyn\_t structure
  - dynalloc() 292
  - elements 293
  - initialization 301

## E

- EACCES 339, 361
- EAGAIN 557, 1057
- EBADF 339, 557, 702, 1057
- EBADMSG 557
- EBUSY 340
- ECONNRESET 1183
- ecvt() library function 303
- \_\_EDC\_COMPAT environment variable 385, 474, 1658
- \_\_EDC\_UMASK\_DFLT environment variable 1647
- effective ID
  - group 514, 1220
  - user 519, 1278

- EINTR 557, 1057
- EINVAL 340, 361, 557, 687, 1057
- ELOOP 340, 362
- EMVSBADCHAR symbolic constant 27
- EMVSCATLG symbolic constant 27
- EMVSCVAF symbolic constant 27
- EMVSDYNALC symbolic constant 27
- EMVSERR symbolic constant 27
- EMVSNORTL symbolic constant 27
- EMVSNOTUP symbolic constant 27
- EMVSPARM symbolic constant 27
- EMVSPATHOPTS symbolic constant 27
- EMVSPFSFILE symbolic constant 27
- EMVSPFSPERM symbolic constant 27
- EMVSSAF2ERR symbolic constant 27
- EMVSSAFEXTRERR symbolic constant 27
- EMVSTODNOTSET symbolic constant 27
- ENAMETOOLONG 340, 361
- ENAMETOOLONG symbolic constant 27
- encrypt() library function 305
- endgrent() library function 307
- endhostent() library function 308
- ending a process or program 71
- endnetent() library function 309
- endprotoent() library function 310
- endpwent() library function 311
- endservent() library function 312
- endutxent() library function 313
- ENFILE symbolic constant 27
- ENODEV 687
- ENODEV symbolic constant 27
- ENOENT 339, 361
- ENOENT symbolic constant 27
- ENOEXEC symbolic constant 27
- ENOLCK symbolic constant 27
- ENOMEM symbolic constant 27
- ENOSPC symbolic constant 27
- ENOSR 1058
- ENOSTR 557, 1058
- ENOSYS symbolic constant 27
- ENOTDIR 340, 361
- ENOTDIR symbolic constant 27
- ENOTEMPTY symbolic constant 27
- ENOTTY symbolic constant 27
- env.h header file 25
- environ 1054
- environment
  - table 515
  - variables 324, 515
    - \_\_EDC\_COMPAT 385, 474, 1658
    - add, delete or change 1215
    - clearing 184
    - setting 25
- ENXIO 1058
- ENXIO symbolic constant 27

- EOF (end of file)
  - clearing 1139
  - flag 365
  - indicator reset 187
- eof macro 43
- EPERM 340, 361
- EPERM symbolic constant 27
- EPIPE or EIO 1058
- EPIPE symbolic constant 27
- erand48() library function 314
- ERANGE 1058
- erf() library function 316
- erfc() library function 316
- EROFS symbolic constant 27
- \_\_err2ad() library function 318
- errno
  - in perror() 903
  - values 25
- errno's
  - ECONNRESET 1183
  - ENXIO 1058
- errno values 66
- errno.h header file 25
- \_\_errno2() library function 319
- errnos
  - EAGAIN 557
  - EBADF 557
  - EBADMSG 557
  - EINTR 557
  - EINVAL 557
  - ENOSTR 557
- error
  - functions 316
  - in files 367
  - indicator 367
  - indicator, clearing 1139
  - messages
    - diagnostic, specifying 111
    - pointer to 1427
    - printing 903
- ESDS (Entry-Sequenced Data Set) 493
- ESDS (Entry-Sequenced Data Set)
  - use of 493
- ESPIPE symbolic constant 27
- ESRCH symbolic constant 27
- establish
  - cancelability point 1047
  - cleanup thread handler 941
- \_\_etoa() library function 320
- \_\_etoa\_l() library function 321
- examples
  - machine-readable xxxi
  - naming of xxxi
  - softcopy xxxi
- exception handling
  - assert() library function 111

- exception handling (*continued*)
  - clearerr() library function 187
  - in C++ 1560, 1654
  - in C++ 1252
  - perror() library function 903
  - signal.h header file 41
- EXDEV symbolic constant 27
- exec family of functions
  - described 322
  - execl() library function 322
  - execle() library function 322
  - execlp() library function 322
  - execv() library function 322
  - execve() library function 322
  - execvp() library function 322
- \_exit() library function 332
- exit a thread 969
- EXIT\_FAILURE macro 45
- EXIT\_FAILURE macro in stdlib.h 330
- EXIT\_SUCCESS macro 45
- EXIT\_SUCCESS macro in stdlib.h 330
- exit() library function 330
- exiting a program 330
- exp() library function 334
- expm1() library function 335
- exponential functions 334
- external
  - symbolic link, create 336
- extlink\_np() library function 336
- ezbzdnc.h header file 28

## F

- fabs() library function 338
- fattach() library function 339
- fchmod() library function 341
- fchdir() library function 343
- fchmod() library function 344
- fchown() library function 346
- fclose() library function 348
- fcntl.h header file 28
- fcntl() library function 350
- fcvt() library function 358
- fdelrec() library function 359
- fdetach() library function 361
- fdopen() library function 363
- feature test macro 17
- features.h header file 28
- FECB (fetch control block) 382
- feof() library function 365
- ferror() library function 367
- fetch
  - a module 369
  - control block 382
  - fetchable module, program flow 370
  - without FETCHABLE 370

- fetch() library function 369
  - alternatives under C++ 372
  - examples of alternatives under C++ 375
- FETCHABLE preprocessor directive 370
- fetchep() library function 382
- fflush() library function 385
- ffs() library function 387
- fgetc() library function 388
- fgetpos() library function 390
- fgets() library function 392
- fgetwc() library function 394
- fgetws() library function 396
- FIFO 874
  - special files
    - creating 820, 823
- file system
  - mount a 838
  - mounted, information 1756
  - removing 1649
  - status 1789
- FILE type 43
- FILENAME\_MAX macro 43
- fileno() library reference 399
- files
  - changing mode 177, 344
  - closing 191
  - descriptor 872
    - associating with streams 363
    - controlling 350
    - duplicate 288, 290
    - open stream 399
  - descriptor flags 351
  - errors in 187
  - locking 353, 355
  - maximum opened 43
  - name
    - temporary 1584
  - name length 43
  - names
    - length 43
  - offset 872, 1080, 1780
  - opening 417
  - positioning 390, 474, 477, 485, 1139
  - renaming 1136
  - status flags 351
  - time access 1664
  - type=record used with putwc() or putwchar() 1066
  - writing to 1780
- fixed-point decimal
  - decimal.h 24
- flags
  - audit 165
  - EOF 365
  - file descriptor 352
  - open
    - append mode 352
    - asynchronous update 352
- flags (*continued*)
  - open (*continued*)
    - blocking 352
    - extracting 353
    - file access mode 353
    - synchronous update 352
- fldata\_t data structure elements 403
- fldata() library function 402
  - associated macros 44
- float.h header file
  - constants defined in 28
  - defined 28
- floating-point
  - absolute value 338
  - break up value 837
  - breaking down value 462
  - data type 28
- flocate() library function 405
  - associated macros 44
- floor() library function 408
- flushing, buffers 385
- fmod() library function 410
- fmtmsg.h header file 29
- fmtmsg() library function 412
- fnmatch.h header file 29
- fnmatch() library function 415
- FOPEN\_MAX macro 43
- fopen() library function
  - maximum simultaneous files 43
- fork() library function 422
- format specification
  - fprintf family 436
  - fscanf() and scanf() library functions 465
- format specification for fprintf() family 437
- formatted I/O 436
- formatted time 1432
- fortrc() library function 426
- \_\_4kmalc() library function 1795
- fpathconf() library function 427
- fpos\_t type in stdio.h file 43
- fprintf() library function 436
- fputc() library function 448
- fputs() library function 450
- fputwc() library function 452
- fputws() library function 454
- fread() library function 456
- free() library function 458
- freeing storage 458
- freopen() library function 460
- frexp() library function 462
- fscanf() library function 464
- fseek() library function 474
- fsetpos() library function 477
- fstat() library function 479
- fstatvfs() library function 481

- fsync() library function 483
- \_\_ftchep entry point in fetchep() 382
- ftell() library function 485
- ftime() library function 487
- ftok() library function 488
- ftruncate() library function 489
- FTW\_CHDIR symbolic constant 861
- FTW\_D symbolic constant 491, 861
- FTW\_DEPTH symbolic constant 861
- FTW\_DNR symbolic constant 491, 861
- FTW\_DP symbolic constant 861
- FTW\_F symbolic constant 491, 861
- FTW\_MOUNT symbolic constant 861
- FTW\_NS symbolic constant 491, 862
- FTW\_PHYS symbolic constant 861
- FTW\_SL symbolic constant 491, 862
- FTW\_SLN symbolic constant 862
- ftw.h header file 29
- ftw() library function 491
- functions
  - arguments 1670
- fupdate() library function 493
- fwrite() library function 495

## G

- gamma() library function 497
- gcv() library function 498
- \_\_gderr() function 67
- get
  - condition variable attribute object 957
  - file position, fgetpos() 390
- GETALL symbolic constant 1170
- getc() library function 499
- getchar() library function 499
- \_\_getclientid() library function 503
- getclientid() library function 501
- getcontext() library function 505
- getcwd() library function 508
- getdate() library function 510
- getdtablesize() library function 513
- getegid() library function 514
- \_\_getenv() library function 517
- getenv() library function 515
- geteuid() library function 519
- getgid() library function 521
- getgrent() library function 307
- getgrgid() library function 523
- getgrnam() library function 525
- getgroups() library function 527
- getgroupsbyname() library function 529
- gethostbyaddr() library function 531
- gethostbyname() library function 534
- gethostent() library function 537
- gethostid() library function 539
- gethostname() library function 540
- getibmopt() library function 542
- getibmsockopt() library function 543
- \_\_getipc() library function 546
- getitimer() library function 548
- getlogin() library function 550
- \_\_getlogin1() library function 552
- getmccoll() library function 554
- getmsg() library function 555
- GETNCNT symbolic constant 1170
- getnetbyaddr() library function 558
- getnetbyname() library function 560
- getnetent() library function 562
- getopt() library function 564
- getpagesize() library function 566
- getpass() library function 567
- getpeername() library function 568
- getpgid() library function 570
- getpgrp() library function 571
- GETPID symbolic constant 1170
- getpid() library function 573
- getpmsg() library function 555
- getppid() library function 576
- getpriority() library function 578
- getprotobyname() library function 580
- getprotobyndnumber() library function 582
- getprotoent() library function 584
- getpwent() library function 311
- getpwnam() library function 587
- getpwuid() library function 589
- getrlimit() library function 591
- getrusage() library function 594
- gets() library function 595
- getservbyname() library function 597
- getservbyport() library function 599
- getservernt() library function 601
- getsid() library function 603
- getsockname() library function 604
- getsockopt() library function 606
- getstablesz() library function 611
- getsubopt() library function 612
- getsyntax() library function 614
- gettimeofday() library function 616
- getuid() library function 618
- \_\_GET\_USERID symbolic constant 226
- getutxent() library function 620
- getutxid() library function 622
- getutxline() library function 624
- \_\_GET\_UUID symbolic constant 226
- GETVAL symbolic constant 1169
- getw() library function 626
- getwc() library function 627
- getwchar() library function 629
- getwd() library function 631
- getwmccoll() library function 632



- GETZCNT symbolic constant 1170
- givesocket() library function 633
- GLOB\_ABORTED symbolic constant 638
- GLOB\_APPEND symbolic constant 637
- GLOB\_DOOFFS symbolic constant 637
- GLOB\_ERR symbolic constant 637
- GLOB\_MARK symbolic constant 637
- GLOB\_NOCHECK symbolic constant 637
- GLOB\_NOESCAPE symbolic constant 637
- GLOB\_NOMATCH symbolic constant 638
- GLOB\_NOSORT symbolic constant 637
- GLOB\_NOSPACE symbolic constant 638
- glob.h header file 29
- glob() library function 636
- globfree() library function 639
- gmtime() library function 640
- grantpt() library function 642
- group database, getgrgid() library function 523
- group database, getgrnam() library function 525
- group ID
  - effective 514
  - job control 1254
  - lstat() 778
  - process 571
  - real 521
  - setting 1213
  - stat() 1404
  - supplementary 527, 529
- grp.h header file 29

## H

- \_\_ftp 29
- handle, character property class 1753
- handling interrupt signals 1330
- hcreate() library function 643
- hdestroy() library function 644
- header files
  - \_\_ftp.h header file 29
  - aio.h header file 23
  - arpa/inet.h header file 23
  - assert.h header file 23
  - cics.h header file 23
  - collate.h header file 23
  - cpio.h header file 23
  - csp.h header file 24
  - ctest.h header file 24
  - ctype.h header file 24
  - decimal.h header file 24
  - dirent.h header file 24
  - dll.h header file 25
  - dynit.h header file 25
  - env.h header file 25
  - errno.h header file 25
  - ezbzsdc.h header file 28
  - fcntl.h header file 28

- header files (*continued*)
  - features.h header file 28
  - float.h header file 28
  - fmtmsg.h header file 29
  - fnmatch.h header file 29
  - ftw.h header file 29
  - glob.h header file 29
  - grp.h header file 29
  - iconv.h header file 30
  - ims.h header file 30
  - langinfo.h header file 30
  - lc\_core.h header file 31
  - leawi.h header file 31
  - libgen.h header file 31
  - limits.h header file 32
  - localdef.h header file 33
  - locale.h header file 33
  - math.h header file 35
  - memory.h header file 36
  - monetary.h header file 36
  - msgcat.h header file 36
  - mtf.h header file 37
  - ndbm.h header file 37
  - net/if.h header file 37
  - net/rtroute.h header file 37
  - netdb.h header file 37
  - netinet/in.h header file 37
  - new.h header file 37
  - nl\_langinfo.h header file 38
  - nl\_types.h header file 38
  - nlist.h header file 37
  - poll.h header file 38
  - pthread.h header file 38
  - re\_comp.h header file 39
  - regex.h header file 40
  - regexp.h header file 40
  - rexec.h header file 40
  - search.h header file 40
  - setjmp.h header file 40
  - signal.h header file 41
  - spawn.h header file 42
  - spc.h header file 42
  - stdarg.h header file 42
  - stddef.h header file 42
  - stdio.h header file 43
  - stdlib.h header file 45
  - string.h header file 46
  - strings.h header file 46
  - stropts.h header file 46
  - sys/\_\_getip.h header file 47
  - sys/file.h header file 47
  - sys/ioctl.h header file 47
  - sys/ipc.h header file 47
  - sys/mman.h header file 47
  - sys/modes.h header file 47
  - sys/msg.h header file 48

## header files (*continued*)

- sys/resource.h header file 48
- sys/sem.h header file 48
- sys/shm.h header file 48
- sys/socket.h header file 48
- sys/statvfs.h header file 49
- sys/time.h header file 49
- sys/timeb.h header file 49
- sys/ttydev.h header file 49
- sys/uio.h header file 50
- sys/un.h header file 50
- sys/wlm.h header file 51
- syslog.h header file 47
- tar.h header file 51
- terminat.h header file 51
- time.h header file 51
- ucontext.h header file 52
- uheap.h header file 52
- ulimit.h header file 53
- unexpected.h header file 52
- utmpx.h header file 54
- varargs.h header file 54
- variant.h header file 54
- wchar.h header file 54
- wcstr.h header file 55
- wctype.h header file 56
- wordexp.h header file 56
- xti.h header file 56

## headers

- ioctl() 687
- \_\_heaprpt() library function 645
- \_\_h\_errno() function 67

## HFS (Hierarchical File System)

- adding system 838

## host address 531

## host byte order

- short integer translated to 871
- translating long integer to 870

## host information data set

- opening 1224

## host information data sets 308

## host name 534

## host name entry 537

## hsearch() library function 647

## htonl() library function 648

## htons() library function 649

## HUGE\_VAL macro 36

## hyperbolic cosine, calculating 231

## hyperbolic sine, calculating 1362

## hyperbolic tangent, calculating 1502

## hypot() library function 650

## I

### I/O

- controlling devices 1762

## I/O (*continued*)

- error testing 367
- errors 187
- opening files 417
- write to a file 1780

## ibmsflush() library function 652

## iconv\_close() library function 657

## iconv\_open() library function 658

## iconv.h header file 30

## iconv() library function 654

## \_ICONV\_UCS2 environment variable 658

## \_ICONV\_UCS2\_PREFIX environment variable 658

## if.h header file 37

## ilogb() library function 659

## importing functions and variables 280

## ims.h header file 30

## in.h header file 37

## index() library function 660

## indicators, error 187

## inet\_addr() library function 661

## inet\_pton() library function 663

## inet\_makeaddr() library function 664

## inet\_netof() library function 665

## inet\_network() library function 666

## inet\_ntoa() library function 667

## initgroups() library function 669

## initialization

- of a mutex 988
- of a mutex attribute object 1004
- of condition attribute objects 959
- of condition variables 947
- of thread attributes objects 927

## strings 1442

## initstate() library function 670

## inode

- lstat() 778
- stat() 1404

## input

- baud rate
- determining 156
- termios 161

## insque() library function 671

## integer

- division 740
- long absolute value 733
- pseudorandom 1077
- short translated to host byte order 871
- translating 649
- translating long to host byte order 870
- unsigned short 649
- wide 715

## internationalization header file 33

## Internet address

- host 664
- decimal 667
- from network number 665

- Internet address (*continued*)
  - into network byte order 661
- interoperability
  - fortrc() 423
  - vfork() 1677
- interrupt signal 1330
- intrinsic functions 64
  - See *also* built-in library functions
- invoke a function once 1013
- invoking commands from a library function 1488
- ioctl() library function 672
- \_IOFBF macro 44
- \_IOLBF macro 44
- \_IONBF macro 44
- IPC\_CREAT symbolic constant 847, 1172, 1290
- IPC\_EXCL symbolic constant 847, 1172, 1290
- IPC\_NOWAIT symbolic constant 850, 852, 854, 855, 1176
- IPC\_PRIVATE symbolic constant 1172, 1290
- IPC\_RMID symbolic constant 845, 1171, 1288
- IPC\_SET symbolic constant 845, 1170, 1287
- IPC\_STAT symbolic constant 845, 1170, 1287
- IPCQALL symbolic constant 546
- IPCQMSG symbolic constant 546
- IPCQOVER symbolic constant 546
- IPCQSEM symbolic constant 546
- IPCQSHM symbolic constant 546
- \_\_ipdbcs() library function 688
- \_\_ipdspix() library function 689
- \_\_iphost() library function 690
- \_\_ipmsgc() library function 691
- \_\_ipnode() 692
- \_\_iptcpn() library function 693
- isalnum() library function 694
- isalpha() library function 694
- isascii() library function 697
- isastream() library function 702
- isatty() library function 703
- isblank() library function 706
- iscics() library function 708
- iscntrl() library function 694
- isdigit() library function 694
- isgraph() library function 694
- islower() library function 694
- ismccollet() library function 710
- isnan() library function 712
- \_\_isPosixOn() library function 713
- isprint() library function 694
- ispunct() library function 694
- isspace() library function 694
- isupper() library function 694
- iswalnum() library function 715
- iswblank() library function 718
- iswcntrl() library function 715
- iswctype() library function 720

- iswdigit() library function 715
- iswgraph() library function 715
- iswlower() library function 715
- iswprint() library function 715
- iswpunct() library function 715
- iswspace() library function 715
- iswupper() library function 715
- iswxdigit() library function 715
- isxdigit() library function 694
- ITIMER\_PROF symbolic constant 548, 1232
- ITIMER\_REAL symbolic constant 548, 1232
- ITIMER\_VIRTUAL symbolic constant 548, 1232

## J

- j0() library function 726
- j1() library function 726
- jn() library function 726
- job control process group ID 1254
- JoinWorkUnit() library function 723
- rand48() library function 724

## K

- key identifier
  - create thread-specific data key 981
  - get the specific value 971, 974
  - set the specific value 1043
- keyword parameters 419
- kill() library function 728
- killpg() library function 731
- kind attribute
  - getting from a mutex attribute object 998
  - setting from a mutex attribute object 1006
- KSDS (Key Sequenced Data Set) 493

## L

- L\_ctermid macro 44
- L\_tmpnam macro 44
- l64a() library function 782
- labs() library function 733
- langinfo.h header file 30
- language collation string comparison 1705
- Language Environment
  - effect of setlocale() 1241
- lc\_core.h header file 31
- LC\_CTYPE locale variable 396
- LC\_MONETARY locale variable 1430
- LC\_SYNTAX locale variable 614, 1429
- lchown() library function 734
- lcong48() library function 736
- lconv structure, elements of 33
- ldexp() library function 738
- ldiv() library function 740

- LeaveWorkUnit() library function 742
- LEAWI\_INCLUDED macro 31
- leawi.h header file 31
- length function 1436
- lfind() library function 743
- lgamma() library function 744
- libgen.h header file 31
- library
  - library function, calloc() 141
  - release 746
- \_\_librel() library function 746
- limits
  - resource 32
- limits.h header file 32
- line
  - mode 1537
  - reading with fgets() 392
  - reading with fgetwc() 394
  - reading with fgetws() 396
  - writing with puts() 1059
- link count 749
- link() library function 749
- listen() library function 752
- load module
  - fetchep() library function 382
  - fetching 369
  - release() library function 1130
- \_\_loc1() function 67
- local network address
  - into host byte order 663
- local time corrections 758
- localdef.h header file 33
- localdtconv() library function 754
- locale
  - categories
    - LC\_CTYPE locale variable 396
    - LC\_SYNTAX variable 614
  - character class 149
  - collate.h header file 23
  - collating elements
    - converting 206
    - equivalent 200
    - list of 202
    - rangelist 204
  - default 1244
  - elements
    - converting collating 206
    - equivalent collating 200
    - list of collating 202
    - rangelist of collating 204
  - iconv.h header file 30
  - ims.h header file 30
  - library functions
    - localeconv() 756
  - locale.h header file 33
  - nl\_langinfo.h header file 38
- locale (*continued*)
  - nl\_types.h header file 38
  - null-string ("" ) category 1243
  - retrieving information 866
  - setlocale() library function 1241
  - strxfrm() library function 1465
  - time.h header file 51
  - variant.h header file 54
- localeconv() library function 756
- localtime() library function 758
- locating storage 458
- lock
  - attempt to a mutex object 992
  - wait for on a mutex object 990
- lockf() library function 760
- LOG\_ALERT symbolic constant 1486
- LOG\_CONS symbolic constant 882
- LOG\_CRIT symbolic constant 1486
- LOG\_DEBUG symbolic constant 1486
- LOG\_EMERG symbolic constant 1486
- LOG\_ERR symbolic constant 1486
- LOG\_INFO symbolic constant 1486
- LOG\_LOCAL0 symbolic constant 1487
- LOG\_LOCAL1 symbolic constant 1487
- LOG\_LOCAL2 symbolic constant 1487
- LOG\_LOCAL3 symbolic constant 1487
- LOG\_LOCAL4 symbolic constant 1487
- LOG\_LOCAL5 symbolic constant 1487
- LOG\_LOCAL6 symbolic constant 1487
- LOG\_LOCAL7 symbolic constant 1487
- LOG\_MASK macro 1250
- LOG\_NDELAY symbolic constant 882
- LOG\_NOTICE symbolic constant 1486
- LOG\_NOWAIT symbolic constant 882
- LOG\_ODELAY symbolic constant 882
- LOG\_PID symbolic constant 882
- LOG\_UPTO macro 1250
- LOG\_USER symbolic constant 882, 1487
- LOG\_WARNING symbolic constant 1486
- log() library function 762
- log10() library function 767
- log1p() library function 766
- logb() library function 763
- logic errors 111
- logical record length parameter 419
  - See also* LRECL (logical record length) parameter
- \_\_login() library function 764
- login name
  - \_\_getlogin1() 552
  - getlogin() 550
- long integer, translating 648
- \_longjmp() library function 771
- longjmp() library function 768
- \_\_LOWER clrmemf() argument 197
- \_\_LOWER macro 44

- lrand48() library function 773
- LRECL (logical record length) parameter
  - fopen() library function 419
  - fopen() library function
    - description 419
- LRECL parameter for fopen() library function 419
- lsearch() library function 775
- lseek() library function 776
- lstat() library function 778

## M

### macros

- accessing arguments, variable-length lists 42
- assert() macro 23
- \_\_csplist 24
- defined in assert.h 23
- defined in csp.h 24
- defined in dynit.h 25
- defined in errno.h 25
- defined in langinfo.h 30
- defined in leawi.h 31
- defined in locale.h 33
- defined in math.h 35
- defined in nl\_langinfo.h 38
- defined in regex.h 40
- defined in signal.h 41
- defined in stdarg.h 42
- defined in stddef.h 42
- defined in stdio.h 43
- defined in stdlib.h header 45
- defined in string.h header 47
- defined in sys/modes.h header 47
- defined in time.h header 52
- defined in unistd.h header 53
- defined in wchar.h header 54
- feature test 17
- HUGE\_VAL 36
- LEAWI\_INCLUDED 31
- NULL 42
- offsetof 42
- OMIT\_FC 31
- preprocessor 65
- regular expressions 40
- use with \_\_amrc structure 44
- use with clrmemf() 44
- use with fldata() 44
- use with flocate() 44
- WEOF 55, 56
- makecontext() library function 783
- malloc() library function 786
- matching failure 470
- math functions
  - bessel library functions of the first kind 726
  - bessel library functions of the second kind 1793

- math.h header file 35
- maxcoll() library function 788
- maxdesc() library function 789
- maximum
  - file names 43
  - opened files 43
  - temporary file name 43
- MB\_CUR\_MAX macro 45
- mblen() library function 791
- mbrlen() library function 794
- mbrtowc() library function 797
- mbsinit() library function 800
- mbsrtowcs() library function 802
- mbstowcs() library function 804
- mbtowc() library function 806
- memccpy() library function 808
- memchr() library function 809
- memcmp() library function 811
- memcpy() library function 813
- memmove() library function 814
- memory files
  - across system calls 1490
  - clearing 197
- memory files across system calls 1490
- memory.h header file 36
- memset() library function 816
- messages
  - receive and store in buffers 1106, 1110
  - sending on socket 1185
- miscellaneous functions 111
- \_\_MIXED symbolic constant 691
- mkdir() library function 817
- mkfifo() library function 820
- mknod() library function 823
- mkstemp() library function 826
- mktemp() library function 827
- mktime() library function 828
- \_\_mlockall() library function 831
- mmap() library function 832
- mntent.h header file 47
- mode
  - changing 174, 344
- mode, fopen() 417
- modf() library function 837
- monetary.h header file 36
- MORECTL symbolic constant 556, 1057
- MOREDATA symbolic constant 556, 1057
- mount() library function 838
- mprotect() library function 841
- mrnd48() library function 843
- \_MSE\_PROTOS feature test macro 20
- MSG\_ANY symbolic constant 556, 1057
- MSG\_BAND symbolic constant 556, 1057
- MSG\_HIPRI symbolic constant 556, 1057
- MSG\_NOERROR symbolic constant 850, 854

- msgcat.h header file 36
- msgctl() library function 845
- msgget() library function 847
- msgrcv() library function 850
- msgsnd() library function 852
- msgxrcv() library function 854
- msync() library function 856
- MTF (multitasking facility)
  - functions 37
  - terminating subtasks 1630
- mtf.h header file 37
- multibyte characters; character set identifier 240
- multiple entry points 382
- munmap() library function 858
- mutex
  - attribute object
    - destroy a 996
    - initialize a 1004
    - kind attribute, get a 998
    - kind attribute, set a 1006
  - object
    - delete a 986
    - initialize a 988
    - lock, attempt to 992
    - lock, wait for a 990
    - unlock a 994
- MVS (Multiple Virtual System)
  - compiling fetched modules 372
  - interoperability 423, 1677

## N

- name, binding to a socket 128
- natural logarithm 762
- ndbm.h header file 37
- NDEBUG compiler option 111
- net/if.h header file 37
- net/rtroute.h header file 37
- netdb.h header file 37
- netinet/in.h header file 37
- network
  - services information data set, opening 1267
- network byte order 648, 649, 661
- network entry 560
- network information data set 309
  - opening 1251
- network name 558
- network number
  - getting decimal host address 666
  - getting Internet host address 665
- network protocol information data sets 310
- network service
  - by name 597
  - by port 599
- network services information data sets 312
- new.h header file 37
- nextafter() library function 860
- nftw() library function 861
- nice() library function 864
- nl\_langinfo.h header file 38
- nl\_langinfo() library function 866
- nl\_types.h header file 38
- nlist.h header file 37
- nlist() library function 865
- nonlocal goto
  - longjmp() 768
  - setjmp() 1234
- nonrecursive mutex
  - pthread\_mutexattr\_getkind\_np() 998
  - pthread\_mutexattr\_setkind\_np() 1006
- nrand48() library function 868
- ntohl() library function 870
- ntohs() library function 871
- NULL macro 42, 44
- NULL pointer 42, 43
- NULL pointer constant 45
- null-string locale category 1243

## O

- O\_NONBLOCK symbolic constant 556, 1057
- \_OE\_SOCKETS feature test macro 19
- OFFSET compiler option 252
- offsetof macro 42
- OMIT\_FC macro 31
- \_\_opargf() function 68
- \_OPEN\_DEFAULT feature test macro 19
- \_OPEN\_SOURCE feature test macro 19
- \_\_open\_stat() library function 885
- \_OPEN\_SYS feature test macro 18
- \_OPEN\_SYS\_IPC\_EXTENSIONS feature test macro 19
- \_OPEN\_SYS\_PTY\_EXTENSIONS feature test macro 19
- \_OPEN\_THREADS feature test macro 18
- \_OPEN\_MSGQ\_EXT feature test macro 20
- open() library function 872
- opendir() library function 877
- \_\_opendir2() library function 880
- opening
  - files 417
  - streams 417, 460
- openlog() library function 882
- \_\_openMvsRel() library function 884
- \_OPEN\_SYS SOCK\_EXT feature test macro 20
- \_\_operrf() function 68
- \_\_opindf() function 68
- \_\_opoptf() function 68
- options, socket 1270
- OS/390 UNIX System Services
  - debugging, OS/390 UNIX reason codes 319

OS/390 UNIX System Services (*continued*)

fcntl.h header file 28

output

    baud rate

        determining 159

        termios 163

ownership of files/directories 177, 346

## P

parameters, understanding the socket() 1372

parent process ID 576

\_\_passwd() library function 894

password structure 39

path name

    of controlling terminal 246

pathconf() library function 896

pause() library function 899

pclose() library function 901

peer

    socket address, presetting 1253

perror() library function 903

\_\_pid\_affinity() library function 905

pipe() library function 907

PKTXTND symbolic constant 1542

poll.h header file 38

poll() library function 910

popen() library function 914

position options for flock() library function 405

positional parameters 417

\_POSIX\_C\_SOURCE feature test macro 18

\_POSIX\_SOURCE feature test macro 18

\_POSIX1\_SOURCE feature test macro 18

pow() library function 916

pragmas

    linkage with fetch() library function 370

precision argument, fprintf() family 439

printf() library function 436

PRIO\_PGRP symbolic constant 578, 1257

PRIO\_PROCESS symbolic constant 578, 1257

PRIO\_USER symbolic constant 578, 1257

process

    control from within programs 1488

    creating 422

    data 1759

    group 1268

    group ID 571, 1268

    group ID, foreground 1520, 1546

    group leader 1268

    ID 573, 576, 1268

    signal 728

protocol

    getting name by name 580

    getting name by number 582

    getting next entry 584

    information data set, opening 1259

ps.h header file 48

pseudorandom integers 1077

pthread\_attr\_destroy() library function 918

pthread\_attr\_getdetachstate() library function 920

pthread\_attr\_getstacksize() library function 922

pthread\_attr\_getsynctype\_np() library function 924

pthread\_attr\_getweight\_np() library function 925

pthread\_attr\_init() library function 927

pthread\_attr\_setdetachstate() library function 929

pthread\_attr\_setstacksize() library function 931

pthread\_attr\_setsynctype\_np() library function 933

pthread\_attr\_setweight\_np() library function 934

pthread\_cancel() library function 936

pthread\_cleanup\_pop() library function 939

pthread\_cleanup\_push() library function 941

pthread\_cond\_broadcast() library function 943

pthread\_cond\_destroy() library function 945

pthread\_cond\_init() library function 947

pthread\_cond\_signal() library function 948

pthread\_cond\_timedwait() library function 950

pthread\_cond\_wait() library function 952

pthread\_condattr\_destroy() library function 955

pthread\_condattr\_getkind\_np() library function 957

pthread\_condattr\_init() library function 959

pthread\_condattr\_setkind\_np() library function 961

pthread\_create() library function 963

pthread\_detach() library function 965

pthread\_equal() library function 967

pthread\_exit() library function 969

pthread\_getspecific() library function 971

pthread\_getspecific\_d8\_np() library function 974

PTHREAD\_INTR\_ASYNCHRONOUS cancelability  
type 936

PTHREAD\_INTR\_CONTROLLED cancelability  
type 936

PTHREAD\_INTR\_DISABLE cancelability type 936

PTHREAD\_INTR\_ENABLE cancelability type 936

pthread\_join() library function 977

pthread\_join\_d4\_np() library function 979

pthread\_key\_create() library function 981

pthread\_kill() library function 984

pthread\_mutex\_destroy() library function 986

pthread\_mutex\_init() library function 988

pthread\_mutex\_lock() library function 990

pthread\_mutex\_trylock() library function 992

pthread\_mutex\_unlock() library function 994

pthread\_mutexattr\_destroy() library function 996

pthread\_mutexattr\_getkind\_np() library function 998

pthread\_mutexattr\_init() library function 1004

pthread\_mutexattr\_setkind\_np() library function 1006

pthread\_once() library function 1013

pthread\_security\_np() library function 1029

pthread\_self() library function 1033

pthread\_setinr() library function 1035

pthread\_setinrtype() library function 1038

- pthread\_set\_limit\_np() library function 1041
- pthread\_setspecific() library function 1043
- pthread\_tag\_np() library function 1046
- pthread\_testintr() library function 1047
- pthread\_yield() library function 1049
- pthread.h header file 38
- ptrdiff\_t type in stddef header file 42
- ptsname() library function 1051
- pushing characters back onto input stream 1655
- putc() library function 1052
- putchar() library function 1052
- putenv() library function 1054
- putmsg() library function 1056
- putpmsg() library function 1056
- puts() library function 1059
- pututxline() library function 1061
- putw() library function 1063
- putwc() library function 1064
- putwchar() library function 1066
- pwd.h header file 39

## Q

- qsort() library function 1068
- QueryMetrics() library function 1070
- QuerySchEnv() library function 1072
- quick sort 1068

## R

- raise() library function 1074
- RAND\_MAX macro 45
- rand() library function 1077
- random
  - access 474, 485
  - number generator 1077, 1398
- random() library function 1079
- re\_comp.h header file 39
- re\_comp() library function 1100
- re\_exec() library function 1115
- read operations with fgetc() 388
- \_\_READ\_RESOURCE symbolic constant 170
- read() library function 1080
- readdir() library function 1086
- \_\_readdir2() library function 1089
- reading
  - buffers, data stored in 1093
  - character from stdin 388, 499, 627, 629
  - character from stream 499, 627, 629
  - data items from stream 456
  - data, and store in buffers 1093
  - directory, readdir() library function 1086, 1089
  - formatted 464
  - from file, read() library function 1080
  - line from stdin 595
  - line from stream 392, 394, 396

- reading (*continued*)
  - lock while reading a file 353
  - read a string, fgets() library function 392
  - scanning 464
  - value of symbolic link, readlink() library function 1091
- readlink() library function 1091
- readv() library function 1093
- real
  - group ID 521, 1220
  - user ID 618, 1278
- realloc() library function 1096
- reallocation of block size 1096
- realpath() library function 1099
- reason codes
  - debugging with \_\_errno2() 319
- receiving
  - data and store in buffers 1103
  - messages and store in buffers 1106, 1110
- rcfm parameter 419
- record
  - format parameter 419
- recursive mutex 998, 1006
- recv() library function 1103
- recvfrom() library function 1106
- recvmsg() library function 1110
- redirection
  - streams, using freopen() 460
- REG\_EXTENDED macro 40
- REG\_ICASE macro 40
- REG\_NEWLINE macro 40
- REG\_NOSUB macro 40
- REG\_NOTEOL macro 40
- regcmp() library function 1116
- regcomp() library function 1120
- regerror() library function 1123
- regex.h header file 40
- regex() library function 1125
- regexec() library function 1126
- regex.h header file 40
- regfree() library function 1129
- regular expressions 40, 209, 1100, 1120
- release
  - \_\_librel() library function 746
  - level macro 746
- release processor to other threads 1049
- release() library function 1130
- remainder library function 1132
- remainder of division 277
- remove cleanup thread handler 939
- remove() library function 1133
- remque() library function 1135
- rename() library function 1136
- renaming files 1136
- reopening streams 460



- reserved names 61
- resource limits defined 32
- response match 1150
- rewind a stream 1139
- rewind() library function 1139
- rewinddir() library function 1141
- rexec.h header file 40
- rexec() library function 1143
- rindex() library function 1145
- rint() library function 1146
- rmdir() library function 1147
- rpmatch() library function 1150
- RRDS (Relative Record Data Set) 493
- RS\_HIPRI
  - RS\_HIPRI symbolic constant 1056
- RS\_HIPRI symbolic constant 1057
- rtroute.h header file 37

## S

- S\_IRGRP symbolic constant 848, 1172, 1291
- S\_IROTH symbolic constant 848, 1172, 1291
- S\_IRUSR symbolic constant 847, 1172, 1291
- S\_ISBLK(mode) macro 779, 1405
- S\_ISCHR(mode) macro 779, 1405
- S\_ISDIR(mode) macro 779, 1405
- S\_ISEXTL(mode) macro 779, 1405
- S\_ISFIFO(mode) macro 779, 1405
- S\_ISLNK(mode) macro 779, 1405
- S\_ISREG(mode) macro 779, 1405
- S\_ISSOCK(mode) macro 1405
- S\_IWGRP symbolic constant 848, 1172, 1291
- S\_IWOTH symbolic constant 848, 1173, 1291
- S\_IWUSR symbolic constant 847, 1172, 1291
- \_\_S99parms structure in svc99() 1467
- safety, signal 1295
- saved set-user-ID 1278
- sbrk() library function 1152
- scalb() library function 1154
- scanf() library function 464
- search.h header file 40
- searching
  - buffers 809
  - qsort 1068
  - strings 1416, 1444
  - strings for tokens 1458
- seed for random numbers 1398
- seed48() library function 1156
- SEEK\_CUR macro 44
  - effects of ungetc() and ungetwc() 474
- SEEK\_END macro 44
- SEEK\_SET macro 44
- seekdir() library function 1158
- select() library function 1159
- selectex() library function 1166
- SEM\_UNDO symbolic constant 1176
- semctl() library function 1169
- semget() library function 1172
- semop() library function 1175
- send() library function 1178
- send a signal to a thread 984
- sendmsg() library function 1185
- sendto() library function 1190
- serial number
  - lstat() 778
  - stat() 1404
- server
  - in the AF\_INET domain 215
  - in the AF\_UNIX domain 215
  - incoming client requests 752
- \_\_server\_classify() library function 1194
- \_\_server\_classify\_create() library function 1197
- \_\_server\_classify\_destroy() library function 1198
- \_\_server\_classify\_reset() library function 1199
- \_\_server\_init() library function 1200
- \_\_server\_pwu() library function 1203
- session leader 1268
- set a condition variable attribute object 961
- set\_new\_handler() library function 1252
- set\_terminate() library function 1276
- set\_unexpected() library function 1281
- SETALL symbolic constant 1170
- setbuf() library function 1208
- setcontext() library function 1210
- setegid() library function 1213
- setenv() library function 1215
- seteuid() library function 1218
- setgid() library function 1220
- setgrent() library function 307
- setgroups() library function 1223
- sethostent() library function 1224
- setibmopt() library function 1225
- setibmsockopt() library function 1227
- setitimer() library function 1232
- \_setjmp() library function 1237
- setjmp.h header file 40
- setjmp() library function 1234
- setkey() library function 1239
- setlocale() library function 1241
- setlogmask() library function 1250
- setnetent() library function 1251
- setpeer() library function. 1253
- setpgid() library function 1254
- setpgrp() library function 1256
- setpriority() library function 1257
- setprotoent() library function 1259
- setpwent() library function 1260
- setregid() library function 1261
- setreuid() library function 1262
- setrlimit() library function 1264

- setservent() library function 1267
- setsid() library function 1268
- setsockopt() library function 1270
- setstate() library function 1275
- setuid() library function 1278
- setutxent() library function 1282
- SETVAL symbolic constant 1169
- setvbuf() library function 1283
- shared
  - memory files across system calls 1490
- SHM\_RDONLY symbolic constant 1285
- SHM\_RND symbolic constant 1285
- shmat() library function 1285
- shmctl() library function 1287
- shmdt() library function 1289
- shmget() library function 1290
- shutdown
  - duplex connection 1293
- shutdown() library function 1293
- sig argument in signal() library function 1330
- SIG\_DFL macro 41
- SIG\_DFL signal action 1343
- SIG\_ERR macro 41
- SIG\_HOLD 1344
- SIG\_IGN macro 41
- SIG\_IGN signal action 1344
- SIG\_PROMOTE macro 41
- SIGABND macro 41
- SIGABRT macro 41
- sigaction() library function 1295
- \_\_sigactionset() library function 1305
- sigaddset() library function 1312
- SIGALRM signal. 102
- sigaltstack() library function 1314
- sigdelset() library function 1316
- sigemptyset() library function 1318
- sigfillset() library function 1320
- SIGFPE macro 41
- sighold() library function 1322
- sigignore() library function 1323
- SIGILL macro 41
- SIGINT macro 41
- siginterrupt() library function 1324
- SIGIOERR macro 41
- sigismember() library function 1325
- siglongjmp() library function 1327
- signal
  - change mask and suspend thread 1346, 1351
  - handler 1330
  - mask 899
  - pending 1337
  - safety 1295
  - send to a thread 984
  - sets 1312
  - unblock a thread 948
- signal.h header file 41
- signal() library function 1330
- \_\_siggam() library function 1335
- \_\_siggam() library function 1335
- sigpause() library function 1336
- sigpending() library function 1337
- sigprocmask() library function 1339
- sigrelse() library function 1342
- SIGSEGV macro 41
- sigset() library function 1343
- sigsetjmp() library function 1346
- sigstack() library function 1349
- sigsuspend() library function 1351
- SIGTERM macro 41
- SIGTTOU signal in tcdrain() library function 1507
- SIGUSR1 macro 41
- SIGUSR2 macro 41
- sigwait() library function 1356
- sin() library function 1360
- sine
  - calculating 1360
  - hyperbolic, calculating 1362
- sinh() library function 1362
- size\_t structure 42
- sleep() library function 1364
- \_\_smf\_record() library function 1366
- sock\_debug() library function 1367
- sock\_debug\_bulk\_perf0() library function 1368
- sock\_do\_bulkmode() library function 1369
- sock\_do\_teststor library function 1370
- socket
  - address, peer 1253
  - creating 1371
  - creating a pair 1375
  - data, sending on 1190
  - data, writing 1785
  - datagrams, sending on 1178
  - descriptor AF\_UNIX domain 129
  - descriptor in AF\_INET domain 128
  - getting name 604
  - messages, sending on 1185
  - operating characteristics, specifying 672
  - options, getting 606
  - options, setting 1270
  - pairs, creating 1375
  - peer address, presetting 1253
  - peer connected to 568
  - send data on 1178, 1190
  - send messages on 1185
  - shutdown 1293
  - writing data on 1785
- socket() library function 1371
- socket() parameters, understanding the 1372
- socketpair() library function 1375
- sorting 1068

- space= parameter 419
- spawn.h header file 42
- spawn() library function 1377
- spawn2() library function 1388
- spawnp() library function 1377
- spawnp2() library function 1388
- spc.h header file 42
- special file, create a 823
- specific value for a key
  - get 971, 974
  - set 1043
- sprintf() library function 436
- square root function 1396
- srand() library function 1398
- srand48() library function 1401
- random() library function 1400
- SS\_DISABLE symbolic constant 1314
- SS\_ONSTACK symbolic constant 1314
- sscanf() library function 464
- ST\_NOSUID 1409
- ST\_OEEXPOSED 1409
- ST\_RDONLY 1409
- stack
  - restoring the environment 768
  - saving an environment 1234
- stacksize attribute
  - get 922
  - set 931
- standard
  - stream
    - redirecting 460
- standard streams 43
- standards, indicated by table 62
- START
  - character 1533
- stat structure 1404
- stat.h header file 48
- stat() library function 1404
- statfs.h header file 49
- status analysis macros 1687, 1693
- statvfs() library function 1408
- stdarg.h header file 42
- stddef.h header file 42
- stdio.h header file 43
- stdlib.h header file 45
- stdout
  - format and print data 1681
- stdout, format and print data 1681
- step() library function 1411
- STEPLIB environment variable 322
- STIMER\_REAL TQE 189
- stop bits 1535
- STOP character 1533
- stopping a process or program 71
- storage
  - allocation 42
- storage (*continued*)
  - allocation, library function, calloc() 141
  - free 458
  - locating 458
  - reserving with malloc() 786
- strbuf 555
- strcasecmp() library function 1413
- strcat() library function 1414
- strchr() library function 1416
- strcmp() library function 1418
- strcoll() library function 1420
- strcpy() library function 1422
- strcspn() library function 1424
- strdup() library function 1426
- streams
  - access mode 460
  - associating with file descriptor 363
  - binary mode 460
  - buffering 1208
  - changing current file position 474, 485
  - changing file position 1139
  - closing 348
  - EOF (end of file) 365
  - format and print data 1679
  - formatted I/O 436, 464
  - Input/Output 348
  - opening 417
  - reading characters with fgetc() 388
  - reading characters with getc() and getchar() 499
  - reading characters with getwc() 627
  - reading characters with getwchar() 629
  - reading data items with fread() 456
  - reading lines with fgets() 392
  - reading lines with fgetwc() 394
  - reading lines with fgetws() 396
  - reading lines with gets() 595
  - redirection 460
  - reopening 460
  - rewinding 1139
  - text mode 460
  - translation mode 460
  - ungetting characters 1655
  - ungetting wide characters 1658
  - updating 417, 460
  - writing characters with fputc() 448
  - writing characters with fputwc() 452
  - writing characters with putc() and putchar() 1052
  - writing data items 495
  - writing lines with puts() 1059
  - writing strings 450, 454
- STREAMS data areas
  - strbuf 555
- STREAMS interfaces
  - fattach() 339
  - fdetach() 361
  - getmsg() - getpmsg() 555

## STREAMS interfaces (*continued*)

- isastream() 702
- putmsg() - putmsg() 1056
- strerror() library function 1427
- strfmon() library function 1428
- strftime() library function 1432
- string.h header file 46
- strings
  - classification with ctype.h 24
  - classifying 694
  - comparing 1424, 1440
    - language collation 1705
  - concatenating 1414, 1438
  - conversion to double 1456
  - conversion to unsigned integer 1462
  - converting case 1592, 1604
  - converting to integer 122
  - copying 1422, 1442
  - finding in a string 1444
  - ignoring case 1418, 1424
  - initializing 1442
  - length of 1436
  - locating 1453
  - multibyte
    - conversion using mbrtowc() 797
    - conversion using mbstowcs() 804
    - conversion using mbtowc() 806
    - conversion with mbsrtowcs() 802
  - length 791, 794
  - property 720
  - property classification 1753
  - reading with fgetc() 388
  - reading with getc() and getchar() 499
  - reading with getwc() 627
  - reading with getwchar() 629
  - searching 1416, 1444
  - searching for tokens 1458
  - searching, strspn() library function 1451
  - setting 816
  - testing 694
  - ungetting 1655, 1658
  - writing with fputc() 448
  - writing with fputs() 450
  - writing with fputwc() 452
  - writing with fputws() 454
  - writing with putc() and putchar() 1052
- strings.h header file 46
- strlen() library function 1436
- strncasecmp() library function 1437
- strncat() library function 1438
- strncmp() library function 1440
- strncpy() library function 1442
- stropts.h 687
- stropts.h header file 46
- strpbrk() library function 1444

- strptime() library function 1445
- strrchr() library function 1449
- strspn() library function 1451
- strstr() library function 1453
- strtcoll() library function 1454
- strtod() library function 1456
- strtok() library function 1458
- strtol() library function 1460
- strtoul() library function 1462
- strxfrm() library function 1465
- supplementary group ID 1220
- svc99() library function 292, 1467
- swab() library function 1471
- swapcontext() library function 1472
- switching, AMODE 371
- swprintf() library function 1475
- swscanf() library function 1477
- symbolic constants in errno.h 25
- symlink() library function 1479
- sync() library function 1482
- syntax diagrams, how to read xxxiii
- syntax of format for fprintf() family 437
- sys/\_\_getip.h header file 47
- sys/\_\_messag.h header file 48
- sys/\_\_ussos.h header file 50
- sys/file.h header file 47
- sys/ioctl.h header file 47
- sys/ipc.h header file 47
- sys/mman.h header file 47
- sys/mntent.h header file 47
- sys/modes.h header file 47
- sys/msg.h header file 48
- sys/ps.h header file 48
- sys/resource.h header file 48
- sys/sem.h header file 48
- sys/server.h header file 48
- sys/shm.h header file 48
- sys/socket.h header file 48
- sys/stat.h header file 48
- sys/statfs.h header file 49
- sys/statvfs.h header file 49
- sys/time.h header file 49
- sys/timeb.h header file 49
- sys/times.h header file 49
- sys/ttydev.h header file 49
- sys/types.h header file 49
- sys/uio.h header file 50
- sys/un.h header file 50
- sys/utsname.h header file 50
- sys/wait.h header file 51
- sys/wlm.h header file 51
- sysconf() library function 1483
- syslog.h header file 47
- syslog() library function 1486
- system configuration options 1483

- system() library function 1488
  - calls across mixed environments 1489
  - calls, general discussion 1488
  - operating
    - displaying name 1652
    - programming environment 42

## T

- t\_accept() library function 1493
- t\_alloc() library function 1498
- t\_bind() library function 1504
- t\_close() library function 1523
- t\_connect() library function 1524
- t\_error() library function 1561
- t\_free() library function 1564
- t\_getinfo() library function 1566
- t\_getprotaddr library function 1568
- t\_getstate() library function 1569
- t\_listen() library function 1577
- t\_look() library function 1579
- t\_open() library function 1594
- t\_optmgmt() library function 1596
- t\_rcv() library function 1605
- t\_rcvconnect() library function 1607
- t\_rcvdis() library function 1609
- t\_rcvrel() library function 1611
- t\_rcvudata library function 1612
- t\_rcvuderr library function 1613
- t\_snd() library function 1619
- t\_snddis() library function 1621
- t\_sndrel() library function 1623
- t\_sndudata library function 1624
- t\_strerror() library function 1625
- t\_sync() library function 1626
- t\_unbind() library function 1635
- takesocket() library function 1496
- tan() library function 1500
- tangent
  - calculating 1500
  - hyperbolic, calculating 1502
- tanh() library function 1502
- tar.h header file 51
- t\_bind() library function 1504
- \_TCCP\_BINARY symbolic constant 1518, 1543, 1549
- \_TCCP\_CPNAMEMAX symbolic constant 1543, 1550
- \_TCCP\_FASTP symbolic constant 1517, 1543, 1550
- \_\_tccp\_flags 1543, 1549
- \_\_tccp\_fromname 1543, 1550
- \_\_tccp\_toname 1543, 1550
- tcdrain() library function 1507
- tcflow() library function 1509
- tcflush() library function 1512
- tcgetattr() library function 1515
- \_\_tcgetcp() library function 1517
- tcgetpgrp() library function 1520
- tcgetsid() library function 1522
- t\_close() library function 1523
- t\_connect() library function 1524
- tcpperror() library function 1527
- tcsendbreak() library function 1529
- tcsetattr() library function 1531
- \_\_tcsetcp() library function 1542
- tcsetpgrp() library function 1546
- \_\_tcsettables() library function 1549
- tdelete() library function 1555
- telldir() library function 1557
- tempnam() library function 1558
- temporary files 1582
  - names 43, 1584
  - number of 43
- \_\_termcp structure 1543
- terminals
  - attributes 1515
  - break condition 1529
  - control modes 1535
  - I/O
    - flush 1512
  - input modes 1531
  - local modes 1536
  - output modes 1533
  - suspend/resume data flow 1509
- terminat.h header file 51
- terminate() library function 1560
- termination
  - a program 330
  - abort() library function 71
  - atexit() library function 116
  - exit() library function 330
  - process or program 71
- termios structure 1531
- termios.h header file 51
- \_\_t\_errno() function 68
- t\_error() library function 1561
- testing characters 694
- text
  - files 418
- tfind() library function 1563
- t\_free() library function 1564
- t\_getinfo() library function 1566
- t\_getprotaddr library function 1568
- t\_getstate() library function 1569
- threads
  - asynchronous signal, wait for an 1356
  - attribute object
    - destroy the definition 918
    - detachstate, get the current value 920
    - detachstate, set the current value 929
    - initialize a 927
    - stacksize, get the 922
    - stacksize, set the 931
    - weight, get the current 925

## threads (*continued*)

- attribute object (*continued*)
  - weight, set the current 934
- broadcast a condition 943
- caller's ID, get the 1033
- cancel 936
- cancelability point, establish a 1047
- cancelability states, set the calling thread's 1035
- cancelability types, set the calling thread's 1038
- changing signal mask 1339
- compare thread IDs 967
- condition variable attribute object, destroy 955
- condition variable attribute object, get 957
- condition variable attribute object, initialize 959
- condition variable attribute object, set 961
- create 963
- create key identifier 981
- delete a mutex object 986
- destroy a condition variable 945
- destroy a mutex attribute object 996
- destroy the thread attributes object 918
- detach 965
- detachstate, get the 920
- detachstate, set the 929
- establish cleanup handler 941
- exit 969
- initialize a condition variable 947
- initialize a mutex 988
- initialize a mutex attribute object 1004
- initialize a thread attributes object 927
- invoke a function once 1013
- kind attribute, get from mutex attribute object 998
- kind attribute, set from a mutex attribute object 1006
- lock, attempt to a mutex object 992
- lock, wait for on a mutex object 990
- pthread.h header file 38
- release processor to other threads 1049
- remove cleanup handler 939
- send a signal 984
- signal a condition 948
- specific value for a key, get the 971, 974
- specific value for a key, set the 1043
- stacksize, get the 922
- stacksize, set the 931
- unlock a mutex object 994
- wait for thread to end 977, 979
- wait on a condition variable 952
- wait on a condition variable for a limited time 950
- weight, get the current 925
- weight, set the current 934

## time

- considerations 189
- conversion 1445
- converting from long integer to string 250
- converting time structure to string 106

## time (*continued*)

- correcting for local time 758
- file access/modification 1664
- format 1711
- formatted 1432
- zone, testing 1637
- time\_t type 273
- time.h header file 51
- time() library function 1570
- times.h header file 49
- times() library function 1572
- tinit() library function 1575
- TIOCPKT\_CHCP symbolic constant 1542
- t\_listen() library function 1577
- t\_look() library function 1579
- TLOOK error 1579
- tm structure 640
- TMP\_MAX macro 44, 1584
- tmpfile() library function 1582
- tmpnam() library function 1584
  - file name specs in stdio.h file 43
- toascii() library function 1586
- tokens 1458
- \_tolower() library function 1593
- tolower() library function 1592
- t\_open() library function 1594
- t\_optmgmt() library function 1596
- \_toupper() library function 1603
- toupper() library function 1592
- towlower() library function 1604
- towupper() library function 1604
- traceback 252
- transforming strings 1465
- t\_rcv() library function 1605
- t\_rcvconnect() library function 1607
- t\_rcvdis() library function 1609
- t\_rcvrel() library function 1611
- t\_rcvudata library function 1612
- t\_rcvuderr library function 1613
- trigonometric functions
  - hyperbolic sine 1362
  - hyperbolic tangent 1502
  - sine 1360
  - tangent 1500
- truncate() library function 1614
- tsched() library function 1615
- tsearch() library function 1617
- t\_snd() library function 1619
- t\_snddis() library function 1621
- t\_sndrel() library function 1623
- t\_sndudata library function 1624
- t\_strerror() library function 1625
- t\_sync() library function 1626
- tsyncro() library function 1628
- tterm() library function 1630

ttyname() library function 1632  
 ttyslot() library function 1634  
 t\_unbind() library function 1635  
 twalk() library function 1636  
 \_\_24malc() library function 1795  
 typedef  
     definitions in stddef.h 42, 47  
 types.h header file 49  
 TZ environment variable 1637  
 \_\_tzone() function 69  
 tzset() library function 1637

## U

ualarm() library function 1640  
 ucontext.h header file 52  
 uheap.h header file 52  
 UL\_GETFSIZE symbolic constant 1645  
 UL\_SETFSIZE symbolic constant 1645  
 ulimit.h header file 53  
 ulimit() library function 1645  
 umask default 1647  
 umask() library function 1647  
 umount() library function 1649  
 uname() library function 1652  
 unblock a thread 943, 948  
 underscore character mapping to \_\_ 61  
 unexpect.h header file 52  
 unexpected() library function 1654  
 ungetc() library function 1655  
 ungetwc() library function 1658  
 unistd.h header file 53  
 unlink() library function 1660  
 unlock a mutex object 994  
 unlockpt() library function 1662  
 unsigned short integer 649  
 \_\_UPDATE\_RESOURCE symbolic constant 170  
 \_\_UPPER symbolic constant 691  
 user database 587, 589  
 user ID  
     effective 519  
     real 618  
     setting 1218  
 usleep() library function 1663  
 utime.h header file 53  
 utime() library function 1664  
 utimes() library function 1667  
 utmpx.h header file 54  
 \_\_utmpxname() library function 1669  
 utsname.h header file 50

## V

va\_arg() macro 1670  
 va\_end() macro 1670

valloc() library function 1675  
 \_VARARG\_EXT\_ feature test macro 20, 1671  
 varargs.h header file 54  
 variables  
     configurable path name 896  
     configuration 427  
 variant structure 614  
 variant.h header file 54  
 va\_start() macro 1670  
 vfork() library function 1676  
 vfprintf() library function 1679  
 vprintf() library function 1681  
 VSAM (Virtual Storage Access Method)  
     I/O operations  
         deleting a record 359  
         updating a record 493  
 vsprintf() library function 1683  
 vswprintf() library function 1685

## W

wait  
     asynchronous signal 1356  
     child process 1687, 1692  
     condition variable 952  
     condition variable for a limited time 950  
     thread to end 977, 979  
 wait.h header file 51  
 wait() library function 1687  
 wait3() library function 1696  
 waitid() library function 1690  
 waitpid() library function 1692  
 wchar.h header file 54  
 wctomb() library function 1697  
 wcsat() library function 1699  
 wcschr() library function 1701  
 wcscmp() library function 1703  
 wscoll() library function 1705  
 wcsncpy() library function 1707  
 wcscspn() library function 1709  
 wcsftime() library function 1711  
 wcsid() library function 1713  
 wcslen() library function 1715  
 wcsncat() library function 1716  
 wcsncmp() library function 1718  
 wcsncpy() library function 1720  
 wcsrchr() library function 1722  
 wcsrchr() library function 1724  
 wcsrtombs() library function 1726  
 wcspn() library function 1729  
 wcsstr() library function 1731  
 wcstod() library function 1733  
 wcstok() library function 1735  
 wcstol() library function 1738  
 wcstombs() library function 1740

- wcstoul() library function 1742
- wcstr.h header file 55
- wcswcs() library function 1744
- wcswidth() library function 1746
- wcsxfrm() library function 1747
- wctob() library function 1749
- wctomb() library function 1751
- wctype.h header file 56
- wctype() library function 1753
- wcwidth() library function 1754
- weight
  - get the current thread 925
  - set the current thread 934
- WEOF macro 55, 56
- w\_getmntent() library function 1756
- w\_getpsent() library function 1759
- wide characters
  - appending to strings 1716
  - break string into tokens 1735
  - character conversion 452
  - character set ID 1713
  - compare strings 1718
  - conversion of string to double floating point 1733
  - conversion of string to multibyte 1740
  - conversion of string to unsigned long integer 1742
  - conversion to byte 1749
  - conversion to multibyte 1697, 1751, 1764, 1766, 1768, 1770, 1772
  - convert string to long integer 1738
  - converting to multibyte string 1726
  - copying strings with wcsncpy() 1707
  - copying strings with wcsncpy() 1720
  - display width 1746, 1754
  - I/O functions 627, 629
  - locating in a string 1722, 1724
  - locating sequence 1731
  - locating substring 1744
  - match offset 1709
  - reading streams and files 396
  - searching for 1729
  - string comparison 1703
  - string conversion 454
  - string length 1715
  - string, reading 1477
  - substring 1701
  - transform string 1747
  - write to wide-character array 1475
  - writing 1685
- wide integer 715
- w\_iocctl() library function 1762
- wmemchr library function 1764
- wmemcmp library function 1766
- wmemcpy library function 1768
- wmemmove library function 1770
- wmemset library function 1772
- wordexp.h header file 56
- wordexp() library function 1774
- wordfree() library function 1778
- working directory
  - changing 167
  - path name 508
- wrapping of output 1059
- writable static
  - shared 382
- write
  - lock 353
  - operations
    - character to stdout 448, 452, 1052
    - character to stream 448, 452, 1052, 1655, 1658
    - data items from stream 495
    - formatted 436
    - line to stream 1059, 1064, 1066
    - printing 495
    - string to stream 450, 454
- write() library function 1780
- writew() library function . 1785
- writing
  - data on sockets 1785
- \_\_wsinit() library function 1788
- w\_statfs() library function 1789
- w\_statvfs() library function 1791

## X

- xhotc() 1795
- xhotl() 1795
- xhott() 1795
- \_XOPEN\_SOURCE feature test macro 19
- xregs() 1795
- xsacc() 1795
- xsrvc() 1795
- xti.h header file 56
- xusr() 1795
- xusr2() 1795

## Y

- y0() library function 1793
- y1() library function 1793
- yield (release processor to other threads) 1049
- yn() library function 1793





---

# Communicating Your Comments to IBM

OS/390  
C/C++ Run-Time  
Library Reference, Volume 1  
Publication No. SC28-1663-05

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM. Whichever method you choose, make sure you send your name, address, and telephone number if you would like a reply.

Feel free to comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. However, the comments you send should pertain to only the information in this manual and the way in which the information is presented. To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

If you are mailing a reader's comment form (RCF) from a country other than the United States, you can give the RCF to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
  - FAX: (International Access Code)+1+914+432-9405
- If you prefer to send comments electronically, use one of these network IDs:
  - IBM Mail Exchange: USIB6TC9 at IBMMAIL
  - Internet e-mail: mhvrcfs@us.ibm.com
  - World Wide Web: <http://www.ibm.com/s390/os390/>

Make sure to include the following in your note:

- Title and publication number of this book
- Page number or topic to which your comment applies

Optionally, if you include your telephone number, we will be able to respond to your comments by phone.

---

## Reader's Comments — We'd Like to Hear from You

OS/390

C/C++ Run-Time

Library Reference, Volume 1

Publication No. SC28-1663-05

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

**Note:** Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Today's date: \_\_\_\_\_

What is your occupation?

Newsletter number of latest Technical Newsletter (if any) concerning this publication:

How did you use this publication?

- |                          |                               |                          |                        |
|--------------------------|-------------------------------|--------------------------|------------------------|
| <input type="checkbox"/> | As an introduction            | <input type="checkbox"/> | As a text (student)    |
| <input type="checkbox"/> | As a reference manual         | <input type="checkbox"/> | As a text (instructor) |
| <input type="checkbox"/> | For another purpose (explain) |                          |                        |

---

Is there anything you especially like or dislike about the organization, presentation, or writing in this manual? Helpful comments include general usefulness of the book; possible additions, deletions, and clarifications; specific errors and omissions.

Page Number:

Comment:

\_\_\_\_\_  
Name

\_\_\_\_\_  
Address

\_\_\_\_\_  
Company or Organization

\_\_\_\_\_  
Phone No.



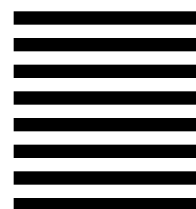
Fold and Tape

Please do not staple

Fold and Tape



NO POSTAGE  
NECESSARY  
IF MAILED IN THE  
UNITED STATES



## BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation  
Department 55JA, Mail Station P384  
522 South Road  
Poughkeepsie NY 12601-5400



Fold and Tape

Please do not staple

Fold and Tape





Program Number: 5647-A01



Printed in the United States of America  
on recycled paper containing 10%  
recovered post-consumer fiber.

SC28-1663-05

