XL C/C++ for z/VM
1.3

Runtime Library Reference

SC09-7624-74

Note:

Before you use this information and the product it supports, read the information in "Notices" on page 1001.

This edition applies to version 1, release 3 of IBM® XL C/C++ for z/VM® (product number 5654-A22) and to all subsequent releases and modifications until otherwise indicated in new editions.

Last updated: 2024-09-18

© Copyright International Business Machines Corporation 2002, 2024.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures ..... **xix**

Tables ..... **xxi**

About This Document ..... **xxiii**
Intended Audience xxiii
Where to Find More Information xxiii
Links to Other Documents and Websites xxiii

How to provide feedback to IBM ..... **xxv**

Summary of Changes for XL C/C++ for z/VM: Runtime Library Reference ..... **xxvii**
SC09-7624-74, XL C/C++ for z/VM, 1.3 (September 2024) xxvii
SC09-7624-73, XL C/C++ for z/VM, 1.3 (December 2023) xxvii
SC09-7624-73, XL C/C++ for z/VM, 1.3 (September 2023) xxvii
SC09-7624-73, XL C/C++ for z/VM, 1.3 (May 2023) xxvii

Chapter 1. About IBM XL C/C++ for z/VM **1**
XL C/C++ Compiler 1
The C Language 1
The C++ Language 1
Common Features of the C Compiler and C++ Compiler Functions 1
Specific Features of the z/VM C Compiler Function 2
Specific Features of the z/VM C++ Compiler Function 3
IBM XL C/C++ for z/VM Runtime Library 3
Class Libraries 3
Utilities 3
Language Environment 4
About Prelinking, Linking, and Binding 5
Notes on the Prelinking Process 5
The Program Management Binder 5
OpenExtensions Services 6
IBM XL C/C++ for z/VM Applications with OpenExtensions Services 7
IBM XL C/C++ for z/VM Applications with OpenExtensions Interoperability 7
Input and Output 8
I/O Interfaces 8
File Types 9
Additional I/O Features 9
Additional Features of IBM XL C/C++ for z/VM 10

Chapter 2. Header Files **13**
Feature Test Macros 13
Header File Descriptions 18
aio.h 18
arpa/inet.h 18
arpa/nameser.h 18
assert.h 18
bpxhmnt.h 19
cassert 19

_Ccsid.h..... 19
cctype..... 19
ceeedcct.h..... 19
cerrno..... 19
cfloat..... 19
cics.h..... 19
ciso646..... 20
climits..... 20
clocale..... 20
cmath..... 20
collate.h..... 20
cpio.h..... 20
csetjmp..... 21
csignal..... 21
csp.h..... 21
cstdarg..... 21
cstddef..... 21
cstdio..... 21
cstdlib..... 22
cstring..... 22
ctest.h..... 22
ctime..... 22
ctype.h..... 22
cwchar..... 23
cwctype..... 23
decimal.h..... 23
dirent.h..... 23
dll.h..... 23
dynit.h..... 23
env.h..... 24
errno.h..... 24
exception..... 24
fcntl.h..... 25
features.h..... 25
float.h..... 25
fmtmsg.h..... 26
fnmatch.h..... 26
fpxcp.h..... 26
__ftp.h..... 26
ftw.h..... 26
glob.h..... 26
grp.h..... 26
iconv.h..... 27
_Ieee754.h..... 27
ims.h..... 27
inttypes.h..... 27
iso646.h..... 27
langinfo.h..... 28
lc_core.h..... 28
lc_sys.h..... 28
leawi.h..... 28
libgen.h..... 28
limits.h..... 28
localdef.h..... 30
locale.h..... 30
math.h..... 32
memory.h..... 33
monetary.h..... 33

msgcat.h.... 33
mtf.h.... 34
_Nascii.h.... 34
ndbm.h.... 34
net/if.h.... 34
net/rtrouteh.h.... 34
netdb.h.... 34
netinet/in.h.... 34
new.... 34
new.h.... 35
nlist.h.... 35
nl_langinfo.h.... 36
nl_types.h.... 36
poll.h.... 36
pthread.h.... 36
pwd.h.... 37
re_comp.h.... 37
regex.h.... 37
regexp.h.... 38
resolv.h.... 38
rexec.h.... 38
saiucv.h.... 38
search.h.... 38
setjmp.h.... 38
signal.h.... 38
spawn.h.... 39
spc.h.... 39
stdarg.h.... 39
stddef.h.... 40
stdefs.h.... 40
stdio.h.... 40
stdlib.h.... 42
string.h.... 43
strings.h.... 44
stropts.h.... 44
syslog.h.... 44
sys/acl.h.... 44
sys/__cpl.h.... 44
sys/file.h.... 44
sys/_getipc.h.... 44
sys/ioctl.h.... 44
sys/ipc.h.... 45
sys/layout.h.... 45
sys/mman.h.... 45
sys/__messag.h.... 45
sys/mntent.h.... 45
sys/modes.h.... 45
sys/msg.h.... 45
sys/ps.h.... 45
sys/resource.h.... 45
sys/sem.h.... 45
sys/server.h.... 45
sys/shm.h.... 46
sys/socket.h.... 46
sys/stat.h.... 46
sys/statfs.h.... 46
sys/statvfs.h.... 46
sys/time.h.... 46

sys/timeb.h....46
sys/times.h....46
sys/ttydev.h....46
sys/types.h....46
sys/uio.h....48
sys/un.h....48
sys/__ussos.h....48
sys/utsname.h....48
sys/wait.h....48
sys/__wlm.h....48
tar.h....49
terminat.h....49
termios.h....49
time.h....49
typeinfo....50
typeinfo.h....51
ucontext.h....51
uheap.h....51
ulimit.h....51
unexpect.h....51
unistd.h....51
utime.h....52
utmpx.h....52
varargs.h....52
variant.h....52
wchar.h....53
wcstr.h....54
wctype.h....54
wordexp.h....54
xti.h....54

Chapter 3. Library Functions....61

Names....61
Standards....61
Online Examples....63
Built-in Functions....63
IEEE floating-point....63
External Variables....64
errno....64
daylight....64
getdate_err....64
_loc1....65
loc1....65
loc2....65
locs....65
optarg....65
opterr....65
optind....65
optopt....65
signgam....65
stdin....66
stderr....66
stdout....66
timezone....66
tzname....66
Library Function Descriptions....66
abort() — Stop a Program....66

abs(), absf(), absl() — Calculate Integer Absolute Value....68
accept() — Accept a New Connection on a Socket....69
access() — Determine Whether a File Can be Accessed....71
acos(), acosf(), acosl() — Calculate Arccosine....73
acosh() — Calculate Hyperbolic Arcosine....75
alarm() — Set an Alarm....76
asctime() — Convert Time to Character String....78
asin(), asinf(), asinl() — Calculate Arcsine....79
asinh() — Calculate Hyperbolic Arcsine....81
assert() — Verify Condition....81
atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate Arctangent....83
atanh() — Calculate Hyperbolic Arctangent....84
atexit() — Register Program Termination Function....85
__atoe() — Convert ISO8859-1 String to EBCDIC....87
__atoe_l() — Convert ISO8859-1 Bytes to EBCDIC....87
atof() — Convert Character String to Double....88
atoi() — Convert Character String to Integer....89
atol() — Convert Character String to Long....89
a64l() — Convert Base-64 String Representation to Long Integer....90
basename() — Return the Last Component of a Path Name....91
bcmp() — Compare Bytes in Memory....91
bcopy() — Copy Bytes in Memory....92
bind() — Bind a Name to a Socket....92
brk() — Change Space Allocation....98
bsd_signal() — Establish BSD Version of signal()....99
bsearch() — Search Arrays....100
bzero() — Zero Out Bytes in Memory....102
calloc() — Reserve and Initialize Storage....103
cbrt() — Calculate Cube Root....104
cclass() — Return Characters in a Character Class....104
cds() — Compare Double and Swap....106
cdump() — Request Main Storage Dump....106
ceil(), ceilf(), ceill()— Round Up to Integral Value....107
cfgetispeed() — Determine Input Baud Rate....108
cfgetospeed() — Determine Output Baud Rate....110
cfsetispeed() — Set Input Baud Rate in the Termios....112
cfsetospeed() — Set Output Baud Rate in the Termios....113
chaudit() — Change Audit Flags for a File by Path....114
chdir() — Change the Working Directory....116
chmod() — Change the Mode of a File or Directory....117
chown() — Change the Owner or Group of a File or Directory....120
clearenv() — Clear Environment Variables....122
clearerr() — Reset Error and End-of-File....124
clock() — Determine Processor Time....125
close() — Close a File....127
closedir() — Close a Directory....129
clrmemf() — Clear Memory Files....131
collequiv() — Return List of Equivalent Collating Elements....132
collorder() — Return List of Collating Elements....133
collrange() — Calculate the Range List of Collating Elements....134
colltostr() — Return a String for a Collating Element....136
confstr() — Get Configurable Variables....137
connect() — Connect a Socket....138
cos(), cosf(), cosl()— Calculate Cosine....143
cosh(), coshf(), coshl()— Calculate Hyperbolic Cosine....145
creat() — Create a New File or Rewrite an Existing One....146
cs() — Compare and Swap....148
csid() — Character Set ID for Multibyte Character....149

csnap() — Request a Condensed Dump....150
__csplist() — Retrieve CSP Parameters....151
ctermid() — Generate Path Name for Controlling Terminal....151
ctest() — Start Debug Tool....152
ctime() — Convert Time to a Character String....153
ctrace() — Request a Traceback....155
cuserid() — Return Character Login of the User....156
dbm_clearerr() — Clear Database Error Indicator....157
dbm_close() — Close a Database....158
dbm_delete() — Delete Database Record....158
dbm_error() — Check Database Error Indicator....159
dbm_fetch() — Get Database Content....160
dbm_firstkey() — Get First Key in Database....161
dbm_nextkey() — Get Next Key in Database....162
dbm_open() — Open a Database....163
dbm_store() — Store Database Record....164
difftime() — Compute Time Difference....166
dirname() — Report the Parent Directory of a Path Name....167
div() — Calculate Quotient and Remainder....168
dlclose() — Close a dlopen() object....168
dlerror() — Get Diagnostic Information....169
dlopen() — Gain Access to a Dynamic Link Library (DLL)....170
dlsym() — Obtain the Address of a Symbol from a dlopen() Object....172
dllfree() — Free the Supplied DLL....174
dllload() — Load the DLL and Connect it to the Application....174
dllqueryfn() — Obtain a Pointer to a DLL Function....174
dllqueryvar() — Obtain a Pointer to a DLL Variable....174
dn_comp() — Resolver Domain Name Compression....174
dn_expand() — Resolver Domain Name Expansion....175
dn_find() — Resolver Domain Name Find....176
dn_skipname() — Resolver Domain Name Skipping....177
drand48() — Generate Pseudo-Random Number....178
dup() — Duplicate an Open File Descriptor....180
dup2() — Duplicate an Open File Descriptor to Another....181
ecvt() — Convert Double to String....183
endhostent() — Work with a Host Entry....183
endnetent() — Close Network Information Data Set....184
endprotoent() — Work with a Protocol Entry....185
endservent() — Close Network Services Information Data Sets....185
endutxent() — Close the utmpx Database....186
erand48() — Generate Pseudo-Random Number....187
erf() - erfc() — Calculate Error and Complementary Error Functions....189
__errno2() — Return Reason Code Information....190
__etoa() — Convert EBCDIC String to ISO8859-1....191
__etoa_l() — Convert EBCDIC Bytes to ISO8859-1....191
exec Functions....192
_exit() — End a Process and Bypass the Cleanup....195
exit() — End Program....197
exp(), expf(), expl()— Calculate Exponential Function....199
expm1() — Calculate Exponential Minus One....200
fabs(), fabsf(), fabsl() — Calculate Floating-Point Absolute Value....200
fchaudit() — Change Audit Flags for a File by Descriptor....201
fchmod() — Change the Mode of a File or Directory by Descriptor....203
fchown() — Change the Owner or Group by File Descriptor....204
fclose() — Close File....206
fcntl() — Control Open File Descriptors....207
fcvt() — Convert Double to String....215
fdelrec() — Delete a VSAM Record....216

fdopen() — Associate a Stream with an Open File Descriptor.... 217
feof() — Test End-of-File Indicator.... 219
ferror() — Test for Read/Write Errors.... 220
fetch() — Get a Load Module.... 221
fetchep() — Share Writable Static.... 231
fflush() — Write Buffer to File.... 233
ffs() — Find First Set Bit in an Integer.... 235
fgetc() — Read a Character.... 235
fgetpos() — Get File Position.... 236
fgets() — Read a String from a Stream.... 238
fgetwc() — Get Next Wide Character.... 239
fgetws() — Get a Wide-Character String.... 241
fileno() — Get the File Descriptor from an Open Stream.... 242
fldata() — Retrieve File Information.... 244
flocate() — Locate a VSAM Record.... 246
floor(), floorf(), floorl() — Round Down to Integral Value.... 249
fmod(), fmodf(), fmodl() — Calculate Floating-Point Remainder.... 250
fmtmsg() — Display a Message in the Specified Format.... 251
fnmatch() — Match File Name or Path Name.... 253
fopen() — Open a File.... 254
fork() — Create a New Process.... 259
fpathconf() — Determine Configurable Path Name Variables.... 261
fprintf() - printf() - sprintf() — Format and Write Data.... 264
fputc() — Write a Character.... 272
fputs() — Write a String.... 273
fputwc() — Output a Wide Character.... 274
fputws() — Output a Wide-Character String.... 276
fread() — Read Items.... 277
free() — Free a Block of Storage.... 278
freeaddrinfo() — Free Addrinfo Storage.... 279
freopen() — Redirect an Open File.... 280
frexp() — Extract Mantissa and Exponent of the Floating-Point Value.... 282
fscanf() - scanf() - sscanf() — Read and Format Data.... 283
fseek() — Change File Position.... 292
fsetpos() — Set File Position.... 294
fstat() — Get Status Information about a File.... 296
fstatvfs() — Get File System Information.... 297
fsync() — Write Changes to Direct-Access Storage.... 299
ftell() — Get Current File Position.... 300
ftime() — Get the Date and Time.... 302
ftok() — Generate an Interprocess Communication (IPC) Key.... 302
ftruncate() — Truncate a File.... 303
ftw() — Traverse a File Tree.... 305
fupdate() — Update a VSAM Record.... 307
fwrite() — Write Items.... 308
gai_strerror() — Address and Name Information Error Description.... 309
gamma() — Calculate Gamma Function.... 310
gcvt() — Convert Double to String.... 311
getc() - getchar() — Read a Character.... 312
getaddrinfo() — Get Address Information.... 313
getclientid() — Get Identifier for Calling Application.... 316
getcontext() — Get User Context.... 318
getcwd() — Get Path Name of the Working Directory.... 320
getdate() — Convert User Format Date and Time.... 322
getdtablesize() — Get File Descriptor Table Size.... 324
getegid() — Get the Effective Group ID.... 325
getenv() — Get Value of Environment Variables.... 326
geteuid() — Get the Effective User ID.... 327

getgid() — Get the Real Group ID....329
getgrgid() — Access the Group Database by ID....330
getgrnam() — Access the Group Database by Name....331
getgroups() — Get a List of Supplementary Group IDs....332
getgroupsbyname() — Get Supplementary Group IDs by User Name....334
gethostbyaddr() — Get a Host Entry by Address....335
gethostbyname() — Get a Host Entry by Name....337
gethostent() — Get Next Host Entry....339
gethostid() — Get Unique Identifier of Current Host....340
gethostname() — Get Name of Host Processor....340
getibmsockopt() — Get Options Associated with a Bulk Mode Socket....341
getlogin() — Get the User Login Name....344
getmccoll() — Get Next Collating Element from String....345
getnameinfo() — Get Name Information....346
getnetbyaddr() — Get a Network Entry by Address....347
getnetbyname() — Get a Network Entry by Name....348
getnetent() — Get Next Network Entry....350
getopt() — Parse Command Option....351
getpagesize() — Get the Current Page Size....352
getpass() — Read a Character String Without Echo....352
getpeername() — Get Name of Peer Connected to a Socket....353
getpgrp() — Get the Process Group ID....354
getpid() — Get the Process ID....355
getppid() — Get the Parent Process ID....356
getprotobyname() — Get a Protocol Entry by Name....358
getprotobynumber() — Get a Protocol Entry by Number....359
getprotoent() — Get Next Protocol Entry....360
getpwnam() — Access the User Database by User Name....361
getpwuid() — Access the User Database by User ID....362
gets() — Read a String....364
getservbyname() — Get a Server Entry by Name....365
getservbyport() — Get a Service Entry by Port....366
getservent() — Get Next Service Entry....367
getsockname() — Get Name of a Socket....368
getsockopt() — Get Options Associated with a Socket....370
getstablesize() — Get Socket Table Size....374
getsubopt() — Parse Suboption Arguments....375
getsyntx() — Return LC_SYNTAX Characters....376
gettimeofday() — Get Date and Time....377
getuid() — Get the Real User ID....378
getutxent() — Read Next Entry in utmpx Database....379
getutxid() — Search by ID utmpx Database....380
getutxline() — Search by Line utmpx Database....381
getw() — Get a Machine Word from a Stream....383
getwc() — Get a Wide Character....384
getwchar() — Get a Wide Character....385
getwmccoll() — Get Next Collating Element from Wide String....386
givesocket() — Make Specified Socket Available....387
glob() — Generate Path names Matching a Pattern....389
globfree() — Free Storage Allocated by glob()....391
gmtime() — Convert Time to Broken-Down UTC Time....391
hcreate() — Create Hash Search Tables....393
hdestroy() — Destroy Hash Search Tables....394
hsearch() — Search Hash Tables....394
htonl() — Translate Address Host to Network Long....395
htons() — Translate an Unsigned Short Integer into Network Byte Order....396
hypot() — Calculate Hypotenuse....397
ibmsflush() — Flush Application-side Datagram Queue....398

iconv() — Code Conversion....399
iconv_close() — Deallocate Code Conversion Descriptor....401
iconv_open() — Allocate Code Conversion Descriptor....402
if_freenameindex() — Free the Memory Allocated by if_nameindex()....404
if_indextoname() — Map a Network Interface Index to Its Corresponding Name....404
if_nameindex() — Return All Network Interface Names and Indexes....405
if_nametoindex() — Map a Network Interface Name to Its Corresponding Index....406
ilogb() — Integer Unbiased Exponent....406
index() — Search for Character....407
inet_addr() — Translate an Internet Address into Network Order....408
inet_lnaof() — Translate a Local Network Address into Host Byte Order....409
inet_makeaddr() — Create an Internet Host Address....410
inet_netof() — Get Network Number from Internet Host Address....410
inet_network() — Get Network Number from Decimal Host Address....411
inet_ntoa() — Get Decimal Internet Host Address....412
inet_ntop() — Convert Internet Address Format from Binary to Text....413
inet_pton() — Convert Internet Address Format from Text to Binary....414
initstate() — Initialize Generator for random()....415
insque() — Insert an Element into a Doubly Linked List....416
ioctl() — Control Device....417
isalnum() to isxdigit() — Test Integer Value....419
isascii() — Test for 7-bit US-ASCII Character....421
isatty() — Test if Descriptor Represents a Terminal....426
isblank() — Test for Blank Character Classification....427
iscics() — Verify Whether CICS is Running....428
iscntrl() — Test for Control Classification....429
isdigit() — Test for Hexadecimal-Digit Classification....429
isgraph() — Test for Graphic Classification....429
islower() — Test for Lowercase....429
ismccollel() — Identify a Multicharacter Collating Element....429
isnan() — Test for NaN....431
isprint() — Test for Printable Character Classification....431
ispunct() — Test for Punctuation Classification....431
isspace() — Test for Space Character Classification....431
isupper() — Test for Uppercase Letter Classification....431
iswalnum() to iswxdigit() — Test Wide Integer Value....432
iswblank() — Test for Blank Character Classification....433
iswcntrl() — Test for Control Classification....434
iswctype() — Test for Character Property....434
iswdigit() — Test for Hexadecimal-Digit Classification....435
iswgraph() — Test for Graphic Classification....435
iswlower() — Test for Lowercase....436
iswprint() — Test for Printable Character Classification....436
iswpunct() — Test for Punctuation Classification....436
iswspace() — Test for Space Character Classification....436
iswupper() — Test for Uppercase Letter Classification....436
iswxdigit() — Test for Hexadecimal-Digit Classification....436
isxdigit() — Test for Hexadecimal-Digit Classification....436
jrand48() — Generate Pseudorandom Number....436
j0(), j1(), jn() — Bessel Functions of the First Kind....438
kill() — Send a Signal to a Process....439
killpg() — Send a Signal to a Process Group....441
labs() — Calculate Long Absolute Value....443
lcong48() — Initialize Pseudorandom Number....444
ldexp(), ldexpf(), ldexpl() — Multiply by a Power of Two....445
ldiv() — Compute Quotient and Remainder of Integral Division....446
lfind() — Linear Search Routine....447
lgamma() — Log Gamma Function....448

__librel() — Query Release Level.... 449
link() — Create a Link to a File.... 450
listen() — Prepare Server for Incoming Client Requests.... 452
localdtconv() — Date/Time Formatting Convention Inquiry.... 453
localeconv() — Query Numeric Conventions.... 454
localtime() — Convert Time and Correct for Local Time.... 456
lockf() — Record Locking on Files.... 458
log(), logf(), logl() — Calculate Natural Logarithm.... 460
logb() — Calculate Unbiased Exponent.... 461
log1p() — Calculate Natural Log of x plus 1.... 461
log10(), log10f(), log10l() — Calculate Base 10 Logarithm.... 462
_longjmp() — Nonlocal Goto.... 463
longjmp() — Restore Stack Environment.... 465
lrand48() — Generate Pseudorandom Number.... 467
lsearch() — Linear Search and Update.... 468
lseek() — Change the Offset of a File.... 469
lstat() — Get Status of File, External Link, or Symbolic Link.... 471
l64a() — Convert Long to Base-64 String Representation.... 474
makecontext() — Modify User Context.... 475
malloc() — Reserve Storage Block.... 477
maxcoll() — Return Maximum Collating Element.... 479
maxdesc() — Get Socket Numbers to Extend Beyond Default Range.... 479
mblen() — Calculate Length of Multibyte Character.... 480
mbrlen() — Calculate Length of Multibyte Character.... 482
mbrtowc() — Convert a Multibyte Character to a Wide Character.... 484
mbsinit() — Test State Object for Initial State.... 486
mbsrtowcs() — Convert a Multibyte String to a Wide-Character String.... 487
mbstowcs() — Convert Multibyte Characters to Wide Characters.... 489
mbtowc() — Convert Multibyte Character to Wide Character.... 490
memccpy() — Copy Bytes in Memory.... 492
memchr() — Search Buffer.... 492
memcmp() — Compare Bytes.... 493
memcpy() — Copy Buffer.... 495
memmove() — Move Buffer.... 496
memset() — Set Buffer to Value.... 497
mkdir() — Make a Directory.... 498
mkfifo() — Make a FIFO Special File.... 499
mknod() — Make a FIFO or Charcter Special File.... 501
mkstemp() — Make a Unique File Name.... 503
mktemp() — Make a Unique File Name.... 504
mktime() — Convert Local Time.... 505
modf() — Extract Fractional and Integral Parts of a FLoating-Point Value.... 507
mount() — Make a File System Available.... 508
mrand48() — Generate Pseudorandom Number.... 510
msgctl() — Message Control Operations.... 512
msgget() — Get Message Queue.... 513
msgrcv() — Read and Receive a Message.... 515
msgsnd() — Send a Message.... 516
msgxrcv() — Read and Receive an Extended Message.... 518
nextafter() — Calculate Next Representable Double Float.... 520
nftw() — Traverse a File Tree.... 520
nl_langinfo() — Retrieve Locale Information.... 522
nrand48() — Pseudo-Random Number Generator.... 523
ntohl() — Translate a Long Integer into Host Byte Order.... 525
ntohs() — Translate an Unsigned Short Integer into Host Byte Order.... 526
open() — Open a File.... 527
opendir() — Open a Directory.... 531
pathconf() — Determine Configurable Path Name Variables.... 533

pause() — Suspend a Thread Pending a Signal..... 535
pclose() — Close a Pipe Stream to or from a Process..... 537
perror() — Print Error Message..... 538
pipe() — Create an Unnamed Pipe..... 539
popen() — Initiate a Pipe Stream to or from a Process..... 541
pow() — Raise to Power..... 543
printf() — Format and Write Data..... 544
pthread_attr_destroy() — Destroy the Thread Attributes Object..... 544
pthread_attr_getdetachstate() — Get the Detach State Attribute..... 545
pthread_attr_getstacksize() — Get the Thread Attribute Stacksize..... 547
pthread_attr_getweight_np() — Get Weight of Thread Attribute Object..... 548
pthread_attr_init() — Initialize a Thread Attribute Object..... 549
pthread_attr_setdetachstate() — Set the Detach State Attribute Object..... 551
pthread_attr_setstacksize() — Set the Stacksize Attribute Object..... 552
pthread_attr_setweight_np() — Set Weight of Thread Attribute Object..... 554
pthread_cancel() — Cancel a Thread..... 555
pthread_cleanup_pop() — Remove a Cleanup Handler..... 557
pthread_cleanup_push() — Establish a Cleanup Handler..... 558
pthread_cond_broadcast() — Broadcast a Condition..... 560
pthread_cond_destroy() — Destroy the Condition Variable Object..... 561
pthread_cond_init() — Initialize a Condition Variable..... 562
pthread_cond_signal() — Signal a Condition..... 563
pthread_cond_timedwait() — Wait on a Condition Variable..... 564
pthread_cond_wait() — Wait on a Condition Variable..... 566
pthread_condattr_destroy() — Destroy Condition Variable Attribute Object..... 568
pthread_condattr_getkind_np() — Get Kind Attribute from a Condition Variable Attribute Object 569
pthread_condattr_init() — Initialize a Condition Attribute Object..... 571
pthread_condattr_setkind_np() — Set Kind Attribute from a Condition Variable Attribute Object. 572
pthread_create() — Create a Thread..... 573
pthread_detach() — Detach a Thread..... 575
pthread_equal() — Compare Thread IDs..... 576
pthread_exit() — Exit a Thread..... 577
pthread_getspecific() — Get the Thread-Specific Value for a Key..... 579
pthread_join() — Wait for a Thread to End..... 581
pthread_join_d4_np() — Wait for a Thread to End (.4a)..... 582
pthread_key_create() — Create Thread-Specific Data Key..... 584
pthread_kill() — Send a Signal to a Thread..... 586
pthread_mutex_destroy() — Delete a Mutex Object..... 588
pthread_mutex_init() — Initialize a Mutex Object..... 589
pthread_mutex_lock() — Wait for a Lock on a Mutex Object..... 590
pthread_mutex_trylock() — Attempt to Lock a Mutex Object..... 592
pthread_mutex_unlock() — Unlock a Mutex Object..... 593
pthread_mutexattr_destroy() — Destroy a Mutex Attribute Object..... 595
pthread_mutexattr_getkind_np() — Get Kind from a Mutex Attribute Object..... 596
pthread_mutexattr_init() — Initialize a Mutex Attribute Object..... 598
pthread_mutexattr_setkind_np() — Set Kind for a Mutex Attribute Object..... 599
pthread_once() — Invoke a Function Once..... 601
pthread_self() — Get the Caller..... 603
pthread_setintr() — Set the Cancelability State of a Thread..... 604
pthread_setintrtype() — Set the Cancelability Type of a Thread..... 606
pthread_setspecific() — Set the Thread-Specific Value for a Key..... 608
pthread_testintr() — Establish a Cancelability Point..... 610
pthread_yield() — Release the Processor to Other Threads..... 611
putc() - putchar() — Write a Character..... 613
putenv() — Change or Add an Environment Variable..... 614
puts() — Write a String..... 615
pututxline() — Write Entry to utmpx Database..... 616
putw() — Put a Machine Word on a Stream..... 618

putwc() — Output a Wide Character..... 619
putwchar() — Output a Wide Character to Standard Output..... 620
qsort() — Sort Array..... 622
raise() — Raise Signal..... 623
rand() — Generate Random Number..... 626
random() — Generate a Better Random Number..... 627
read() — Read From a File 628
readdir() — Read an Entry from a Directory..... 632
readlink() — Read the Value of a Symbolic Link..... 633
readv() — Read Data on a Socket and Store in a Set of Buffers..... 635
realloc() — Change Reserved Storage Block Size..... 637
realpath() — Resolve Path Name..... 639
recv() — Receive Data on a Socket..... 640
recvfrom() — Receive Messages on a Socket..... 642
recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers..... 645
re_comp() — Compile Regular Expression..... 648
re_exec() — Match Regular Expression..... 650
regcomp() — Compile Regular Expression..... 651
regerror() — Return Error Message..... 653
regexec() — Execute Compiled Regular Expression..... 655
regfree() — Free Memory for Regular Expression..... 657
release() — Delete a Load Module..... 657
remainder() — Remainder Function..... 658
remove() — Delete File..... 659
remque() — Remove an Element from a Doubly Linked List..... 660
rename() — Rename File..... 661
res_init() — Domain Name Resolver Initialization..... 663
res_mkquery() — Make Resolver Query for Domain Name Servers (DNS)..... 665
res_query() — Resolver Query for Domain Name Servers (DNS)..... 666
res_querydomain() — Build Domain Name and Resolver Query 667
res_search() — Resolver Query for Domain Name Servers (DNS) With Optional Rules..... 668
res_send() — Send Resolver Query for Domain Name Servers (DNS)..... 669
rewind() — Set File Position to Beginning of File..... 671
rewinddir() — Reposition a Directory Stream to the Beginning..... 672
rindex() — Search for a Character..... 673
rint() — Round to Nearest Integral Value..... 674
rmdir() — Remove a Directory..... 674
rpmatch() — Test for a Yes/No Response Match..... 676
sbrk() — Change Space Allocation..... 677
scalb() — Load Exponent..... 678
scanf() — Read and Format Data..... 678
seed48() — Initialize Pseudorandom Number..... 679
select() — Monitor Activity on Files/Sockets and Message Queues..... 680
selectex() — Monitor Activity on Files/Sockets and Message Queues..... 685
semctl() — Perform Semaphore Control Operations..... 687
semget() — Get a Set of Semaphores..... 690
semop() — Perform Semaphore Operations..... 692
send() — Send Data on a Socket..... 694
sendmsg() — Send Messages on a Socket..... 697
sendto() — Send Data on a Socket..... 700
setbuf() — Control Buffering..... 703
setcontext() — Restore User Context..... 704
setegid() — Set the Effective Group ID..... 706
setenv() — Add, Delete, and Change Environment Variables..... 708
seteuid() — Set the Effective User ID..... 710
setgid() — Set the Group ID..... 711
sethostent() — Open Host Information Data Set..... 712
setibmsockopt() — Set Options Associated with a Bulk Mode Socket..... 713

_setjmp() — Set Jump Point for a Nonlocal Goto.... 717
setjmp() — Preserve Stack Environment.... 718
setlocale() — Set Locale.... 721
setnetent() — Open Network Information Data Set.... 728
setpgid() — Set Process Group ID for Job Control.... 729
setpgrp() — Set Process Group ID.... 731
setprotoent() — Open Protocol Information File.... 731
setservent() — Open Network Services Information File.... 732
setsid() — Create Session, Set Process Group ID.... 733
setsockopt() — Set Options Associated with a Socket.... 734
setstate() — Change Generator for random().... 739
setuid() — Set the Effective User ID.... 739
setutxent() — Reset to Start of utmpx Database.... 741
setvbuf() — Control Buffering.... 742
shmat() — Attach Shared Memory Segment.... 743
shmctl() — Shared Memory Control Operations.... 744
shmdt() — Detach Shared Memory Segment.... 745
shmget() — Get a Shared Memory Segment.... 746
shutdown() — Shut Down All or Part of a Duplex Connection.... 748
sigaction() — Examine or Change a Signal Action.... 749
sigaddset() — Add a Signal to the Signal Mask.... 754
sigaltstack() — Set and/or Get Signal Alternate Stack Context.... 756
sigdelset() — Delete a Signal from the Signal Mask.... 757
sigemptyset() — Initialize a Signal Mask to Exclude All Signals.... 758
sigfillset() — Initialize a Signal Mask to Include All Signals.... 760
sighold() — Add a Signal to a Thread.... 761
sigignore() — Set Disposition to Ignore a Signal.... 762
siginterrupt() — Allow Signals to Interrupt Functions.... 762
sigismember() — Test If a Signal Is in a Signal Mask.... 763
siglongjmp() — Restore the Stack Environment and Signal Mask.... 764
signal() — Handle Interrupts.... 767
__signgam() — Return __signgam Reference.... 771
sigpause() — Unblock a Signal and Wait for a Signal.... 771
sigpending() — Examine Pending Signals.... 772
sigprocmask() — Examine or Change a Thread.... 773
sigrelse() — Remove a Signal from a Thread.... 776
sigset() — Change a Signal Action and/or a Thread.... 776
sigsetjmp() — Save Stack Environment and Signal Mask.... 778
sigstack() — Set and/or Get Signal Stack Context.... 780
sigsuspend() — Change Mask and Suspend the Thread.... 781
sigwait() — Wait for an Asynchronous Signal.... 784
sin() — Calculate Sine.... 785
sinh() — Calculate Hyperbolic Sine.... 786
sleep() — Suspend Execution of a Process.... 787
sock_debug() — Provide OE Syscall Tracing Facility.... 789
sock_debug_bulk_perf0() — Produce a Report When a Socket Configured.... 789
sock_do_bulkmode() — Use Bulk Mode for Messages Read by Socket.... 790
sock_do_teststor — Check for Attempt to Access Storage.... 791
socket() — Create a Socket.... 791
spawn() - spawnp() — Spawn a New Process.... 795
sprintf() — Format and Write Data to Buffer.... 799
sqrt() — Calculate Square Root.... 799
srand() — Set Seed for rand() Function.... 800
srandom() — Use Seed to Initialize Generator for random().... 801
srand48() — Initialize Pseudorandom Number.... 801
sscanf() — Read and Format Data from Buffer.... 803
stat() — Get File Information.... 803
statvfs() — Get File System Information.... 806

strcasecmp() — Compare Case-Insensitive Strings.....809
strcat() — Concatenate Strings.....809
strchr() — Search for Character.....811
strcmp() — Compare Strings.....812
strcoll() — Compare Strings.....813
strcpy() — Copy String.....815
strcspn() — Compare Strings.....816
strdup() — Duplicate a String.....817
strerror() — Get Pointer to Runtime Error Message.....817
strfmon() — Convert Monetary Value to String.....818
strftime() — Convert to Formatted Time.....821
strlen() — Determine String Length.....825
strncasecmp() — Compare Case-Insensitive Strings.....826
strncat() — Concatenate Strings.....827
strncmp() — Compare Strings.....828
strncpy() — Copy String.....830
strpbrk() — Find Characters in String.....831
strptime() — Date and Time Conversion.....832
strrchr() — Find Last Occurrence of Character in String.....836
strspn() — Search String.....837
strstr() — Locate Substring.....838
strtocoll() — Return Collating Element for String.....839
strtod() — Convert Character String to Double.....840
strtok() — Tokenize String.....842
strtol() — Convert Character String to Long.....843
strtoul() — Convert String to Unsigned Integer.....845
strxfrm() — Transform String.....847
swab() — Copy and Swap Bytes.....849
swapcontext() — Save and Restore User Context.....849
swprintf() — Write Wide Characters to a Wide-Character Array.....851
swscanf() — Read a Wide-Character String.....852
symlink() — Create a Symbolic Link to a Path Name.....854
sysconf() — Determine System Configuration Options.....856
system() — Execute a Command.....858
takesocket() — Acquire a Socket from Another Program.....860
tan() — Calculate Tangent.....861
tanh() — Calculate Hyperbolic Tangent.....862
tcdrain() — Wait Until Output Has Been Transmitted.....863
tcflow() — Suspend or Resume Data Flow on a Terminal.....864
tcflush() — Flush Input or Output on a Terminal.....867
tcgetattr() — Get the Attributes for a Terminal.....869
tcgetpgrp() — Get the Foreground Process Group ID.....870
tcperror() — Print Error Messages of a Socket Function.....872
tcsendbreak() — Send a Break Condition to a Terminal.....873
tcsetattr() — Set the Attributes for a Terminal.....874
tcsetpgrp() — Set the Foreground Process Group ID.....880
tdelete() — Delete Binary Tree Node.....882
tempnam() — Generate a Temporary File Name.....883
tfind() — Find Binary Tree Node.....884
time() — Determine Current Time.....885
times() — Get Process and Child Process Times.....886
tmpfile() — Create Temporary File.....888
tmpnam() — Produce Temporary File Name.....889
toascii() — Translate Integer to a 7-Bit ASCII Character.....890
tolower(), toupper() — Convert Character Case.....895
towlower(), towupper() — Convert Wide-Character Case.....896
tsearch() — Search Binary Tree.....896
ttyname() — Get the Name of a Terminal.....897

ttyslot() — Find the Slot in the utmpx File of the Current User.... 898
twalk() — Walk Binary Tree.... 899
tzset() — Set the Time Zone.... 900
umask() — Set or Display the File Mode Creation Mask.... 902
umount() — Remove a Virtual File System.... 903
uname() — Display Current Operating System Name.... 905
ungetc() — Push Character onto Input Stream.... 907
ungetwc() — Push a Wide Character onto a Stream.... 908
unlink() — Remove a Directory Entry.... 910
usleep() — Suspend Execution for an Interval.... 911
utime() — Set File Access and Modification Times.... 912
utimes() — Set File Access and Modification Times.... 914
__utmpxname() — Change the utmpx Database Name.... 915
va_arg(), va_end(), va_start() — Access Function Arguments.... 916
valloc() — Allocate Page-Aligned Memory.... 919
vfork() — Create a New Process.... 919
vfprintf() — Format and Print Data to Stream.... 922
vprintf() — Format and Print Data to stdout.... 923
vsprint() — Format and Print Data to Buffer.... 924
vswprintf() — Write Wide Characters.... 925
wait() — Wait for a Child Process to End.... 927
waitid() — Wait for Child Process to Change State.... 929
waitpid() — Wait for a Specific Child Process to End.... 930
wcrtomb() — Convert a Wide Character to a Multibyte Character.... 933
wcscat() — Append to Wide-Character String.... 935
wcschr() — Search for Wide-Character Substring.... 936
wcscmp() — Compare Wide-Character Strings.... 937
wcscoll() — Language Collation String Comparison.... 938
wcscpy() — Copy Wide-Character String.... 940
wcscspn() — Find Offset of First Wide-Character Match.... 941
wcsftime() — Format Date and Time.... 942
wcsid() — Character Set ID for Wide Character.... 943
wcslen() — Calculate Length of Wide-Character String.... 944
wcsncat() — Append to Wide-Character String.... 945
wcsncmp() — Compare Wide-Character Strings.... 946
wcsncpy() — Copy Wide-Character String.... 948
wcspbrk() — Locate First Wide Character in String.... 949
wcsrchr() — Locate Last Wide Character in String.... 950
wcsrtombs() — Convert Wide-Character String to Multibyte String.... 951
wcsspn() — Search for Wide Characters in a String.... 953
wcsstr() — Locate a Wide-Character Sequence.... 954
wcstod() — Convert Wide-Character String to a Double Floating-Point.... 955
wcstok() — Break a Wide-Character String into Tokens.... 957
wcstol() — Convert a Wide-Character String to a Long Integer.... 959
wcstombs() — Convert a Wide-Character String to Multibyte Character.... 960
wcstoul() — Convert a Wide-Character String to an Unsigned Long.... 962
wcswcs() — Locate Wide-Character Substring in Wide Character String.... 963
wcswidth() — Determine the Display Width of a Wide-Character String.... 964
wcsxfrm() — Transform a Wide-Character String.... 965
wctob() — Convert Wide Character to Byte.... 966
wctomb() — Convert Wide Character to Multibyte Character.... 967
wctype() — Obtain Handle for Character Property Classification.... 968
wcwidth() — Determine the Display Width of a Wide Character.... 969
w_getpsent() — Get Process Data.... 970
wordexp() — Perform Shell Word Expansions.... 972
wordfree() — Perform Shell Word Expansions.... 975
write() — Write Data on a File or Socket.... 975
writev() — Write Data on a Socket from an Array.... 979

w_statvfs() — Get File System Status.... 982
y0(), y1(), yn() — Bessel Functions of the Second Kind....983
Library Functions for the System Programming C Facilities.... 984

Appendix A. Table of C Macros.... 985

Notices....1001
Programming Interface Information....1002
Trademarks....1002
Terms and Conditions for Product Documentation.... 1002
IBM Online Privacy Statement.... 1003
Standards....1003

Bibliography.... 1005
Where to Get z/VM Information.... 1005
z/VM Base Library....1005
z/VM Facilities and Features.... 1006
Prerequisite Products....1008
Related Products.... 1008

Index.... 1009

Figures

1. Libraries Comprising Language Environment..... 4

2. Overlap of C Standards and Extensions.....62

3. Program Flow of a Fetchable Module..... 223

Tables

1. IBM XL C/C++ for z/VM Features..... 10

2. Feature Test Macros and Standards..... 13

3. Definitions in float.h..... 25

4. Definitions of Resource Limits..... 28

5. Elements of the lconv Structure..... 30

6. Symbolic Constants defined in sys/__cpl.h..... 44

7. Symbolic Constants defined in xti.h..... 55

8. Codes to Set Baud Rate..... 108

9. Struct f_cnvrt Element Descriptions..... 210

10. Elements Returned in fldata_t Data Structure..... 245

11. Position Options Parameter for flocate()..... 247

12. Values for the Positional Parameter..... 255

13. Keyword Parameters for File Mode..... 256

14. Flag Characters for fprintf() Family..... 265

15. Precision Argument in fprintf() family..... 266

16. Type Characters and their Meanings..... 267

17. Conversion Specifiers in fscanf() and scanf()..... 285

18. Fields of the tm structure..... 392

19. Characters for which isascii() returns nonzero..... 422

20. Elements of the stat Structure..... 471

21. Control Signals..... 624

22. Values for Category Arguments of setlocale..... 721

23. Return String as Determined by Category and Locale Values..... 726

24. Control Signals..... 750

25. Control Signals Supported Under C only — POSIX(ON)..... 767

26. Control Signals Supported Under C — POSIX(OFF)..... 769

27. Values Returned in the stat Structure..... 804

28. Values Returned in the statvfs Structure..... 807

29. Conversion Specifiers Used by strftime()..... 822

30. Conversion Specifiers Used by strptime()..... 833

31. Modified Directives Used by strptime()..... 834

32. Variables Stored in Structure..... 971

33. z/VM C Macros..... 985

About This Document

This document provides descriptions of the IBM XL C/C++ language header files, library functions, macros, and built-in functions supported under the z/VM operating system and Language Environment®.

Intended Audience

This information is intended for programmers who want to write C and C++ applications on the z/VM platform that conform to ANSI or POSIX standards.

Where to Find More Information

This document is intended to be used in conjunction with the following documents:

- z/OS® XL C/C++ documents (included in the z/OS library)
- z/VM Language Environment documents (included in the z/VM library)
- z/OS Language Environment documents (included in the z/OS library)
- z/VM OpenExtensions documents (included in the z/VM library)
- z/VM and z/OS Program Management Binder documents (included in the z/VM library)

For more information, see the "Bibliography" on page 1005 and IBM Documentation - z/OS (https://www.ibm.com/docs/en/zos).

Links to Other Documents and Websites

The PDF version of this document contains links to other documents and websites. A link from this document to another document works only when both documents are in the same directory or database, and a link to a website works only if you have access to the Internet. A document link is to a specific edition. If a new edition of a linked document has been published since the publication of this document, the linked document might not be the latest edition.

© Copyright IBM Corp. 2002, 2024

How to provide feedback to IBM

We welcome any feedback that you have, including comments on the clarity, accuracy, or completeness of the information. See How to send feedback to IBM for additional information.

© Copyright IBM Corp. 2002, 2024

Summary of Changes for XL C/C++ for z/VM: Runtime Library Reference

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

SC09-7624-74, XL C/C++ for z/VM, 1.3 (September 2024)

This edition supports z/VM 7.4. It includes terminology, maintenance, and editorial changes. Note that the publication number suffix (-74) indicates the z/VM release to which this edition applies.

SC09-7624-73, XL C/C++ for z/VM, 1.3 (December 2023)

This edition includes terminology, maintenance, and editorial changes.

SC09-7624-73, XL C/C++ for z/VM, 1.3 (September 2023)

This edition includes terminology, maintenance, and editorial changes.

SC09-7624-73, XL C/C++ for z/VM, 1.3 (May 2023)

This edition supports z/VM 7.3. It includes terminology, maintenance, and editorial changes. Note that the publication number suffix (-73) indicates the z/VM release to which this edition applies.

The following topics are updated:

- “close() — Close a File” on page 127
- “givesocket() — Make Specified Socket Available” on page 387
- “mknod() — Make a FIFO or Charcter Special File” on page 501
- “open() — Open a File” on page 527
- “swscanf() — Read a Wide-Character String” on page 852
- “tcdrain() — Wait Until Output Has Been Transmitted” on page 863
- “tcflow() — Suspend or Resume Data Flow on a Terminal” on page 864
- “tcflush() — Flush Input or Output on a Terminal” on page 867
- “tcsetattr() — Set the Attributes for a Terminal” on page 874

Language Environment upgrade

The z/VM Language Environment runtime libraries have been upgraded to z/OS 2.5 equivalence.

© Copyright IBM Corp. 2002, 2024

Chapter 1. About IBM XL C/C++ for z/VM

IBM XL C/C++ for z/VM is the language-centered C/C++ application development environment on the z/VM platform. It is a z/VM-enabled version of z/OS XL C/C++.

IBM XL C/C++ for z/VM includes:

- The C/C++ compiler (referred to as the XL C/C++ compiler)

  The XL C/C++ compiler can compile C and C++ programs. The common features and the specific features of the C compiler function and the C++ compiler function are described in this reference information.
- C/C++ application development utilities
- Support for the Standard C++ Library and the Standard Template Library that are available with the base z/VM operating system.

You will also need the XL C/C++ runtime library to compile and run your C and C++ applications. The XL C/C++ runtime library is packaged with Language Environment.

XL C/C++ Compiler

The following sections describe the C and C++ language and the XL C/C++ compiler.

The C Language

The C language is a general purpose, function-oriented programming language that allows a programmer to create applications quickly and easily. C provides high-level control statements and data types as do other structured programming languages, and it also provides many of the benefits of a low-level language. Using the C language, you can write portable code conforming to the following standards: ANSI or POSIX.

IBM offers the C language on other platforms, such as the AIX®, OS/400®, VSE/ESA, OS/390®, and z/OS operating systems.

The C++ Language

The C++ language is based on the C language and includes all of the advantages of C previously listed. In addition, C++ also supports object-oriented concepts, type genericity or templates, and an extensive library. For a detailed description of the differences between C and C++, see *z/OS: XL C/C++ Language Reference (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/cbclx01_v2r5.pdf)*.

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 C Compiler and C++ Compiler Functions

The C compiler and C++ compiler functions of the XL C/C++ compiler, when used with Language Environment, provide you with features such as the following:

- Optimization support:

© Copyright IBM Corp. 2002, 2024

  - Algorithms to take advantage of S/390® architecture to get better optimization for speed and use of computer resources through the `OPTIMIZE` compiler option.
  - The `OPTIMIZE` compiler option, which instructs the compiler to optimize the machine instructions it generates to produce faster-running object code, which improves application performance at run time.
- DLLs (Dynamic Link Libraries) to share parts among applications or parts of applications, and dynamically link to exported varaibles and functions at run time.

  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 refers to a function or variable that resides in a DLL, XL C/C++ generates code to load the DLL and access the functions and variables within it. This is called load-on-reference. Alternatively, your program can use C library functions to load a DLL and look up the address of functions and variables within it. This is called load-on-demand. Your application explicitly controlls 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 saved in a shared segment and run by multiple users ended simultaneously uses less storage. It also reduces processor I/O when the program starts up, and improves program performance by reducing the transfer of data to auxiliary storage. Programmers using the C compiler function of the XL C/C++ compiler can design programs that are naturally reentrant. For those programs that are not naturally reentrant, C programmers can use constructed reentrancy by compiling them with the `RENT` option and using the Language Environment prelinker and program management binder to make them reentrant. The C++ compiler function of the XL C/C++ compiler always ensures that C++ programs are reentrant.
- INLINE compiler option

  Additional optimization capabilities are available with the INLINE compiler option.
- Locale-based internationalization support defined in *ISO/IEC DIS 9945-2:1922/IEEE POSIX 1003.2-1992, Draft 12*.
- The ability to call and be called by other languages such as assembler, COBOL, and PL/1, to enable programmers to integrate z/OS C/C++ code with existing applications.
- Support for the following standards at the system level:
  - A subset of 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*, with the exception of the fork() function, which is not fully compliant with the standard.
  - A subset of *IEEE POSIX 1003.1a, Draft 6, July 1991*
  - *IEEE Portable Operating System Interface (POSIX) Part 2, P1003.2, Draft 12*
  - A subset of *IEEE POSIX 1003.1c, Draft 6, February 1992* (Former name: *POSIX 1003.4a*)
  - A subset of *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.
  - A subset of *X/Open CAE Specification, Network Services, Issue 4*
- Year 2000 support
- Support for the Euro currency

Specific Features of the z/VM C Compiler Function

In addition to the features common to the C and C++ compiler functions of the XL C/C++ compiler, the C compiler function provides you with the following capabilities:

- The ability to write portable code that supports the following standards:
  - All elements of the joint ISO and IEC standard: *ISO/IEC 9899:1990 (E)*
  - *ANSI/ISO-IEC 9899:1990[1992]*(formerly *ANSI X3J11.159–1989 C*)
  - *X/Open Specification Programming Language Issue 3, Common Usage C*
  - *FIPS-151*
- Extensions of the standard C language definitions to provide programmers with support for the z/VM environment, such as fixed-point (packed) decimal data support.

Specific Features of the z/VM C++ Compiler Function

In addition to the features common to the C and C++ compiler functions of the XL C/C++ compiler, the C++ compiler function supports the *International Standard for the C++ Programming Language (ISO/IEC 14882:1998)* specification.

IBM XL C/C++ for z/VM Runtime Library

When compiling and running your C or C++ application, you need to be aware of the following conditions:

- If you have an existing C application that contains only C coding and you need to recompile, you can use the IBM C for VM/ESA compiler, the IBM C/C++ for z/VM compiler, or the IBM XL C/C++ for z/VM compiler:
  - If you use the IBM C for VM/ESA compiler, you can use the IBM C for VM/ESA runtime library, the IBM C/C++ for z/VM runtime library, or the IBM XL C/C++ for z/VM runtime library.
  - If you use the C/C++ for z/VM compiler, you can use the IBM C/C++ for z/VM runtime library or the IBM XL C/C++ for z/VM runtime library.
  - If you use the IBM XL C/C++ for z/VM compiler, you must use the IBM XL C/C++ for z/VM runtime library.
- If you have an existing C++ application and you need to recompile, you should use the IBM XL C/C++ for z/VM compiler and the IBM XL C/C++ for z/VM runtime library.
- If you are writing a new C or C++ application, you should use the IBM XL C/C++ for z/VM compiler and the IBM XL C/C++ for z/VM runtime library.

Class Libraries

IBM XL C/C++ for z/VM provides the following class library, which is thread-safe:

- C++ Standard Library, including the Standard Template Library (STL) and other library features of ISO C++ 1998

The C++ Standard Library includes the following:

- The C++ Standard I/O Stream Library for performing input and output (I/O) operations
- The C++ Standard Complex Mathematics Library for manipulating complex numbers
- The Standard Template Library (STL), which is composed of C++ template-based algorithms, container classes, iterators, localization objects, and the string class.

See *z/OS: XL C/C++ Compiler and Run-Time Migration Guide for the Application Programmer* for more details on the components of this library.

Utilities

The IBM XL C/C++ for z/VM compiler provides the following utilities:

- The CXXFILT utility to map C++ mangled names to the original source.

- The DSECT Conversion utility to convert descriptive assembler DSECTs into IBM XL C/C++ for z/VM data structures.
- The localedef utility to read the locale definition file and produce a locale object that the locale-specific library functions can use.

Language Environment provides:

- The Object Library Utility (C370LIB), which updates C/370 CMS text libraries. C370LIB must be used to update a TXTLIB with TEXT files produced by compiling C programs with the LONGNAME option. (The CMS TXTLIB command may not be used to do this directly, and an error can result if this is attempted.)
- The prelinker that combines object modules that comprise an IBM XL C/C++ for z/VM application, to produce a single object module.

Language Environment

IBM XL C/C++ for z/VM exploits the runtime library and common library of base routines available with z/VM and the C/C++ Language Environment for z/VM (referred to as Language Environment.) The C/C++ Language Environment for z/VM is based on z/OS Language Environment.

The following libraries are provided with Language Environment product. Language Environment establishes a common runtime environment and common runtime services for language products, user programs, and other products.

Figure 1. Libraries Comprising Language Environment

The common execution environment is made up of data items and services performed by library routines available to a particular application running in the environment. The services that Language Environment provides to your application may include:

- Services that satisfy basic requirements common to most applications. These include support for the initialization and termination of applications, allocation of storage, and inter-language communication (ILC) and condition handling.
- Extended services often needed by applications. These functions are contained 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 the execution, performance tuning, performance, and diagnosis of your application.
- Access to operating system services—OpenExtensions services are available to an application programmer or program through the IBM XL C/C++ for z/VM language bindings.
- Access to language-specific library routines, such as the IBM XL C/C++ for z/VM functions.

About Prelinking, Linking, and Binding

Note: The prelinker is supported only for IBM C for VM/ESA. It is not supported for IBM XL C/C++ for z/VM.

When describing the process to build an application, this document refers to the bind step.

Normally the Program Management Binder is used to perform the bind step. However, in many cases the prelink and link steps can be used in place of the bind step. When they cannot be substituted, and the Program Management binder alone must be used, it will be stated. For more information about the bind, prelink, and link steps, see the *XL C/C++ for z/VM: User's Guide*.

The terms bind and link have multiple meanings.

- With respect to building an application:

  In both instances, the Program Management Binder is performing the actual processing of converting the object file(s) into the application executable module.

  Object files with longname symbols, reentrant writable static symbols, and DLL-style function calls require additional processing to build global data for the application.

  The term *link* refers to the case where the binder does not perform this addtional processing due to one of the following:

  – The processing is not required because none of the object files in the application use constructed reentrancy, use long names, are DLL, or are C++.

  – The processing is handled by executing the prelinker step before running the binder.

- With respect to executing code in an application:

  The linkage definition refers to the program call linkage between program functions and methods. This includes the passing of control and parameters. For more information about linkage specification, see *z/OS: XL C/C++ Language Reference (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/cbclx01_v2r5.pdf)*.

  Some platforms have a single linkage convention. S/390 has a number of linkage conventions, including standard operating system linkage, Extra Performance Linkage (XPLINK), and different non-XPLINK linkage conventions for C and C++.

The term *bind* refers to the case where the binder is required to perform this processing.

Notes on the Prelinking Process

IBM recommends using the binder without the prelinker whenever possible.

The prelinker was designed to process long names and support constructed reentrancy in earlier versions of the C compiler.

The binder is designed to include the function of the prelinker, the linkage editor, the loader, and a number of APIs to manipulate the program object. Thus, the binder is a superset of the linkage editor. Its functionality provides a high level of compatibility with the prelinker and linkage editor, but provides additional functionality in some areas. Generally, the terms binding and linking are interchangeable. For more information on the compatibility between the binder, and the linker and prelinker, see *z/OS MVS Program Management: User's Guide and Reference*.

The Program Management Binder

The binder provided with z/VM combines the object modules, load modules, and program objects comprising an application. It produces a single z/VM 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 BFS or CMS file.

If your program contains C++ code, or C code compiled with RENT, LONGNAME or DLL, the binder cannot write it to a LOADLIB.

The binder has the following advantages over the prelinker:

- Faster rebinds when recompiling and rebinding a few of your source files
- Rebinding at the single compile unit level of granularity
- 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

Note: For further information on the binder, refer to the DFSMS website at

```
http://www.ibm.com/storage/software/sms/
```

OpenExtensions Services

OpenExtensions provides capabilities under VM to make it easier to implement or port applications in an open, distributed environment.

OpenExtensions services are available to IBM XL C/C++ for z/VM application programs through the IBM XL C/C++ for z/VM language bindings available with Language Environment.

Together, the OpenExtensions services, Language Environment, and IBM XL C/C++ for z/VM compiler provide an application programming interface that supports industry standards.

Support for POSIX and UNIX-like interfaces includes:

- C programming language support as defined by ISO C.
- C++ programming language support as defined by ISO C++.
- OpenExtensions services, which include C-language programming interfaces defined by the IEEE POSIX 1003.1 standard (with the exception of the fork() function) and subsets of the draft 1003.1a and 1003.1c standards, as well as OpenExtensions-unique extensions
- 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 CAU 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 OpenExtensions decriptions of sockets and X/Open Transport Interface (XTI).
- OpenExtensions extensions that provide z/VM-specific support beyond the defined standards.
- OpenExtensions Shell and Utilities feature, which provides a UNIX-like user interface and an application development environment for creating IBM XL C/C++ for z/VM programs for OpenExtensions, includes the following utilities:

  c89
  Compile and link-edit IBM XL C/C++ for z/VM applications

  make
  Software build and maintenance tool

  ar
  Create and maintain library archives

  iconv
  Converts characters from one code set to another

  localdef
  Creates a compiled locale object.

  c++
  Compiles, assembles, and binds OpenExtensions C++ applications.

- Byte File System (BFS), which provides the POSIX file system
- Support for IBM XL C/C++ for z/VM DLLs.

OpenExtensions also offers:

- Program portability (with support for POSIX.1 and POSIX.1a, a subset of XPG4 and XPG4.2) across multi-vendor operating systems.

  Program portability (with support for POSIX.1, POSIX.1a (draft 6), POSIX.2, POSIX.4a (draft 6), and XPG4.2.) across multi-vendor operating systems.
- A byte file system (BFS) in VM (with support for POSIX.1) including access to data in either VM files or the BFS.
- A UNIX-like user interface (with support for POSIX.2 and XPG4.2) including access to services provided through the OpenExtensions Shell and Utilities feature
- Application threads (with support for a subset of POSIX.1c)
- OpenExtensions extensions that provide VM-specific support beyond the defined standards.

This support is integrated with VM and Language Environment for use by both existing VM applications and for new OpenExtensions applications. OpenExtensions provides capabilities under VM to make it easier to implement or port applications in an open, distributed environment.

Application developers who have worked with UNIX-like environments will find OpenExtensions Shell and Utilities a familiar environment for C and C++ application development. Those familiar with existing VM development environments may find that the OpenExtensions environment can enhance their productivity. For more information on the OpenExtensions Shell and Utilities, see *z/VM: OpenExtensions User's Guide*.

IBM XL C/C++ for z/VM Applications with OpenExtensions Services

To use OpenExtensions services, a IBM XL C/C++ for z/VM program needs to be running on a z/VM system. Furthermore, the program must be an OpenExtensions POSIX program with Language Environment runtime option POSIX(ON), or it must use the interoperability support for OpenExtensions. (For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Interoperability" on page 7.)

A IBM XL C/C++ for z/VM program can use OpenExtensions services by invoking the program in one of the following ways:

- From another program, or from the OpenExtensions shell, using one of the `exec` functions
- Using the POSIX system() call
- From the CMS command line with the POSIX(ON) override option or through the OPENVM RUN command

Functions with dependencies on the OpenExtensions kernel, such as spawn(), or the threading functions, such as pthread_create() are strictly limited to use within the OpenExtensions POSIX environment.

IBM XL C/C++ for z/VM Applications with OpenExtensions Interoperability

OpenExtensions *interoperability* means that:

- OpenExtensions applications running under POSIX(ON) can access traditional VM services and data.
- Traditional VM applications running under POSIX(OFF) can access the OpenExtensions services that permit access to BFS data.

For example, for functions such as fopen() and freopen(), the following statement will open a BFS file named `parts.instock`:

```
fopen("./parts.instock","r")
```

The next statement will open a VM file named PARTS INSTOCK:

```
fopen("//parts.instock","r")
```

Changing the runtime option POSIX(OFF) to POSIX(ON) will affect the environment in which these functions execute. For example, the following statement will open the BFS file named, if POSIX(ON) is in effect, and will open the CMS record file if POSIX(OFF) is in effect:

```
fopen("parts.instock","r")
```

Some of the C and C++ language functions that use OpenExtensions services can be invoked from applications running in the non-POSIX CMS environment, as specified with the runtime option POSIX(OFF).

For example, the following statement will open a BFS file named `parts.instock` whether the application is running under POSIX(OFF) or POSIX(ON):

```
open("parts.instock",O_RDONLY)
```

For more information, see *XL C/C++ for z/VM: User's Guide*.

Here are instructions on how to compile non-OpenExtensions IBM XL C/C++ for z/VM code that references the interoperable POSIX functions.

To use the POSIX functions in IBM XL C/C++ for z/VM, we recommend that you place `#define _OPEN_SYS` before including the required header files that define the POSIX functions you want to use. Once you have included these header files, add a `#undef _OPEN_SYS` so as not to affect any subsequent header files which may define constants or functions that have special behavior under POSIX.

The following example lists header files where this recommendation applies. Note that errno.h is part of the list because it contains errno values that are set by the POSIX functions listed above. These values are not ordinarily visible to a non-POSIX application. You cannot perform your own declaration of errno. Doing so results in undefined behavior.

```
#define _OPEN_SYS
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <dirent.h>
#include <termios.h>
#include <errno.h>
#undef _OPEN_SYS
(other header files)….
```

Input and Output

The IBM XL C/C++ for z/VM runtime library that supports the IBM XL C/C++ for z/VM compiler supports different input and output (I/O) interfaces, file types, and access methods. Additional support is provided by the C++ Standard I/O Stream Library.

I/O Interfaces

The z/VM 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. All I/O is processed by character.

Record I/O
Your input and output can also be processed by record. A record is a set of data treated as a unit. Record processing of VSAM data sets is also supported. Record I/O is a C extension to the ANSI standard.

TCP/IP Sockets I/O

OpenExtensions provides support for an enhanced version of an industry-accepted protocol for client/server communication that is known as sockets. A set of C and C++ language functions provides support for OpenExtensions sockets. OpenExtensions sockets correspond closely to the sockets 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 OpenExtensions socket application program interface (API) provides support for both OpenExtensions domain sockets and Internet domain sockets. OpenExtensions domain sockets, or local sockets, allow interprocess communication within z/VM, independent of TCP/IP. Local sockets behave like traditional OpenExtensions sockets and allow processes to communicate with one another on a single system. With Internet sockets, application programs can communicate with each other in the network using TCP/IP.

In addition, the C++ I/O Stream libraries support 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 on CMS minidisks and Shared File System (SFS) directories, the IBM XL C/C++ for z/VM runtime library supports the following file types:

Virtual Storage Access Method (VSAM) Data Sets

IBM XL C/C++ for z/VM has native support for three types of VSAM data organization:

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

All three types are ordered and can have keys associated with their records.

Byte File System

When you are running in a CMS virtual machine, IBM XL C/C++ for z/VM recognizes a Byte File System (BFS) file as such if the name specified on the `fopen()` or `freopen()` call conforms to certain rules (see *XL C/C++ for z/VM: User's Guide*). You can create regular BFS files, special character BFS files, or FIFO BFS 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. Because memory files reside in main storage and only exist while the program is executing, they are primarily used as work files. You can access memory files across load modules through calls to non-POSIX `system()` and C or 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 Space

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

Additional I/O Features

IBM XL C/C++ for z/VM provides additional I/O support through the following features:

- User error handling for serious I/O failures (SIGIOERR)

- Multibyte character I/O functions
- Fixed-point (packed) decimal data type support in formatted I/O functions

Additional Features of IBM XL C/C++ for z/VM

Table 1. IBM XL C/C++ for z/VM Features

Feature	Explanation
long long Data Type	The IBM XL C/C++ for z/VM compiler supports long long as a native data type when the compiler option `LANGLVL(LONGLONG)` is turned on. This option is turned on by default by the compiler option `LANGLVL(EXTENDED)`.
Multibyte Character Support	IBM XL C/C++ for z/VM 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 z/VM 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 `wcstombs()`, `mbstowcs()`, `wcsrtombs()`, and `mbsrtowcs()`, as well as the family of wide-character I/O functions. Wide-character data can be represented by the `wchar_t` data type.
Extended Precision Floating-Point Numbers	IBM XL C/C++ for z/VM provides three S/390 floating-point number data types: single precision (32 bits), declared as `float`; double precision (64 bits), declared as `double`; and extended precision (128 bits), declared as `long double` . Extended precision floating-point numbers give greater accuracy to mathematical calculations. IBM XL C/C++ for z/VM also supports IEEE 754 floating-point representation. By default, `float` , `double` , and `long double` values are represented in IBM S/390 floating point format. However, the IEEE 754 floating-point representation is used if you specify the `FLOAT(IEEE754)` compile option. For details on this support, see the description of the FLOAT option in *z/OS: XL C/C++ Programming Guide*.
Command Line Redirection	You can redirect the standard streams `stdin` , `stderr`, and `stdout` from the command line or when calling programs using the `system()` function.
National Language Support	IBM XL C/C++ for z/VM provides message text in either American English or Japanese. You can dynamically switch between these two languages.
Locale Definition Support	IBM XL C/C++ for z/VM 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 `isdigit()`, use this information.
Coded Character Set (Code Page) Support	The IBM XL C/C++ for z/VM compiler can compiler C and C++ source written in different EBCDIC code pages. In addition, the `iconv` utility converts data or source from one code page to another.
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.

Table 1. IBM XL C/C++ for z/VM Features (continued)	
Feature	**Explanation**
Multithreading	Threads are efficient in applications that allow them to take advantage of any underlying parallelism available in the host environment. This underlying parallelism in the host can be exploited either by forking a process and creating a new address space, or by using multiple threads within a single process. For more information, see *z/OS: XL C/C++ Programming Guide*.
Packed Structures and Unions	z/VM C provides support for packed structures and unions. Structures and unions may be packed to reduce the storage requirements of a z/VM C program or to define structures that are laid out according to COBOL or PL/I structure layout rules.
Fixed-point (Packed) Decimal Data	z/VM 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.
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 `system()` function under CMS and OpenExtensions. You can also use the `system()` function to call execs in CMS or Shell scripts using OpenExtensions.
Exploitation of hardware	Use the `ARCHITECTURE` compiler option to select the minimum level of machine architecture on which your program will run. `ARCH(2)` instructs the compiler to generate faster instruction sequences that are available only on newer machines. `ARCH(3)` also generates these faster instruction sequences and enables support for IEEE 754 Binary Floating-Point instructions. Code compiled with `ARCH(2)` runs on G2, G3, G4, and 2003 processors and code compiled with `ARCH(3)` runs on a G5 or G6 processor, and follow-on models. Use the `TUNE` compiler option to optimize your application for a specific machine architecture. `TUNE` impacts performance only; it does not impact the processor model on which you will be able to run your application. `TUNE(3)` optimizes your application for the newer G4, G5, and G6 processors. `TUNE(2)` optimizes your application for other architectures. For more information, see the `ARCHITECTURE` and `TUNE` compiler information in *XL C/C++ for z/VM: User's Guide*.
Built-in Functions for Floating-Point and Other Hardware Instructions	Use built-in functions for floating-point and other hardware instructions that are otherwise inaccessible to C and C++ programs. For more information, see *z/OS: XL C/C++ Programming Guide*.

Chapter 2. Header Files

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

The header files provided with the IBM XL C/C++ for z/VM runtime library contain macro and constant definitions, type definitions, and function declarations. Some functions require definitions and declarations from header files to work correctly. As long as the necessary statements from the files are coded directly into the source, including header files is optional.

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 the *z/OS: XL C/C++ Language Reference (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/cbclx01_v2r5.pdf)* and the *XL C/C++ for z/VM: 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: The LANGLVL compiler option does not define or undefine these macros.

Table Table 2 on page 13 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 2. 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_DEFAULT	Yes	Yes	Yes			
_OPEN_SYS	Yes	Yes	Yes	Yes		
_OPEN_SYS_IPC _EXTENSIONS	Yes	Yes			Yes	
_OPEN_SYS_PTY _EXTENSIONS	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
_OPEN_SOURCE 3	Yes	Yes	Yes	Yes	Yes	Yes
_OE_SOCKETS	Yes	Yes				

© Copyright IBM Corp. 2002, 2024

Table 2. Feature Test Macros and Standards (continued)

Feature Test Macro	POSIX .1	POSIX .1a	POSIX .2	POSIX .4a	XPG4 .2	XPG4 .2 Ext
_OPEN_SYS_SOCK_EXT	Yes	Yes		Yes	Yes	
_ALL_SOURCE_NO_THREADS	Yes	Yes	Yes		Yes	Yes

The following feature test macros are supported:

- _POSIX_SOURCE

  When defined to any value with `#define`, it makes symbols required by POSIX.1 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 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 makes symbols required by POSIX.1 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 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, it makes symbols required by POSIX.1, POSIX.1a, POSIX.2 visible. Any symbols defined by the _OPEN_THREADS macro are also 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 OpenExtensions services extension.
- _OPEN_THREADS

  When defined to 1, it makes symbols required by POSIX.1, POSIX.1a, and POSIX.4a visible.
- _XOPEN_SOURCE

  Defines the functionality defined in the XPG4 standard dated July 1992.
- _XOPEN_SOURCE_EXTENDED

  When defined to 1, this indicates the functionality in the XPG4 standard plus the set of “Common APIs for UNIX-based Operating Systems”, April, 1994, draft.
- _OPEN_SYS_IPC_EXTENSIONS

  Defines OpenExtensions 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 OpenExtensions 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 OpenExtensions services in VM/ESA version 2 release 1.0. This macro is equivalent to specifying _OPEN_SYS.

  When defined to 2, this defines all of the functionality that is available on OpenExtensions services in VM/ESA version 2 release 3.0, including a subset of XPG4, XPG4.2, and most of the OpenExtensions extensions, with the exception of the parity extensions.

- _ALL_SOURCE

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

  When defined to `0`, and if no other feature test macro is defined, 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 (for example, those visible when the LNGLVL(EXTENDED) compiler options specified), may be exposed.

  When defined to 1, this provides the base level of OpenExtensions 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_SOCK_EXT

  Defines the interface for function prototypes and structures for the extended sockets and bulk mode support.
- _OPEN_SYS_SOCK_IPV6

  Allows the use of IPv6-related functions and data structures.
- _SHARE_EXT_VARS

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

  Note: This feature test macro is not necessary when an application is compiled with the XPLINK option. In this case, the POSIX and XPG4 external variables will be resolved through the C RTL side deck CELHS003 in the SCEELIB data set and will be accessible from all dynamically loaded modules. See *z/OS: Language Environment Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf)* for more details.
- _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 stdio.h in the program source.

Note: This feature test macro is not necessary when an application is compiled with the XPLINK option. In this case, the POSIX and XPG4 external variables will be resolved through the C RTL side deck CELHS003 in the SCEELIB data set and will be accessible from all dynamically loaded modules. See *z/OS: Language Environment Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf)* for more details.

- _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*.

  Note: This feature test macro is not necessary when an application is compiled with the XPLINK option. In this case, the POSIX and XPG4 external variables will be resolved through the C RTL side deck CELHS003 in the SCEELIB data set and will be accessible from all dynamically loaded modules. See *z/OS: Language Environment Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf)* for more details.

- _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.

  Note: This feature test macro is not necessary when an application is compiled with the XPLINK option. In this case, the POSIX and XPG4 external variables will be resolved through the C RTL side deck CELHS003 in the SCEELIB data set and will be accessible from all dynamically loaded modules. See *z/OS: Language Environment Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf)* for more details.

- _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.

  Note: This feature test macro is not necessary when an application is compiled with the XPLINK option. In this case, the POSIX and XPG4 external variables will be resolved through the C RTL side deck CELHS003 in the SCEELIB data set and will be accessible from all dynamically loaded modules. See *z/OS: Language Environment Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf)* for more details.

- _SHR__LOC1

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

  Note: This feature test macro is not necessary when an application is compiled with the XPLINK option. In this case, the POSIX and XPG4 external variables will be resolved through the C RTL side deck CELHS003 in the SCEELIB data set and will be accessible from all dynamically loaded modules. See *z/OS: Language Environment Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf)* for more details.

- _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.

  Note: This feature test macro is not necessary when an application is compiled with the XPLINK option. In this case, the POSIX and XPG4 external variables will be resolved through the C RTL side deck CELHS003 in the SCEELIB data set and will be accessible from all dynamically loaded modules. See *z/OS: Language Environment Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf)* for more details.

- _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.

  Note: This feature test macro is not necessary when an application is compiled with the XPLINK option. In this case, the POSIX and XPG4 external variables will be resolved through the C RTL side deck CELHS003 in the SCEELIB data set and will be accessible from all dynamically loaded modules. See *z/OS: Language Environment Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf)* for more details.

- _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.

  Note: This feature test macro is not necessary when an application is compiled with the XPLINK option. In this case, the POSIX and XPG4 external variables will be resolved through the C RTL side deck CELHS003 in the SCEELIB data set and will be accessible from all dynamically loaded modules. See *z/OS: Language Environment Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf)* for more details.

- _SHR_LOC1

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

  Note: This feature test macro is not necessary when an application is compiled with the XPLINK option. In this case, the POSIX and XPG4 external variables will be resolved through the C RTL side deck CELHS003 in the SCEELIB data set and will be accessible from all dynamically loaded modules. See *z/OS: Language Environment Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf)* for more details.

- _SHR_LOC2

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

  Note: This feature test macro is not necessary when an application is compiled with the XPLINK option. In this case, the POSIX and XPG4 external variables will be resolved through the C RTL side deck CELHS003 in the SCEELIB data set and will be accessible from all dynamically loaded modules. See *z/OS: Language Environment Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf)* for more details.

- _SHR_LOCS

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

  Note: This feature test macro is not necessary when an application is compiled with the XPLINK option. In this case, the POSIX and XPG4 external variables will be resolved through the C RTL side deck CELHS003 in the SCEELIB data set and will be accessible from all dynamically loaded modules. See *z/OS: Language Environment Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf)* for more details.

- _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.

  Note: This feature test macro is not necessary when an application is compiled with the XPLINK option. In this case, the POSIX and XPG4 external variables will be resolved through the C RTL side deck CELHS003 in the SCEELIB data set and will be accessible from all dynamically loaded modules. See *z/OS: Language Environment Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf)* for more details.

- _SHR_OPTIND

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

Note: This feature test macro is not necessary when an application is compiled with the XPLINK option. In this case, the POSIX and XPG4 external variables will be resolved through the C RTL side deck CELHS003 in the SCEELIB data set and will be accessible from all dynamically loaded modules. See *z/OS: Language Environment Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf)* for more details.

- _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.

  Note: This feature test macro is not necessary when an application is compiled with the XPLINK option. In this case, the POSIX and XPG4 external variables will be resolved through the C RTL side deck CELHS003 in the SCEELIB data set and will be accessible from all dynamically loaded modules. See *z/OS: Language Environment Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf)* for more details.

- _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.

  Note: This feature test macro is not necessary when an application is compiled with the XPLINK option. In this case, the POSIX and XPG4 external variables will be resolved through the C RTL side deck CELHS003 in the SCEELIB data set and will be accessible from all dynamically loaded modules. See *z/OS: Language Environment Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf)* for more details.

Header File Descriptions

This section contains an alphabetical list and a description of the IBM XL C/C++ for z/VM header files.

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.

arpa/nameser.h

The arpa/nameser.h header file contains the definitions used to support the construction of queries and the inspection of answers received from a Domain Name Server available in a network. It also contains the macros GETSHORT(), PUTSHORT(), GETLONG(), and PUTLONG() that are used to construct or inspect DNS requests.

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().

bpxhmnt.h

The bpxhmnt.h header file declares the BPXHMNT structure, which sets mount options for Network File System (NFS) mounts.

cassert

The cassert header file contains definitions for C++ for enforcing assertions when functions execute. Include the standard header into a C++ program to effectively include the standard header <assert.h> within the std namespace.

```
namespace std {
#include <assert.h>
    };
```

_Ccsid.h

The _Ccsid.h header file declares functions, symbols and data types used in CCSID (Coded Character Set IDentifier) to codeset name conversion.

cctype

The cctype header file contains definitions for C++ for classifying characters. Include the standard header into a C++ program to effectively include the standard header <ctype.h> within the std namespace.

```
namespace std {
#include <ctype.h>
    };
```

ceeedcct.h

The ceeedcct.h header file contains C declarations of the Language Environment condition tokens.

cerrno

The cerrno header file contains definitions for C++ for testing error codes reported by library functions. Include the standard header into a C++ program to effectively include the standard header <errno.h> within the std namespace.

```
namespace std {
#include <errno.h>
    };
```

cfloat

The cfloat header file contains definitions for C++ for testing floating-point type properties. Include the standard header into a C++ program to effectively include the standard header <float.h> within the std namespace.

```
namespace std {
#include <float.h>
    };
```

cics.h

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

ciso646

The ciso646 header file contains definitions for C++ for programming in ISO646 variant character sets. Include the standard header into a C++ program to effectively include the standard header <iso646.h> within the std namespace.

```
namespace std {
#include <iso646.h>
    };
```

climits

The climits header file contains definitions for C++ for testing integer type properties. Include the standard header into a C++ program to effectively include the standard header <limits.h> within the std namespace.

```
namespace std {
#include <limits.h>
    };
```

clocale

The clocale header file contains definitions for C++ for adapting to different cultural conventions. Include the standard header into a C++ program to effectively include the standard header <locale.h> within the std namespace.

```
namespace std {
#include <locale.h>
    };
```

cmath

The cmath header file contains definitions for C++ for computing common mathematical functions. Include the standard header into a C++ program to effectively include the standard header <math.h> within the std namespace.

```
namespace std {
#include <math.h>
    };
```

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()	getwmcc0ll()	ismccollel()	maxcoll()	strtocoll()

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 "Internationalization: Locales and Character Sets" topic in *z/OS: XL C/C++ Programming Guide*.

cpio.h

The cpio.h header file contains CPIO archive values.

csetjmp

The csetjmp header file contains definitions for C++ for executing nonlocal goto statements. Include the standard header into a C++ program to effectively include the standard header <setjmp.h> within the std namespace.

```
namespace std {
#include <setjmp.h>
    };
```

csignal

The csignal header file contains definitions for C++ for controlling various exceptional conditions. Include the standard header into a C++ program to effectively include the standard header <signal.h> within the std namespace.

```
namespace std {
#include <signal.h>
    };
```

csp.h

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

These macros are *not* supported under OpenExtensions, and they are not supported for C++ applications.

cstdarg

The cstdarg header file contains definitions for C++ for accessing a varying number of arguments. Include the standard header into a C++ program to effectively include the standard header <stdarg.h> within the std namespace.

```
namespace std {
#include <stdarg.h>
    };
```

cstddef

The cstddef header file contains definitions for C++ for defining several useful types and macros. Include the standard header into a C++ program to effectively include the standard header <stddef.h> within the std namespace.

```
namespace std {
#include <stddef.h>
    };
```

cstdio

The cstdio header file contains definitions for C++ for performing input and output. Include the standard header into a C++ program to effectively include the standard header <stdio.h> within the std namespace.

```
namespace std {
#include <stdio.h>
    };
```

cstdlib

The cstdlib header file contains definitions for C++ for performing a variety of operations. Include the standard header into a C++ program to effectively include the standard header <stdlib.h> within the std namespace.

```
namespace std {
#include <stdlib.h>
    };
```

cstring

The cstring header file contains definitions for C++ for manipulating several kinds of strings. Include the standard header into a C++ program to effectively include the standard header <string.h> within the std namespace.

```
namespace std {
#include <string.h>
    };
```

ctest.h

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

cdump()	csnap()	ctest()	ctrace()

ctime

The ctime header file contains definitions for C++ for converting between various time and date formats. Include the standard header into a C++ program to effectively include the standard header <time.h> within the std namespace.

```
namespace std {
#include <time.h>
    };
```

ctype.h

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

isalnum()	isalpha()	isblank()	iscntrl()	isdigit()
isgraph()	islower()	isprint()	ispunct()	isspace()
isupper()	isxdigit()	tolower()	toupper()	

_XOPEN_SOURCE

isascii()	toascii()	tolower()	toupper()

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 "Internationalization: Locales and Character Sets" topic in *z/OS: XL C/C++ Programming Guide*.

cwchar

The cwchar header file contains definitions for C++ for manipulating wide streams and several kinds of strings. Include the standard header into a C++ program to effectively include the standard header <wchar.h> within the std namespace.

```
namespace std {
#include <wchar.h>
    };
```

cwctype

The cwctype header file contains definitions for C++ for classifying wide characters. Include the standard header into a C++ program to effectively include the standard header <wctype.h> within the std namespace.

```
namespace std {
#include <wctype.h>
    };
```

decimal.h

The decimal.h header file is not suppored under z/VM C++ applications.

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

decabs() decchk() decfix()

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.

closedir() opendir() readdir() rewinddir()

_XOPEN_SOURCE

seekdir() telldir()

This header file can be used by C++ POSIX(OFF) functions. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

dll.h

The dll.h header file declares the following functions:

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

You should 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 declares 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 symbolic constants that are returned in the external variable errno.

The errno.h header file also defines errno, which is a modifiable `lvalue` 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`.

You can use the strerror() or perror() functions 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.

exception

The exception header file defines several types and functions related to the handling of exceptions.

```
namespace std {
    class exception;
    class bad_exception;
    typedef void (*terminate_handler)();
    typedef void (*unexpected_handler)();
    terminate_handler
        set_terminate(terminate_handler ph) throw();
    unexpected_handler
        set_unexpected(unexpected_handler ph) throw();
    void terminate();
    void unexpected();
    bool uncaught_exception();
    };
```

bad_exception

```
class bad_exception : public exception {
    };
```

The class describes an exception that can be thrown from an unexpected handler. The value returned by what() is an implementation-defined C string. None of the member functions throw any exceptions.

exception

```
class exception {
public:
    exception() throw();
    exception(const exception& rhs) throw();
    exception& operator=(const exception& rhs) throw();
    virtual ~exception() throw();
    virtual const char *what() const throw();
    };
```

The class serves as the base class for all exceptions thrown by certain expressions and by the standard C++ library. The C string value returned by what() is left unspecified by the default constructor, but may be defined by the constructors for certain derived classes as an implementation-defined C string. None of the member functions throw any exceptions.

terminate_handler

```
typedef void (*terminate_handler)();
```

The type describes a pointer to a function suitable for use as a terminate handler.

unexpected_handler

```
typedef void (*unexpected_handler)();
```

The type describes a pointer to a function suitable for use as an unexpected handler.

fcntl.h

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

_POSIX_SOURCE

creat() fcntl() open()

For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

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 13.

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 3. Definitions in float.h

Definition	Meaning
FLT_ROUNDS = 0	The rounding mode for floating-point addition
FLT_RADIX = 16	The radix for IBM XL C/C++ for z/VM applications, FLT_RADIX, is defined to be 16
FLT_MANT_DIG = 6 DBL_MANT_DIG = 14 LDBL_MANT_DIG = 28	The number of hexadecimal digits stored to represent the significant of a fraction
FLT_DIG = 6 DBL_DIG = 15L DBL_DIG = 32	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 = -64 DBL_MIN_EXP = -64 LDBL_MIN_EXP = -64	The minimum negative integer such that FLT_RADIX raised to that power minus 1 is a normalized floating-point number
FLT_MIN_10_EXP = -78 DBL_MIN_10_EXP = -78 LDBL_MIN_10_EXP = -78	The minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers
FLT_MAX_EXP = 63 DBL_MAX_EXP = 63 LDBL_MAX_EXP = 63	The maximum integer such that FLT_RADIX raised to that power minus 1 is a representable finite floating-point number

Table 3. Definitions in float.h (continued)

Definition	Meaning
FLT_MAX_10_EXP = 75 DBL_MAX_10_EXP = 75 LDBL_MAX_10_EXP = 75	The maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers
FLT_MAX = 7.2370051E75 DBL_MAX = 7.2370055E75 LDBL_MAX = 7.2370055E75	The maximum representable finite floating-point number
FLT_EPSILON = 9.536743e-07 DBL_EPSILON = 2.220446e-16 LDBL_EPSILON = 3.081488e-33	The difference between 1.0 and the least value greater than 1.0 that is representable in the given floating-point type
FLT_MIN = 5.397605e-79 DBL_MIN = 5.397605e-79 LDBL_MIN = 5.397605e-79	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 file name matching types.

fpxcp.h

The fpxcp.h header file declares floating-point exception interfaces.

__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 is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

This header file declares functions 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.

The header file also 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.

_Ieee754.h

The _Ieee754.h header file declares IEEE 754 interfaces.

ims.h

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

inttypes.h

The inttypes.h header file contains typedefs for integer data types.

iso646.h

The header file iso646.h allows the user to use the following macros in place of the associated operator.

Macro
Operator

and
&&

and_eq
&=

bitand
&

bitor
|

compl
~

not
!

not_eq
!=

or
||

or_eq
|=

xor
^

xor_eq
^=

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 langinfo.h header file also defines the type `nl_item`, which is the type of the constants defined.

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 *z/OS: XL C/C++ Programming Guide*.

lc_core.h

The lc_core.h header file contains locale-related data structures.

lc_sys.h

The lc_sys.h header file contains definitions used by the localedef utility for building references to locale methods in ASCII locales.

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 4. Definitions of Resource Limits

Name	Value
ATEXIT_MAX	2048
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)

Table 4. Definitions of Resource Limits (continued)

Name	Value
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_SYMLOOP	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

Table 4. Definitions of Resource Limits (continued)

Name	Value
UCHAR_MAX	255
UINT_MAX	4294967295
ULONG_MAX	4294967295U
ULONGLONG_MAX	(18446744073709551615ULL)
USHRT_MAX	65535

localdef.h

The localdef.h header file defines data structures for locale objects which are loaded by setlocale(). The data structures in localdef.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 Table 5 on page 30 below shows the elements of the `lconv` structure and the defaults for the C locale.

Table 5. 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_thousands_sep`	Separator for digits in formatted monetary quantities.	""

Table 5. Elements of the lconv Structure (continued)

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 field is equivalent to calling nl_langinfo() with these keywords:

Keyword	Element
abmon	abbrev_month_names
mon	month_names
abday	abbrev_day_names
day	day_names
d_t_fmt	date_time_format
d_fmt	date_format
t_fmt	time_format
am_pm	am_string
am_pm	pm_string
t_fmt_ampm	time_format_ampm

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

LC_ALL	LC_COLLATE	LC_CTYPE	LC_MONETARY
LC_NUMERIC	LC_TIME	LC_TOD	NULL

You can customize the aspects of a program related to national language or cultural characteristics (such as time zone, currency symbols, and sorting order of characters) at run time by using different locales to suit users' requirements at those locales. The methods for doing so are discussed in the internationalization chapters of *z/OS: XL 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.

absf()	absl()	acos()	acosf()	acosl()	asin()
asinf()	asinl()	atan()	atanf()	atanl()	atan2()
atan2f()	atan2l()	ceil()	ceilf()	ceill()	cos()
cosf()	cosl()	cosh()	coshf()	coshl()	exp()
expf()	expl()	fabsf()	fabsl()	floor()	floorf()
floorl()	fmod()	fmodf()	fmodl()	frexp()	frexpf()
frexpl()	ldexp()	ldexpf()	ldexpl()	log()	logf()
logl()	log10()	log10f()	log10l()	modf()	modff()
modfl()	pow()	powf()	powl()	sin()	sinf()
sinl()	sinh()	sinhf()	sinhl()	sqrt()	sqrtf()

sqrtl()	tan()	tanf()	tanl()	tanh()	tanhf()
tanhl()					

Special Behavior for C++

For C++ applications, each of the base functions in the preceding list is also overloaded for float, double, and long double. For example:

- float sqrt(float)
- double sqrt(double)
- long double sqrt(long double)

_XOPEN_SOURCE

erf()	erfc()	gamma()	hypot()	isnan()
jn()	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 63.

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 a value of `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 "Internationalization: Locales and Character Sets" topic in *z/OS: XL 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.

The message catalog structures defined in this file are not compatible with the OpenExtensions Shell and Utilities message catalog.

mtf.h

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

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

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

This function is not supported in z/VM; however, the header file is supplied for compatability with z/OS.

_Nascii.h

The _Nascii.h header file contains the externals for the correspondence table and functions that support bimodal application development.

ndbm.h

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

net/if.h

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

net/rtrouteh.h

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

netdb.h

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

netinet/in.h

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

The following macros are supported for IPv6:

IN6_IS_ADDR_LINKLOCAL
IN6_IS_ADDR_LOOPBACK
IN6_IS_ADDR_MC_GLOBAL
IN6_IS_ADDR_MC_LINKLOCAL
IN6_IS_ADDR_MC_NODELOCAL
IN6_IS_ADDR_MC_ORGLOCAL
IN6_IS_ADDR_MC_SITELOCAL
IN6_IS_ADDR_MULTICAST
IN6_IS_ADDR_SITELOCAL
IN6_IS_ADDR_UNSPECIFIED
IN6_IS_ADDR_V4COMPAT
IN6_IS_ADDR_V4MAPPED

new

The <new> header file defines several types and functions that control allocation and freeing of storage under program control.

Some of the functions declared in this header are replaceable. The implementation supplies a default version. A program can, however, define a function with the same signature to replace the default version at link time. The replacement version must satisfy the requirements of the function.

```
namespace std {
    typedef void (*new_handler)();
    class bad_alloc;
    class nothrow_t;
    extern const nothrow_t nothrow;
//      FUNCTIONS
    new_handler set_new_handler(new_handler ph) throw();
};
//      OPERATORS
void operator delete(void *p) throw();
void operator delete(void *, void *) throw();
void operator delete(void *p, const std::nothrow_t&) throw();
void operator delete[](void *p) throw();
void operator delete[](void *, void *) throw();
void operator delete[](void *p, const std::nothrow_t&) throw();
void *operator new(std::size_t n) throw(std::bad_alloc);
void *operator new(std::size_t n, const std::nothrow_t&) throw();
void *operator new(std::size_t n, void *p) throw();
void *operator new[](std::size_t n) throw(std::bad_alloc);
void *operator new[](std::size_t n, const std::nothrow_t&) throw();
void *operator new[](std::size_t n, void *p) throw();
```

The <new> header file supercedes the new.h header, which remains for compatibility as a wrapper to <new>.

bad_alloc

```
class bad_alloc : public exception {
    };
```

The class describes an exception thrown to indicate that an allocation request did not succeed. The value returned by what() is an implementation-defined C string. None of the member functions throw any exceptions.

new_handler

```
typedef void (*new_handler)();
```

The type points to a function suitable for use as a new handler.

nothrow

```
extern const nothrow_t nothrow;
```

The object is used as a function argument to match the parameter type nothrow_t.

nothrow_t

```
class nothrow_t {};
```

The class is used as a function parameter to operator new to indicate that the function should return a null pointer to report an allocation failure, rather than throw an exception.

new.h

The ISO/ANSI C++ Standard (ISO/IEC 14882:1998(E)) supersedes this header with the new header <new>. However, new.h remains for compatibility as a wrapper for TARGET releases of z/OS V1.2 and later.

For compilations with a TARGET release prior to z/OS V1.2, the new.h header file declares the set_new_handler() function, which is used for z/OS 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 28.

nl_types.h

The nl_types.h header file contains the definition for the `nl_item` typedef for the nl_langinfo() function.

No feature test macro required.

_XOPEN_SOURCE

catclose()	catgets()	catopen()

Note: These functions cannot be used to access messages in the OpenExtensions Shell and Utilities message catalog which uses its own internal versions of these commands.

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 "Internationalization: Locales and Character Sets" topic in *z/OS: XL C/C++ Programming Guide*.

poll.h

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

pthread.h

The pthread.h header file is supported only in applications with POSIX(ON) specified.

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

pthread_attr_destroy()	pthread_attr_getdetachstate()
pthread_attr_getstacksize()	pthread_attr_getweight_np()
pthread_attr_init()	pthread_attr_setdetachstate()
pthread_attr_setstacksize()	pthread_attr_setweight_np()
pthread_cancel()	pthread_cleanup_pop()
pthread_cleanup_push()	pthread_condattr_destroy()
pthread_condattr_getkind_np()	pthread_condattr_setkind_np()
pthread_condattr_init()	pthread_cond_broadcast()
pthread_cond_destroy()	pthread_cond_init()
pthread_cond_signal()	pthread_cond_timedwait()
pthread_cond_wait()	pthread_create()
pthread_detach()	pthread_equal()
pthread_exit()	pthread_getspecific()
pthread_join()	pthread_join_d4_np()
pthread_key_create()	pthread_kill()
pthread_mutexattr_destroy()	pthread_mutexattr_init()
pthread_mutexattr_getkind_np()	pthread_mutexattr_setkind_np()
pthread_mutex_destroy()	pthread_mutex_init()

pthread_mutex_lock()
pthread_mutex_trylock()
pthread_mutex_unlock()
pthread_once
pthread_self()
pthread_setintr()
pthread_setintrtype()
pthread_setspecific()
pthread_testintr()
pthread_yield()

pthread.h also defines these symbols:

_COND_DEFAULT
_COND_NODEBUG
_DETACHED
_HEAVY_WEIGHT
_MEDIUM_WEIGHT
_MUTEX_NODEBUG
_MUTEX_NONRECURSIVE
_MUTEX_RECURSIVE
_UNDETACHED
NO_PRIO_INHERIT
PRIO_INHERIT
PTHREAD_DEFAULT_SCHED
PTHREAD_INHERIT_SCHED
PTHREAD_INTR_ASYNCHRONOUS
PTHREAD_INTR_CONTROLLED
PTHREAD_INTR_DISABLE
PTHREAD_INTR_ENABLE
PTHREAD_ONCE_INIT
PTHREAD_SCOPE_GLOBAL
PTHREAD_SCOPE_LOCAL
SCHED_FIFO
SCHED_OTHER
SCHED_RR
PRIO_PROTECT

pwd.h

The pwd.h header file is supported only for applications with POSIX(ON) specified.

This header file declares the functions to access the user database through the `passwd` 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 declares the type `size_t`. It contains the definition 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.

resolv.h

The resolv.h header file contains the __res_state structure and the definitions to support the IP Address Resolution functions commonly called the Resolver. It contains the prototypes for the following functions — dn_comp(), dn_expand(), dn_find(), dn_skipname(), res_init(), res_mkquery(), res_query(), res_querydomain(), res_search(), and res_send() — which are used to communicate with a Domain Name Server (DNS).

rexec.h

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

saiucv.h

The saiucv.h header file contains the `sockaddr` structure definition for AF_IUCV sockets.

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().

For more information about using POSIX support, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

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

- 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 declared in signal.h is supported only in a POSIX program. You must specify the POSIX(ON) runtime option for these functions. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

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

The following values are available in OpenExtensions only:

- Signals:

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

- *options* arguments for sigprocmask(): SIG_BLOCK, SIG_UNBLOCK, and SIG_SETMASK.
- Flags for `sa_flags` field: SA_NOCLDSTOP and SA_OLD_STYLE.
- The structures `sigaction`, `sigset_t`, and `pid_t`.

spawn.h

The spawn.h header file contains spawn() constants and inheritance structure and defines the functions spawn() and spawnp().

spc.h

The spc.h header file is supported only for C applications.

The spc.h header file contains declarations for the functions available in the system programming environment. See the system programming C (SPC) facility topics in *z/OS: XL 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 Chapter 3, "Library Functions," on page 61:

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 long type of the result of subtracting two pointers

`size_t`
`typedef` for the type of the value returned by *sizeof*.

`wchar_t`
`typedef` for a wide-character constant.

The stddef.h header file 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 "Internationalization: Locales and Character Sets" topic in *z/OS: XL C/C++ Programming Guide*.

stdefs.h

The stdefs.h header file contains the same information as found in <stddef.h>.

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. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

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, 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 minimum value for the total number of files that you can open simultaneously.

FILENAME_MAX
The integral constant expressions that can be used in the size specification of an array that hold the longest possible name of a file that can be opened.

L_tmpnam
The size of the longest temporary name that can be generated by the tmpnam() function.

L_ctermid
Maximum size of a character array for ctermid() output. This macro is supported only in a POSIX program. See "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7 for more information.

NULL
A pointer which never points to a data object.

TMP_MAX
The minimum number of unique file names that can be generated by the tmpnam() 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.

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

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

__KEY_EQ	__KEY_EQ_BWD	__KEY_FIRST	__KEY_GE
__KEY_LAST	__RBA_EQ	__RBA_EQ_BWD	

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

__CURRENT	__CURRENT_LOWER	__LOWER

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 *z/OS: XL 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
__CMS_CLOSE	__CMS_ERASE	__CMS_EXTRACT
__CMS_LINERD	__CMS_LINEWRT	__CMS_OPEN
__CMS_QUERY	__CMS_READ	__CMS_RENAME
__CMS_STATE	__CMS_WRITE	__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()[1,3]	alloca()[1]	atexit()	atof()
atoi()	atol()	bsearch()	calloc()	cds()[1]
clearenv()	cs()[1]	csid()	div()[3]	exit()

fetch()[2]	fetchep()[2]	free()	getenv()	labs()
ldiv()	llabs()	lldiv()	_librel()	malloc()
mblen()	mbstowcs()	mbtowc()	qsort()	rand()
realloc()	release()[2]	rpmatch()	setenv()	srand()
strtod()	strtol()	strtoll()	strtoul()	strtoull()
system()	wcsid()	wcstombs()	wctomb()	

[1] Built-in function.

[2] Not supported under C++ applications.

[3] Special Behavior for C++ For C++ applications, the functions abs() and div() are also overloaded for the type long.

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 40.

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 atexit() function.

EXIT_FAILURE
Used by the atexit() function.

RAND_MAX
Expands to an integer representing the largest number that the rand() function can return.

MB_CUR_MAX
Expands to an integer representing the maximum number of bytes in a multibyte character. This value is dependent on the current locale.

If MB_CUR_MAX is set to 1, multibyte functions will behave as if all multibyte characters are one byte long; wide-character functions are *not* supported and full DCBS support is *not* provided. If MB_CUR_MAX is 4, all DBCS support provided by the library is enabled.

string.h

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

No feature test macro required.

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

[1] Built-in function.

_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 40.

strings.h

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

stropts.h

The stropts.h header file declares the following functions:

fattach()	fdetach()	getmsg()	getpmsg()
ioctl()	isastream()	putmsg()	putpmsg()

syslog.h

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

sys/acl.h

The sys/acl.h header enables users to manipulate ACLs. It also declares the following functions:

acl_create_entry()	acl_delete_entry()	acl_delete_fd()	acl_delete_file()
acl_first_entry()	acl_free()	acl_from_text()	acl_get_entry()
acl_get_fd()	acl_get_file()	acl_init()	acl_set_fd()
acl_set_file()	acl_sort()	acl_to_text()	acl_update_entry()
acl_valid()			

sys/__cpl.h

The sys/__cpl.h header contains definition for the __cpl() function. It also defines the following constants:

Table 6. Symbolic Constants defined in sys/__cpl.h

Symbolic Constant	Description
CPL_QUERY	Request data from available Coupling Facilities
CPL_CFSIZER	Request a structure size

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/layout.h

The sys/layout.h header file contains declarations for supporting bidirectional (Bidi) conversion of data between Visual (z/OS) and Implicit (USS) formats. Initial support is for Arabic and Hebrew data.

sys/mman.h

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

sys/__messag.h

The sys/__messag.h header file contains definitions for the __console() function.

sys/mntent.h

The sys/mntent.h header file is supported only in a POSIX program by specifying the _OPEN_SYS feature test macro. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

The sys/mntent.h header file declares the w_getmntent() function and 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 C support, these definitions are included in sys/stat.h to make sys/stat.h conform with POSIX.

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 WorkLoad 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 OpenExtensions files and their access:

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

For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

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.

_POSIX_SOURCE

dev_t	Device numbers
gid_t	Group IDs
ino_t	File serial numbers
mode_t	Some file attributes

nlink_t	Link counts
off_t	File sizes
pid_t	Process IDs and process group ids
size_t	Unsigned integer
ssize_t	Signed arithmetic
uid_t	User IDs
time_t	Time values
clock_t	Time values
sigset_t	Signal set
cc_t	tty control chars
speed_t	tty baud rate
tcflag_t	tty modes
mtm_t	Mount requests
rdev_t	Device numbers

_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

_XOPEN_SOURCE

key_t	interprocess communications

_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

_OE_SOCKETS or _ALL_SOURCE

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

_XOPEN_SOURCE 500

blksize_t	Block sizes
blkcnt_t	File block counts
fsblkcnt_t	Filesystem block counts
fsfilcnt_t	File serial numbers
suseconds_t	Time values in range [-1,1,000,000]

_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 input/output 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_THLIIPADDR() macro, which sets a client's IP address for security facility authorization (SAF).

sys/utsname.h

The sys/utsname.h header file is supported only in a POSIX program by specifying the _POSIX_SOURCE feature test macro. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

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

sys/wait.h

This header file is supported only in a POSIX program by specifying the _POSIX_SOURCE and _XOPEN_SOURCE_EXTENDED 1 feature test macros. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

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 WorkLoad Manager functions.

tar.h

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

terminat.h

The ISO/ANSI C++ Standard (ISO/IEC 14882:1998(E)) supersedes this header with the new header <exception>. However, terminat.h remains for compatibility as a wrapper for TARGET releases of z/OS V1.2 and later.

For compilations with a TARGET release prior to z/OS V1.2, the terminat.h header file, which is used for z/OS C++ exception handling, declares the terminate() and set_terminate() functions.

termios.h

These functions are supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

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

cfgetispeed()	cfgetospeed()	cfsetispeed()	cfsetospeed()	tcdrain()
tcflow()	tcflush()	tcgetattr()	tcsendbreak()	tcsetattr()

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()	strftime()	strptime()	time()
tzset()[1]				

[1] These functions are supported only in a POSIX program. See "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7 for more information.

The time.h header file also provides:

- A structure `tm` containing the components of a calendar time. See "gmtime() — Convert Time to Broken-Down UTC Time" on page 391 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 40.
- 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 OpenExtensions times() function. In this context, we recommend using the OpenExtensions sysconf() function, which will eventually supersede CLK_TCK. (See "sys/times.h" on page 46 and "unistd.h" on page 51.)

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.

typeinfo

The typeinfo header file defines several types associated with the type-identification operator typeid, which yields information about both static and dynamic types.

```
namespace std {
    class type_info;
    class bad_cast;
    class bad_typeid;
    };
```

type_info

The class describes type information generated within the program by the implementation. Objects of this class effectively store a pointer to a name for the type, and an encoded value suitable for comparing two types for equality or collating order. The names, encoded values, and collating order for types are all unspecified and may differ between program executions.

An expression of the form typeid x is the only way to construct a (temporary) typeinfo object. The class has only a private copy constructor. Since the assignment operator is also private, you cannot copy or assign objects of class typeinfo either.

```
class type_info {
public:
    virtual ~type_info();
    bool operator==(const type_info& rhs) const;
    bool operator!=(const type_info& rhs) const;
    bool before(const type_info& rhs) const;
    const char *name() const;
private:
    type_info(const type_info& rhs);
    type_info& operator=(const type_info& rhs);
    };

type_info::operator!=

bool operator!=(const type_info& rhs) const;

The function returns !(*this == rhs).

type_info::operator==

bool operator==(const type_info& rhs) const;

The function returns a nonzero value if *this and rhs represent the
same type.

type_info::before

bool before(const type_info& rhs) const;

The function returns a nonzero value if *this precedes rhs in the
collating order for types.

type_info::name

const char *name() const;

The function returns a C string which specifies the name of the type.
```

bad_cast

```
class bad_cast : public exception {
    };
```

The class describes an exception thrown to indicate that a dynamic cast expression, of the form:

```
    dynamic_cast<type>(expression)
```

generated a null pointer to initialize a reference. The value returned by what() is an implementation-defined C string. None of the member functions throw any exceptions.

bad_typeid

```
class bad_typeid : public exception {
    };
```

The class describes an exception thrown to indicate that a typeid operator encountered a null pointer. The value returned by what() is an implementation-defined C string. None of the member functions throw any exceptions.

typeinfo.h

The ISO/ANSI C++ Standard (ISO/IEC 14882:1998(E)) supersedes this header with the new header <typeinfo>. While this header represents function that did not previously exist on the z/VM operating systems, it is being provided now for compatibility as a wrapper to <typeinfo>.

The typeinfo.h header file contains definitions for types associated with the type-identification operator typeid, which yields information about both static and dynamic types.

ucontext.h

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

getcontext()	makecontext()	setcontext()	swapcontext()

uheap.h

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

ucreate()	umalloc()	ufree()	uheapreport()

ulimit.h

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

unexpect.h

The ISO/ANSI C++ Standard (ISO/IEC 14882:1998(E)) supersedes this header with the new header <exception>. However, unexpect.h remains for compatibility as a wrapper for TARGET releases of z/OS V1.2 and later.

For compilations with a TARGET release prior to z/OS V1.2, the unexpect.h header file, which is used for z/OS C++ exception handling, declares the unexpected() and set_unexpected() functions.

unistd.h

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

access()	chdir()	chown()	close()	dup()
dup2()	extlink_np()	fchown()	fpathconf()	fsync()
ftruncate()	getcwd()	_isPosixOn()	link()	lseek()
pathconf()	read()	readlink()	rmdir()	symlink()
sysconf()	unlink()	write()		

The functions in a second group declared in unistd.h are supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

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

Attention: On z/VM, fork() is not full function. You should use this function only during a porting exercise until you can convert to something like spawn() or threading functions.

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:

_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

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. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

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 getsyntx() 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 *getsyntx(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 "Internationalization: Locales and Character Sets" topic in *z/OS: XL 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:

btowc()	fgetwc()	fgetws()	fputwc()	fputws()
getwc()	getwchar()	mbrlen()	mbrtowc()	mbsinit()
mbsrtowc()	putwc()	putwchar()	swprintf()	swscanf()
ungetwc()	vswprintf()	wcrtomb()	wcscat()	wcschr()
wcscmp()	wcscoll()	wcscpy()	wcscspn()	wcsftime()
wcslen()	wcsncat()	wcsncmp()	wcsncpy()	wcspbrk()
wcsrchr()	wcsrtombs()	wcsspn()	wcsstr()	wcstod()
wcstok()	wcstol()	wcstoll()	wcstoul()	wcstoull()
wcswidth()	wcsxfrm()	wctob()	wcwidth()	

You do not need to include stdio.h and stdarg.h to use the header file.

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

mbstate_t
Conversion-state information needed when converting between sequences of multibyte characters and wide characters.

size_t
`typedef` for the type of the value returned by *sizeof*.

wchar_t
`typedef` for a wide-character constant.

win_t
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).

FILE
The `FILE` structure type is defined in both stdio.h and wchar.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.

va_list
This type is defined in both stdarg.h and wchar.h.

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

NULL
A pointer that never points to a data object.

WEOF
Expands to a constant expression of type `wint_t`, whose value does not correspond to any member of the extended character set. It indicates EOF.

WCHAR_MIN
Defines the lower limit of the `wchar_t` type.

WCHAR_MAX
Defines the upper limit of the `wchar_t` type.

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 double-byte character set topic in *z/OS: XL 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 "Internationalization: Locales and Character Sets" topic in *z/OS: XL C/C++ Programming Guide*.

wcstr.h

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

wcscat()	wcschr()	wcscmp()	wcscpy()	wcscspn()
wcslen()	wcsncat()	wcsncmp()	wcsncpy()	wcspbrk()
wcsrchr()	wcsspn()	wcswcs()		

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 IBM XL C/C++ for z/VM 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 7. 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	Could 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
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	Datagram error indication
T_ORDREL	Orderly release indication

Table 7. Symbolic Constants defined in xti.h (continued)

Symbolic Constant	Description
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

Table 7. Symbolic Constants defined in xti.h (continued)

Symbolic Constant	Description
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
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_LTPDUDFLT	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

Table 7. Symbolic Constants defined in xti.h (continued)

Symbolic Constant	Description
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_CONNRESIL	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_KEEPALIVE	TCP check if connections are alive option
T_GARBAGE	Garbage value for t_kpalive kp_timeout value
INET_UDP	UDP options level
UDP_CHECKSUM	UDP checksum option
INET_IP	IP options level
IP_OPTIONS	IP per-packet options

Table 7. Symbolic Constants defined in xti.h (continued)

Symbolic Constant	Description
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()	

Chapter 3. Library Functions

This chapter describes the IBM XL C/C++ for z/VM library functions, including the built-in library functions used by the IBM XL C/C++ for z/VM compiler.

Names

The following identifiers are reserved. You should not use them for any purpose other than those defined in the documentation.

- ISO C (formerly ANSI C) identifiers (function names, macros, types) defined in the ISO C headers
- Identifiers that begin with an underscore and either an uppercase letter or another underscore

All identifiers other than the ISO C identifiers comprise the *user's name space*, and you are free to use any of these names. A number of names in the IBM XL C/C++ for z/VM library, however, 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 IBM XL C/C++ for z/VM library that 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 XL C/C++ for z/VM compiler automatically maps all underscores and lowercase 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 the compiler will automatically map to the external entry point @@FTCHEP. For C++ functions, you must ensure C by declaring the functions as `extern "C"`.

See the following topics in *z/OS: XL C/C++ Language Reference (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/cbclx01_v2r5.pdf)* for more information about external names:

- #pragma csect
- #pragma map
- External name mapping

Also see the following topics in the *XL C/C++ for z/VM: User's Guide*:

- Prelinking a C application
- `LONGNAME` compiler option

Also see the following topics in *z/OS: XL C/C++ Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/cbcpx01_v2r5.pdf)*:

- Naming conventions
- Using environment variables

Standards

Each function description begins with a table like the following one to indicate the standards, extensions, and dependencies:

© Copyright IBM Corp. 2002, 2024

Standards/Extensions	C or C++	Dependencies
POSIX.1 XPG4.2 OpenExtensions	both	POSIX(ON)

The standard refers 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:

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. *OpenExtensions* refers to functions that provide VM-specific support beyond the defined standards.
7. POSIX(ON) *required* refers to whether the use of the function is restricted to a IBM XL C/C++ for z/VM program in OpenExtensions environment running POSIX(ON). For more information about running POSIX programs, please see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

These standards do have some overlap, as illustrated in Figure Figure 2 on page 62.

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.)

Online Examples

Examples in this book are available in machine-readable form. They have a file name of CBC3*yyyy* and a file type of C. These sample files are located (by default) on the P654033A 2C2 minidisk or the VMSYS:P654033A.C390.SAMPLE directory.

The samples should be installed on your system along with the IBM XL C/C++ for z/VM compiler. Contact your system programmer if the default minidisk or SFS directory was not used at your site.

Built-in Functions

Built-in functions are functions 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.

Application code uses built-in functions 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.

The built-in functions are listed in the following table. In the list of library functions, each built-in function is labelled as such.

abs()	cds()	cs()	decabs()	decchk()
decfix()	fabs()	memchr()	memcmp()	memcpy()
memset()	strcat()	strchr()	strcmp()	strcpy()
strlen()	strrchr()			

Preprocessor macros, which are defined in the standard header files, access the built-in versions of these functions. To use these macros, you must include the appropriate header file (such as .*`decimal.h`, `math.h`, `stdlib.h`, or `string.h`) in the source 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

z/VM (including the associated Language Environment and C/C++ compiler) includes 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.

Note:

1. 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 *hexadecimal floating-point*, respectively.
2. Under hexadecimal floating-point format, the rounding mode is set to round toward 0. Under IEEE floating-point format, the rounding mode is to round toward the nearest integer.

The C/C++ compiler provides a FLOAT option to select the format of floating-point numbers produced in a compile unit. The FLOAT option allows you to select either IEEE floating-point or hexadecimal floating-point format. For details on the C/C++ support, see the description of the FLOAT option in *z/OS: XL C/C++ User's Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/cbcux01_v2r5.pdf)*. In addition, two related sub-options, ARCH(3) and TUNE(3), support the IEEE binary floating-point data. For details, see the ARCHITECTURE and TUNE compiler options in *z/OS: XL C/C++ User's Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/cbcux01_v2r5.pdf)*.

The C/C++ runtime library interfaces support IEEE floating-point and hexadecimal floating-point formats.

The primary documentation for the IEEE floating-point support is contained in *z/Architecture Principles of Operation (https://publibfp.dhe.ibm.com/epubs/pdf/a227832d.pdf)* and *z/OS: XL C/C++ User's Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/cbcux01_v2r5.pdf)*.

IBM does not recommend mixing floating-point formats in an application. However, for applications that must handle both formats, the C/C++ runtime library does offer some support. Reference information for IEEE floating-point can also be found in *z/OS: XL C/C++ Language Reference (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/cbclx01_v2r5.pdf)*.

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). Functions to access the thread-specific values of these variables are provided for use when running in a multithreaded environment.

errno

When a runtime 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 _dlght() 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.

_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 _optargf() 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.

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 _tzone() 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.

Library Function Descriptions

This section contains an alphabetical list and a description of the IBM XL C/C++ for z/VM library functions.

abort() — Stop a Program

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.

Note: 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 situation 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 POSIX programs, you can jump out of the handler with sigsetjmp() and siglonjmp().

For more information, see the process termination sections in *z/OS: XL C/C++ Programming Guide*.

Special Behavior for POSIX C

To obtain access to the special POSIX behavior for abort(), the POSIX runtime option must be set on.

Calls to abort() raise the SIGABRT signal, using 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 "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7 for more information about using POSIX support.

Special Behavior for C++

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

By default, the C++ function terminate() calls abort().

Returned Value

The abnormal termination return code for z/VM 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 42
- "assert() — Verify Condition" on page 81
- "atexit() — Register Program Termination Function" on page 85
- "exit() — End Program" on page 197
- "pthread_kill() — Send a Signal to a Thread" on page 586
- "raise() — Raise Signal" on page 623
- "signal() — Handle Interrupts" on page 767

abs(), absf(), absl() — Calculate Integer Absolute Value

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

Format

```
#include <stdlib.h>

int abs(int n);
long abs(long n);                        /* C++ only */

#include <math.h>

double abs(double n);                    /* C++ only */
float abs(float n);                        /* C++ only */
long double abs(long double n);        /* C++ only */
float absf(float n);
long double absl(long double n);
```

General Description

Returns the absolute value of an integer *n*.

The minimum allowable integer is INT_MIN+1. (INT_MIN is a macro that is defined in the limits.h header file.) The z/VM default for INT_MIN+1 is -2147483648. If the value entered cannot be represented as an integer, abs() returns the same value.

For the double, float, and long double versions of abs(), the minimum allowable values are DBL_MIN+1 , FLT_MIN+1 , and LDBL_MIN+1 , respectively. (The floating-point macro constants are defined in the float.h header file.)

If the value entered cannot be represented as an integer, the abs(), absf(), and absl() functions return the same value.

Note: These functions work in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

Special Behavior for C++

For VM C++ applications, abs() is also overloaded for the types long, float, and long double.

Returned Value

Returns the absolute value if it is possible to represent. Otherwise, abs() returns the input value.

No errnos are defined for abs().

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 42
- "fabs(), fabsf(), fabsl() — Calculate Floating-Point Absolute Value" on page 200
- "labs() — Calculate Long Absolute Value" on page 443

accept() — Accept a New Connection on a Socket

Standards/Extensions	C or C++	Dependencies
POSIX.1 XPG4.2 OpenExtensions	both	POSIX(ON)

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

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 a value of -1 and sets errno to EWOULDBLOCK. The new socket descriptor is in active mode and cannot be used to accept new connections. The original socket, *socket*, remains in passive mode and available to accept more connection requests.

socket
: The socket descriptor of the passive socket on which connections are to be accepted. That is, it is a socket which was previously placed into passive mode ("listening mode") by a call to listen().

address
The socket address of the connecting client that is filled in by accept() before it returns. The format of *address* is determined by the domain in which the client resides. This parameter can be NULL if the caller is not interested in the client address.

address_len
Must initially point to an integer that contains the size in bytes of the storage pointed to by *address*. On return, that integer contains the size of the data returned in the storage pointed to by *address*. If *address* is NULL, *address_len* is ignored.

The *socket* parameter is a stream socket file 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, or if the connection request originated in the AF_INET6 domain, *address* points to a **sockaddr_in6** structure. The **sockaddr_in** and **sockaddr_in6** structures are defined in **netinet/in.h**. The *address_len* parameter is used only if the address 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. You cannot 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

If successful, accept() returns a nonnegative socket descriptor.

If unsuccessful, accept() returns a value of -1 and sets errno to one of the following values under the conditions described:

EBADF
The *socket* parameter is not a valid file descriptor.

EFAULT
Using *address* and *address_len* would result in an attempt to copy the address into a portion of the caller's virtual machine into which information cannot be written.

EIBMIUCVERR
An IUCV error occurred.

EINTR
A signal interrupted the accept() call before any connections were available.

EINVAL
The listen() function was not called for socket descriptor *socket* because it is not in passive mode.

EIO
A network or transport failure has occurred.

EMFILE
An attempt was made to open more than the maximum number of file descriptors allowed for this process.

ECMSERR
Two consecutive accept calls that cause an identity change are not allowed. The original identity must be restored (close() 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 *socket* 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 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 92
- “connect() — Connect a Socket” on page 138
- “getpeername() — Get Name of Peer Connected to a Socket” on page 353
- “listen() — Prepare Server for Incoming Client Requests” on page 452
- “socket() — Create a Socket” on page 791

access() — Determine Whether a File Can be Accessed

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

Format

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

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

General Description

Determines how a BFS 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.

The *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.

Returned Value

Returns a value of `0` if the specified access is permitted. If the given file cannot be accessed in the specified way, access() returns a value of -1.

If unsuccessful, access() sets errno to one of the following values under the conditions described:

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 *pathname* prefix.

EINVAL
The value of *how* is incorrect.

ELOOP
A loop exists in the symbolic links.

ENAMETOOLONG
pathname is longer than PATH_MAX characters. The PATH_MAX value is determined using pathconf().

ENOENT
No file named *pathname* exists, or *pathname* is an empty string.

ENOTDIR
Some component of the *pathname* prefix is not a directory.

EROFS
The argument *how* 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 POSIX_SYMLOOP (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, when _POSIX_NO_TRUNC (defined in the unistd.h header file) is in effect. The PATH_MAX and NAME_MAX values are determined using pathconf().

See "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7 for more information about using POSIX support.

Example

CBC3BA03:

```
/* CBC3BA03
   The following example determines how a file is accessed.
 */
#define _OPEN_SYS
#include <unistd.h>
#undef _OPEN_SYS
#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 28
- "unistd.h" on page 51
- "chmod() — Change the Mode of a File or Directory" on page 117
- "stat() — Get File Information" on page 803

acos(), acosf(), acosl() — Calculate Arccosine

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

Format

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

```
double acos(double x);
float acos(float x);                        /* C++ only */
long double acos(long double x);        /* C++ only */
float acosf(float x);
long double acosl(log double x);
```

General Description

Calculates the arccosine of *x*, expressed in radians, in the range 0 to π.

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

Note: These functions work in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

Returned Value

If *x* is less than -1 or greater than 1, the function sets errno to EDOM and returns a value of `0`. If the correct value would cause underflow, a value of `0` is returned and the value of the macro ERANGE is stored in errno.

Special Behavior for XPG4.2

If successful, the function returns the arc cosine of *x*, in the range [`0`,π] radians. If the value of *x* is not in the range [`-1,1`], the function returns a value of `0.0` and sets errno to EDOM.

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

Special Behavior for IEEE

If successful, the function returns the arccosine of the argument *x*.

If *x* is less than -1 or greater than 1, the function sets errno to EDOM and returns NaNQ (Not a Number Quiet). 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 32
- “asin(), asinf(), asinl() — Calculate Arcsine” on page 79
- “atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate Arctangent” on page 83
- “cos(), cosf(), cosl()— Calculate Cosine” on page 143
- “cosh(), coshf(), coshl()— Calculate Hyperbolic Cosine” on page 145
- “sin() — Calculate Sine” on page 785
- “sinh() — Calculate Hyperbolic Sine” on page 786
- “tan() — Calculate Tangent” on page 861
- “tanh() — Calculate Hyperbolic Tangent” on page 862

acosh() — Calculate Hyperbolic Arcosine

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

Format

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

double acosh(double x);
```

General Description

Calculates the hyperbolic arccosine of *x*.

Note: This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE floating-point” on page 63 for more information about IEEE floating-point.

Returned Value

If successful, acosh() returns the hyperbolic arccosine of *x*.

If the value of *x* is less than `1.0`, then acosh() returns a value of `0` and sets errno to EDOM.

Special Behavior for IEEE

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

If *x* is less than 1.0, acosh() sets errno to EDOM and returns NaNQ.

Related Information

- “math.h” on page 32
- “acos(), acosf(), acosl() — Calculate Arccosine” on page 73
- “asin(), asinf(), asinl() — Calculate Arcsine” on page 79
- “asinh() — Calculate Hyperbolic Arcsine” on page 81
- “atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate Arctangent” on page 83
- “atanh() — Calculate Hyperbolic Arctangent” on page 84

- “cos(), cosf(), cosl()— Calculate Cosine” on page 143
- “cosh(), coshf(), coshl()— Calculate Hyperbolic Cosine” on page 145
- “sin() — Calculate Sine” on page 785
- “sinh() — Calculate Hyperbolic Sine” on page 786
- “tan() — Calculate Tangent” on page 861
- “tanh() — Calculate Hyperbolic Tangent” on page 862

alarm() — Set an Alarm

Standards/Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	POSIX(ON)

Format

```
#define _OPEN_SYS
#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 process.

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 “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7 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() interacts with the setitimer() function when the setitimer() function is used to set the 'real' interval timer (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 a value of `0`. Because alarm() is always successful, 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 _OPEN_SYS
#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 38
- “unistd.h” on page 51
- “exec Functions” on page 192
- “pause() — Suspend a Thread Pending a Signal” on page 535
- “sigaction() — Examine or Change a Signal Action” on page 749
- “signal() — Handle Interrupts” on page 767
- “sigprocmask() — Examine or Change a Thread” on page 773
- “sleep() — Suspend Execution of a Process” on page 787

asctime() — Convert Time to Character String

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`.

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, 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 49
- "ctime() — Convert Time to a Character String" on page 153
- "gmtime() — Convert Time to Broken-Down UTC Time" on page 391
- "localtime() — Convert Time and Correct for Local Time" on page 456
- "mktime() — Convert Local Time" on page 505
- "strftime() — Convert to Formatted Time" on page 821
- "time() — Determine Current Time" on page 885

asin(), asinf(), asinl() — Calculate Arcsine

Standards/Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++	both	POSIX(ON)

Format

```
#include <math.h>

double asin(double x);
float asin(float x);                    /C+++only /
long double asin(long double x);    /C+++only /
float asinf(float x);
long double asinl(long double x);
```

General Description

Calculates the arcsine of *x*, in the range -π/2 to π/2 radians.

The value of *x* must be between -1 and 1.

Note: These functions work in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 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.

Special Behavior for IEEE

If successful, the function returns the arcsine of its argument *x*.

If *x* is less than -1 or greater than 1, the function sets errno to EDOM and returns NaNQ. No other errors will occur.

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 32
- "acos(), acosf(), acosl() — Calculate Arccosine" on page 73
- "atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate Arctangent" on page 83
- "cos(), cosf(), cosl()— Calculate Cosine" on page 143
- "cosh(), coshf(), coshl()— Calculate Hyperbolic Cosine" on page 145
- "sin() — Calculate Sine" on page 785
- "sinh() — Calculate Hyperbolic Sine" on page 786

- “tan() — Calculate Tangent” on page 861
- “tanh() — Calculate Hyperbolic Tangent” on page 862

asinh() — Calculate Hyperbolic Arcsine

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

Format

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

double asinh(double x);
```

General Description

Calculates the hyperbolic arcsine of *x*.

Note: This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE floating-point” on page 63 for more information about IEEE floating-point.

Returned Value

Returns the hyperbolic arcsine of *x*. The asinh() function is always successful.

Related Information

- “math.h” on page 32
- “acos(), acosf(), acosl() — Calculate Arccosine” on page 73
- “acosh() — Calculate Hyperbolic Arcosine” on page 75
- “asin(), asinf(), asinl() — Calculate Arcsine” on page 79
- “atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate Arctangent” on page 83
- “atanh() — Calculate Hyperbolic Arctangent” on page 84
- “cos(), cosf(), cosl()— Calculate Cosine” on page 143
- “cosh(), coshf(), coshl()— Calculate Hyperbolic Cosine” on page 145
- “sin() — Calculate Sine” on page 785
- “sinh() — Calculate Hyperbolic Sine” on page 786
- “tan() — Calculate Tangent” on page 861
- “tanh() — Calculate Hyperbolic Tangent” on page 862

assert() — Verify Condition

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

Format

```
#include <assert.h>

void assert(int expression);
```

General Description

A macro that 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 preprocessor expands all assert() invocations to `void` expressions.

Note: Using the `#undef` directive with the assert() macro results in undefined behavior. The assert() macro uses _FILE_ and _LINE_.

Returned Value

The assert() function has 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);
   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 18
- "abort() — Stop a Program" on page 66

atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate Arctangent

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

Format

```
#include <math.h>

double atan(double x);
float atan(float x);                    /C+++only /
long double atan(long double x);    /C+++only /
float atanf(float x);
long double atanl(long double x);

double atan2(double y, double x);
float atan2(float y,float x);                                /C+++only /
long double atan2(long double y,long double x);    /C+++only /
float atan2f(float y,float x);
long double atan2l(long double y,long double x);
```

General Description

The atan() and atan2() functions calculate the arctangent of *x* and *y/x*, respectively.

Note: These functions work in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

Returned Value

The atan() functions returns a value in the range -π/2 to π/2 radians. The atan2() functions returns a value in the range -π to π 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.

Special Behavior for IEEE

If successful, atan2() returns the arctangent of *y/x*.

If both arguments of atan2() are zero, the function sets errno to EDOM and returns 0. No other errors will occur.

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 32
- "acos(), acosf(), acosl() — Calculate Arccosine" on page 73
- "asin(), asinf(), asinl() — Calculate Arcsine" on page 79
- "cos(), cosf(), cosl()— Calculate Cosine" on page 143
- "cosh(), coshf(), coshl()— Calculate Hyperbolic Cosine" on page 145
- "sin() — Calculate Sine" on page 785
- "sinh() — Calculate Hyperbolic Sine" on page 786
- "tan() — Calculate Tangent" on page 861
- "tanh() — Calculate Hyperbolic Tangent" on page 862

atanh() — Calculate Hyperbolic Arctangent

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

Format

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

double atanh(double x);
```

General Description

Calculates the hyperbolic arctangent of *x*.

Note: This function works in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

Returned Value

If successful, atanh() returns the hyperbolic arctangent of *x*.

If unsuccessful, atanh() returns a value of `0` and sets errno to one of the following values under the conditions described:

EDOM
x has a value greater than `1.0`.

ERANGE
x has a value equal to `1.0`.

Special Behavior for IEEE

If successful, atanh() returns the hyperbolic arctangent of its argument *x*.

If the absolute value of *x* is greater than 1, atanh() sets errno to EDOM and returns NaNQ. If the value of *x* is equal to 1, atanh() sets errno to ERANGE and returns +HUGE_VAL.

Related Information

- "math.h" on page 32
- "acos(), acosf(), acosl() — Calculate Arccosine" on page 73
- "acosh() — Calculate Hyperbolic Arcosine" on page 75
- "asin(), asinf(), asinl() — Calculate Arcsine" on page 79
- "asinh() — Calculate Hyperbolic Arcsine" on page 81
- "atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate Arctangent" on page 83
- "cos(), cosf(), cosl()— Calculate Cosine" on page 143
- "cosh(), coshf(), coshl()— Calculate Hyperbolic Cosine" on page 145
- "sin() — Calculate Sine" on page 785
- "sinh() — Calculate Hyperbolic Sine" on page 786
- "tan() — Calculate Tangent" on page 861
- "tanh() — Calculate Hyperbolic Tangent" on page 862

atexit() — Register Program Termination Function

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 OpenExtensions only, when a process ends, the address space is ended; otherwise, the address space persists.

Special Behavior for C

The atexit() function has the following restrictions:

- Any function registered by a released fetched module is removed from the list at the time of release(). (See fetch(), fetchep(), and release() for details about fetching and releasing modules.)
- All C library functions can be used in a registered routine except atexit() and exit().

- 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 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

If successful, atexit() returns a value of zero.

If unsuccessful, atexit() returns a value of nonzero.

Example

CBC3BA10

```
/* CBC3BA10
   This example uses the atexit() function to call the function goodbye()
   at program termination.
 */
#include <stdlib.h>
#include <stdio.h>

void goodbye(void);

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

- “stdlib.h” on page 42
- “abort() — Stop a Program” on page 66

- “exit() — End Program” on page 197
- “fetch() — Get a Load Module” on page 221
- “fetchep() — Share Writable Static” on page 231
- “release() — Delete a Load Module” on page 657
- “signal() — Handle Interrupts” on page 767

__atoe() — Convert ISO8859-1 String to EBCDIC

Standards/Extensions	C or C++	Dependencies
OpenExtensions	both	

Format

```
#include <unistd.h>

int __atoe(char *string);
```

General Description

Converts an ISO8859-1 character string, pointed to by *string*, to its EBCDIC equivalent using the codeset page associated with the current locale.

The __atoe() function changes all characters in the input string up to, but not including, the null from an ISO8859-1 representation to that of the current locale.

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 a value of -1 and sets errno to one of the following values under the conditions described.

EINVAL
The current locale does not describe a single-byte character set.

ENOMEM
Insufficient storage space prevents completion of the conversion process.

Note: This function may internally call iconv_open() and iconv(), which return errnos that are propagated without modification.

Related Information

- “sys/msg.h” on page 45

__atoe_l() — Convert ISO8859-1 Bytes to EBCDIC

Standards/Extensions	C or C++	Dependencies
OpenExtensions	both	

Format

```
#include <unistd.h>

int __atoe_l(char * bufferptr, int leng);
```

General Description

Converts the *leng* of ISO8859-1 bytes in the buffer, *bufferptr*, to the EBCDIC equivalent using the codeset page associated with the current locale.

The __atoe_l() function treats the input buffer as a sequence of bytes, and converts all bytes in the input buffer, including any imbedded null characters.

Returned Value

If successful, __atoe_l() converts the input IOS8859-1 bytes to the equivalent ebcdic value and returns the number of bytes that it converted.

If unsuccessful, __atoe_l() returns a value of -1 and sets errno to one of the following values under the conditions described.

EINVAL
The current locale does not describe a single-byte character set.

ENOMEM
Insufficient storage space prevents completion of the conversion process.

Note: This function may internally call iconv_open() and iconv(), which return errnos that are propagated without modification.

Related Information

- "sys/msg.h" on page 45

atof() — Convert Character String to Double

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

Format

```
#include <stdlib.h>

double atof(const char *string);
```

General Description

This function is defined in terms of strtod(). Refer to "strtod() — Convert Character String to Double" on page 840.

Related Information

- "stdlib.h" on page 42
- "fscanf() - scanf() - sscanf() — Read and Format Data" on page 283
- "strtod() — Convert Character String to Double" on page 840

atoi() — Convert Character String to Integer

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

Format

```
#include <stdlib.h>

int atoi(const char *string);
```

General Description

Converts the initial portion of the string pointed to by *nptr* to an 'int'. This is equivalent to:

```
(int)strtol(nptr, NULL, 10)
```

Refer to “strtol() — Convert Character String to Long” on page 843.

Related Information

- “stdlib.h” on page 42
- “atol() — Convert Character String to Long” on page 89
- “fscanf() - scanf() - sscanf() — Read and Format Data” on page 283
- “strtol() — Convert Character String to Long” on page 843

atol() — Convert Character String to Long

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

Format

```
#include <stdlib.h>

long int atol(const char *string);
```

General Description

Converts the initial portion of the string pointed to by *nptr* to a 'long int'. This is equivalent to:

```
strtol(nptr, NULL, 10)
```

Refer to “strtol() — Convert Character String to Long” on page 843.

Related Information

- “stdlib.h” on page 42

- “atoi() — Convert Character String to Integer” on page 89
- “fscanf() - scanf() - sscanf() — Read and Format Data” on page 283
- “strtol() — Convert Character String to Long” on page 843

a64l() — Convert Base-64 String Representation to Long Integer

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

Converts a string representation of a base-64 number into its corresponding long value. The a64l() function 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 *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 table provides the character-digit equivalents:

Character
: **Digit represented**

.
: 0

/
: 1

0-9
: 2-11

A-Z
: 12-37

a-z
: 38-63

Returned Value

If successful, a64l() returns the long value resulting from conversion of the input string. If *string*, is null, a64l() returns a value of `0`.

No errnos are defined for a64l().

Related Information

- “stdlib.h” on page 42
- “l64a() — Convert Long to Base-64 String Representation” on page 474
- “strtoul() — Convert String to Unsigned Integer” on page 845

basename() — Return the Last Component of a Path Name

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

Format

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

char *basename(char *path);
```

General Description

Returns a pointer to the final component of the path name, *path*, 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

Returns a pointer to the final component of *path*.

No errnos are defined for basename().

Related Information

- “libgen.h” on page 28
- “dirname() — Report the Parent Directory of a Path Name” on page 167

bcmp() — Compare Bytes in Memory

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

Compares the first *n* bytes of the area pointed to by *s1* with the area pointed to by *s2*.

Returned Value

Returns a value of zero if *s1* and *s2* are identical and a value of nonzero otherwise. Both areas are assumed to be at least *n* bytes long.

If the value of *n* is zero, bcmp() returns a value of zero.

No errnos are defined for bcmp().

Related Information

- "strings.h" on page 44
- "bcopy() — Copy Bytes in Memory" on page 92
- "bzero() — Zero Out Bytes in Memory" on page 102
- "memcmp() — Compare Bytes" on page 493

bcopy() — Copy Bytes in Memory

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

Copies *n* bytes from the area pointed to by *s1* to the area pointed to by *s2* using the memcpy() function. The behavior of bcopy() is undefined if copying takes place between objects that overlap.

Returned Value

Returns no values.

No errnos are defined for bcopy().

Related Information

- "strings.h" on page 44
- "bcmp() — Compare Bytes in Memory" on page 91
- "bzero() — Zero Out Bytes in Memory" on page 102
- "memccpy() — Copy Bytes in Memory" on page 492
- "memcpy() — Copy Buffer" on page 495
- "memmove() — Move Buffer" on page 496

bind() — Bind a Name to a Socket

Standards/Extensions	C or C++	Dependencies
POSIX.1 XPG4.2 OpenExtensions	both	POSIX(ON)

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

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.

socket
: The socket descriptor returned by a previous socket() call.

address
: The pointer to a `sockaddr` structure containing the name that is to be bound to *socket*.

address_len
: The size of *address* in bytes.

The *socket* parameter is a socket descriptor of any type created by calling socket().

The *address* parameter is a pointer to a buffer containing the name to be bound to *socket*. The *address_len* parameter is the size, in bytes, of the buffer pointed to by *address*. For AF_UNIX, this function creates a file that you later need to unlink besides closing the socket.

Sockets 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 netinet/in.h header file.

```
struct in_addr
{
        in_addr_t s_addr;
};

 struct sockaddr_in {
     unsigned char  sin_len;      /* length of sockaddr struct */
     unsigned char  sin_family;   /* addressing family */
     unsigned short sin_port;     /* port number */
     struct in_addr sin_addr;     /* IP address */
     unsigned char  sin_zero[8];  /* unassigned */

};
```

The *sin_len* field must be set to either `0` or `sizeof(struct sockaddr_in)`. Both values are treated the same.

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 multihomed 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.

Sockets in the AF_INET6 Domain

If the socket descriptor *socket* was created in the AF_INET6 domain, the format of the name buffer is expected to be sockaddr_in6, as defined in netinet/in.h:

```
 struct sockaddr_in6 {
     uint8_t         sin6_len;       /* length of sockaddr structure */
     sa_family_t     sin6_family;    /* addressing family */
     in_port_t       sin6_port;      /* port number */
     uint32_t        sin6_flowinfo;  /* ignored */
     struct in6_addr sin6_addr;      /* IP address */
     uint32_t        sin6_scope_id;  /* scope ID */
};
```

The *sin6_len* field must be set to either `0` or `sizeof(struct sockaddr_in6)`. Both values are treated the same.

The *sin6_family* must be set to AF_INET6.

The *sin6_port* field is set to the port to which the socket is bound. It must be specified in network byte order.

The *sin6_flowinfo* field is currently unsupported so its contents are ignored.

The *sin6_addr.s6_addr* field is set to the internet address of the interface to which the socket is bound. It must be specified in network byte order.

The *sin6_scope_id* field identifies a set of interfaces as appropriate for the scope of the address carried in the *sin6_addr* field. For link local addresses, the *sin6_scope_id* can be used to specify the outgoing interface index. The z/VM stack supports *sin6_scope_id* for link local addresses only

Sockets in the AF_IUCV Domain

If the socket descriptor *socket* was created in the AF_IUCV domain, the format of the name buffer is expected to be `sockaddr_iucv`, as defined in the saiucv.h header file.

```
struct sockaddr_iucv {
    unsigned char  siucv_len;       /* length of sockaddr struct */
    unsigned char  siucv_family;    /* addressing family */
    unsigned short siucv_port;      /* port number */
    unsigned long  siucv_addr;      /* address */
    unsigned char  siucv_nodeid[8]; /* nodeid to connect to */
    unsigned char  siucv_userid[8]; /* userid to connect to */
    unsigned char  siucv_name[8];   /* iucvname for connect */
};
```

The *siucv_len* field must be set to either `0` or `sizeof(struct sockaddr_iucv)`. Both values are treated the same.

The *siucv_family* field must be set to AF_IUCV.

The *siucv_port*, *siucv_addr*, and *siucv_nodeid* fields are reserved for future use. The *siucv_port* and *siucv_addr* fields must be zeroed. The *siucv_nodeid* field must be set to exactly 8 blank characters.

The *siucv_userid* field need not be initialized for a bind() call. The VM user ID running the calling program will be used.

The *siucv_name* field is set to the application name by which the socket is to be known. It must be unique within the virtual machine because only one socket can be bound to a given name. The recommended form of the name contains 8 characters, padded with blanks to the right. The 8 characters for a connect() call executed by a client must exactly match the 8 characters passed in the bind() call executed by

the server. This name is passed, by the sockets library, as the NAME parameter on invocations of the CMS HNDIUCV and CMSIUCV macros. It must be unique, therefore, among all the names used for these macros by any programs running in the virtual machine.

Note: Internally, dynamic names are built using hexadecimal character strings representing the internal memory address of the socket. An example of this is when an application calls connect() and specifies an unbound socket. You should choose names that contain at least one non-hexadecimal character to prevent potential conflicts. Hexadecimal characters include 0-9 and a-f. Uppercase A-F are not considered hexadecimal in this instance and can be used by the user in building dynamic names.

Sockets 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 sys/un.h header file.

```
struct sockaddr_un {
    unsigned char  sun_len;        /* length of sockaddr struc */
    unsigned char  sun_family;     /* addressing family */
             char  sun_path[108];  /* pathname */
};
```

The *sun_len* field must be set to either `0` or a value greater than or equal to SUN_LEN(&*sa*), where *sa* is the name of the *sockaddr_un* variable, but less than or equal to `sizeof(struct sockaddr_un)`. The SUN_LEN() macro, which is defined in sys/un.h, evaluates to an expression that returns the total length of the used portion of the `sockaddr_un` structure, when *sun_path* has been filled in with a null-terminated file name. The length returned by SUN_LEN() does not include the terminating null character. The SUN_LEN() macro may be used to derive a value for the *address_len* argument too. If `0` is specified for *sun_len*, the *address_len* value determines how long the path name is. If *sun_len* is nonzero, the lesser of *sun_len* and *address_len* is used. In either case, if a null character appears in the string before the given length, path name is considered to end there.

The *sun_family* field must be set to AF_UNIX.

The *sun_path* field should contain the name of the file that will represent the open socket. It need not be null delimited, although it is recommended that it is, so that the SUN_LEN() macro can be used. A file by this name will be created in the Byte File System, so the form of the path name should follow the POSIX conventions. Because the client process must specify this file on its connect() function call, the path name specified should be an absolute path name (beginning with a single slash character). The bind() call will not replace an existing file with the given name, so the unlink() function should be used first to erase any such file, if the file may already exist (say, from a previous, perhaps abnormally terminated, execution of the program). Similarly, the close() function will not erase this file when the socket is closed, so the application should use unlink() after closing the socket to erase the file. The file must exist when any other application uses the connect() function to connect to the socket.

Notes:

- 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, you should issue a chmod() to modify this mode.
- For AF_UNIX, when closing sockets that were bound, you should also use unlink() to delete the file created at bind() time.
- The path name the client uses on the bind() must be unique.
- The sendto() call must specify the path name associated with the server.
- For AF_INET and AF_INET6, 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

If successful, bind() returns a value of `0`.

If unsuccessful, bind() returns a value of -1 and sets errno to one of the following values under the conditions described:

EACCES
Permission is denied.

For AF_INET and AF_INET6 only, permission is denied. A user that is not in the obeylist attempted to bind to a port between 0-1023 when RESTRICTLOWPORTS has been specified on the ASSORTEDPARMS statement. Or, a user attempted to bind to a port that has previously been reserved using a PORT statement in the TCP/IP cofiguration file or through obeyfile processing.

EADDRINUSE
For AF_INET AF_INET6, and AF_IUCV only, the address is already in use. For information about the AF_INET domain, see the SO_REUSEADDR option described under "getsockopt() — Get Options Associated with a Socket" on page 370 and the SO_REUSEADDR option described under "setsockopt() — Set Options Associated with a Socket" on page 734.

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_INET, AF_INET6, AF_IUCV, or AF_UNIX), or the *family* field in the sockaddr structure does not match the family value specified on the socket() function.

EBADF
The *socket* parameter is not a valid file descriptor.

EFAULT
The *address* and *address_len* parameters result in an attempt to copy the address into a nonwritable portion of the caller's virtual storage.

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. This error also occurs for the AF_IUCV domain if the *siucv_nodeid* field in *address* is not all blanks or if the *siucv_addr* or *siucv_port* fields are nonzero.

EIO
A network or transport failure has occurred.

ENOBUFS
The bind() function is unable to obtain a buffer due to insufficient storage.

ENOTSOCK
The descriptor is a valid file descriptor, but it is not 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:

EACCES
The process does not have search permission on a component of the path prefix, or it does not have write access to the directory of the requested name.

EDESTADDRREQ
The *address* argument is a null pointer.

EEXIST
The file name already exists. You should issue unlink() before issuing bind().

EIO
An I/O error occurred.

ELOOP
Too many symbolic links were encountered in translating the path name in *address*.

ENAMETOOLONG
A component of a path name exceeded NAME_MAX characters, or an entire path name exceeded PATH_MAX characters.

ENOENT
A component of the path name does not name an existing file or the path name is an empty string.

ENOTDIR
A component of the path prefix of the path name in *address* is not a directory.

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_INET6 Domain Example

The following example illustrates the bind() call binding to interfaces in the AF_INET6 domain. Note that inet_pton() is used instead of inet_addr(), and that inet_pton() fills in all of the sin6_addr field.

```
int                  s;  /* Socket descriptor. */
int                  rc;
char                 v6addr[16] = "C2C1::129:5:24:1";  /* IPv6 IP address. */
struct sockaddr_in6  myname;
struct in6_addr      IN6ADDR_ANY = IN6ADDR_ANY_INIT;  /* Init required by VM. */

/* Bind to a specific interface in the AF_INET6 domain. */
/* (Not specifying flow info and scope ID.)             */
```

```
memset(&myname,0,sizeof(myname));
myname.sin6_len = sizeof(myname);
myname.sin6_family = AF_INET6;
myname.sin6_addr = inet_pton(AF_INET6,(char *)&v6addr,&(myname.sin6_addr));
myname.sin6_port = htons(1024);
rc = bind(s,struct sockaddr *)&myname,sizeof(myname));

/* Bind to all network interfaces in the AF_INET6 domain. */
memset(&myname,0,sizeof(myname));
myname.sin6_len = sizeof(myname);
myname.sin6_family = AF_INET6;
myname.sin6_addr = IN6ADDR_ANY;
myname.sin6_port = htons(1024);
rc = bind(s,struct sockaddr *)&myname,sizeof(myname));
```

AF_IUCV Domain Example

The following example illustrates the bind() call binding to interfaces in the AF_IUCV domain.

```
int rc;
int s;
struct sockaddr_iucv myname;
#define SERVER_NAME "APPL1"
/* Bind to a name in the IUCV domain */
/* make sure the siucv_addr, siucv_port fields are zeroed and the
   siucv_nodeid field is set to blanks */
memset(&myname, 0, sizeof myname);
myname.siucv_family = AF_IUCV;
memset(myname.siucv_nodeid, ' ', sizeof myname.siucv_nodeid);
memset(myname.siucv_name, ' ', sizeof myname.siucv_name);
memcpy(myname.siucv_name, SERVER_NAME, sizeof SERVER_NAME -1);
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.

```
int rc;
int s;
/* 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 138
- “gethostbyname() — Get a Host Entry by Name” on page 337
- “getnetbyname() — Get a Network Entry by Name” on page 348
- “getsockname() — Get Name of a Socket” on page 368
- “htons() — Translate an Unsigned Short Integer into Network Byte Order” on page 396
- “inet_addr() — Translate an Internet Address into Network Order” on page 408
- “listen() — Prepare Server for Incoming Client Requests” on page 452
- “socket() — Create a Socket” on page 791

brk() — Change Space Allocation

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

Format

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

int brk(void * addr);
```

General Description

Changes the space allocated for the calling process by setting the process' break value to `addr` and allocating the appropriate amount of space. As the break value increases, the amount of allocated space 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 brk() function allocates separate storage space from the storage space used by other memory allocation functions such as malloc() and calloc(). Because this storage space must be a contiguous segment, brk() allocates it from the initial heap segment only. Thus, it 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() call. Because this is a separate segment of storage, an application that uses other memory allocation functions can use the brk() function, provided that the user's region is large enough to support extensive use of both types of memory allocation.

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, you may want to use other memory allocation functions, such as mmap(), to assure portability with all other memory allocation functions and with any function that uses other allocation functions. However, you should not use the other memory allocation functions for applications that require brk() and should run these with an initial heap size that will satisfy the maximum storage requirements of the programs.

Note: The brk() function is not supported from a multithreaded environment; it will return in error if you invoke it in this environment.

Returned Value

If successful, brk() returns a value of `0`.

If unsuccessful, brk() returns a value of -1 and sets errno to the following value:

ENOMEM
The requested change would allocate more space than allowed for the calling process, or the caller is running in a multithreaded environment, which is not a valid environment for this function.

Related Information

- “sbrk() — Change Space Allocation” on page 677

bsd_signal() — Establish BSD Version of signal()

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

Format

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

void  (*bsd_signal (int sig, void (*func)(int)))(int);
```

General Description

Provides a partially compatible interface for programs written to use the BSD form of the signal() function.

Using BSD signal() sets the SA_RESTART flag and clears the SA_RESETHAND, whereas using ANSI signal() clears both of these flags and sets _SA_OLD_STYLE.

Three functions are available for establishing a signal's action: sigaction(), bsd_signal(), and signal(). The sigaction() function is the strategic way to establish a signal's action. The bsd_signal() and signal() functions provide compatibility with BSD and ANSI, respectively.

The *sig* argument is the signal type. The *func* argument is the signal action. You may set it to SIG_DFL, SIG_IGN or the address of a signal-catching function that takes one input argument.

For a list of the supported signal types, see Table 24 on page 750 or refer to the <signal.h> header.

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 sets errno to the following value:

EINVAL
The value of *sig* was not a valid signal type, or an attempt was made to catch a signal that cannot be caught or to ignore a signal that cannot be ignored.

Related Information

- “signal.h” on page 38
- “sigaction() — Examine or Change a Signal Action” on page 749
- “signal() — Handle Interrupts” on page 767
- “sigprocmask() — Examine or Change a Thread” on page 773

bsearch() — Search Arrays

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

< 0
Object pointed to by key is less than the array element.

= 0
Object pointed to by key is equal to the array element.

> 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>

int compare1(const void *arg1, const void *arg2);
int compare2(const void *arg1, const void *arg2);


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

```
progname Is there PATH in this sentence?
```

Output

```
The key <PATH> was found.
```

Related Information

- "stdlib.h" on page 42
- "qsort() — Sort Array" on page 622

bzero() — Zero Out Bytes in Memory

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

Format

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

void bzero(const void *s, size_t n);
```

General Description

Places *n* zero-valued bytes in the area pointed to by *s*.

Returned Value

The bzero() function has no returned value.

No errnos are defined for bzero().

Related Information

- "strings.h" on page 44
- "bcmp() — Compare Bytes in Memory" on page 91
- "bcopy() — Copy Bytes in Memory" on page 92
- "memset() — Set Buffer to Value" on page 497

calloc() — Reserve and Initialize Storage

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 keywords C++ new and delete are not interoperable with calloc(), free(), malloc(), or realloc().

Returned Value

Returns the pointer to the area of memory reserved. If not enough space is available to satisfy the request or if *num* or *size* is `0`, calloc() returns NULL. If calloc() returns NULL because there is not enough storage, it sets errno to one of the following:

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

- For information about locales and character sets, see *z/OS: XL C/C++ Programming Guide*.
- "stdlib.h" on page 42
- "free() — Free a Block of Storage" on page 278
- "malloc() — Reserve Storage Block" on page 477
- "realloc() — Change Reserved Storage Block Size" on page 637

cbrt() — Calculate Cube Root

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

Format

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

double cbrt(double x);
```

General Description

Calculates the cube root of *x*.

Note: These functions work in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

Returned Value

Returns the cube root of *x*. The `cbrt` function is always successful.

Related Information

- "math.h" on page 32

cclass() — Return Characters in a Character Class

Standards/Extensions	C or C++	Dependencies
C library	both	

Format

```
#include <collate.h>

int     cclass(char *class, collel_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 POSIX programs, the function also supports POSIX.2 character classes.

Returned Value

Returns the number of elements in the list pointed to by `list` or -1. The cclass() function 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)
{
   collel_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 30
- For information about locales and character sets, see *z/OS: XL C/C++ Programming Guide*.
- “collequiv() — Return List of Equivalent Collating Elements” on page 132
- “collorder() — Return List of Collating Elements” on page 133
- “collrange() — Calculate the Range List of Collating Elements” on page 134
- “colltostr() — Return a String for a Collating Element” on page 136
- “getmccoll() — Get Next Collating Element from String” on page 345
- “getwmccoll() — Get Next Collating Element from Wide String” on page 386
- “ismccollel() — Identify a Multicharacter Collating Element” on page 429
- “maxcoll() — Return Maximum Collating Element” on page 479
- “setlocale() — Set Locale” on page 721
- “strtocoll() — Return Collating Element for String” on page 839
- “wctype() — Obtain Handle for Character Property Classification” on page 968

cds() — Compare Double and Swap

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

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 nonstandard 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 *z/Architecture Principles of Operation (https://publibfp.dhe.ibm.com/epubs/pdf/a227832d.pdf)*.

Returned Value

Returns a value of `0` if the 8-byte value pointed to by *oldptr* is equal to the 8-byte value pointed to by *curptr*. Otherwise, it returns a value of 1.

Related Information

- *z/Architecture Principles of Operation (https://publibfp.dhe.ibm.com/epubs/pdf/a227832d.pdf)*
- "stdlib.h" on page 42
- "cs() — Compare and Swap" on page 148

cdump() — Request Main Storage Dump

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 Language Environment-formatted dump.
3. If the source file was compiled with `TEST(SYM)`, cdump() displays the contents of the user's variables. The output, which is written to `DDname CEEDUMP`, is identified with *dumptitle*.

To avoid infringing on the user's name space, this nonstandard 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 (that is, 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 successful, cdump() returns a value of `0`.

If unsuccessful, cdump() return a nonzero value.

Related Information

- "ctest.h" on page 22
- "csnap() — Request a Condensed Dump" on page 150
- "ctrace() — Request a Traceback" on page 155

ceil(), ceilf(), ceill()— Round Up to Integral Value

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

Format

```
#include <math.h>

double ceil(double x);
float ceil(float x);              /* C++ only */
long double ceil(long double x);  /* C++ only */
float ceilf(float x);
long double ceill(long double x);
```

General Description

Computes the smallest integer that is greater than or equal to *x*.

Note: These functions work in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

Returned Value

Returns the calculated value as a `double` value. If an overflow occurs, ceil() sets errno to ERANGE and returns HUGE_VAL.

Special Behavior for IEEE

The functions are always successful.

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 32
- "floor(), floorf(), floorl() — Round Down to Integral Value" on page 249

cfgetispeed() — Determine Input Baud Rate

Standards/Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	POSIX(ON)

Format

```
#define _OPEN_SYS
#include <termios.h>

speed_t cfgetispeed(const struct termios *termptr);
```

General Description

Extracts the input baud rate from the `termios` structure **termptr*, which 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 cfgetispeed to set a new baud rate in the structure and tcsetattr() to pass the changed value to the system.

Although in an OpenExtensions 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.

cfgetispeed() is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

Returns a code indicating the baud rate (see Table 8 on page 108). These codes are defined in the termios.h header file and have an unsigned integer type.

No errnos are defined for this function.

Table 8. Codes to Set Baud Rate

Code	Baud Rate
B0	Hang up
B50	50 baud
B75	75 baud

Table 8. Codes to Set Baud Rate (continued)

Code	Baud Rate
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
B38400	38,400 baud

Example

CBC3BC05

```
/* CBC3BC05
   This example determines the speed of stdin.
 */
#define _OPEN_SYS
#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 49
- "cfgetospeed() — Determine Output Baud Rate" on page 110
- "cfsetispeed() — Set Input Baud Rate in the Termios" on page 112
- "cfsetospeed() — Set Output Baud Rate in the Termios" on page 113
- "tcgetattr() — Get the Attributes for a Terminal" on page 869
- "tcsetattr() — Set the Attributes for a Terminal" on page 874

cfgetospeed() — Determine Output Baud Rate

Standards/Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	POSIX(ON)

Format

```
#define _OPEN_SYS
#include <termios.h>

speed_t cfgetospeed(const struct termios *termptr);
```

General Description

Extracts the output baud rate from the `termios` structure **termptr*, which 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 cfgetospeed() to set a new baud rate in the structure and tcsetattr() to pass the changed value to the system.

Although in an OpenExtensions 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.

cfgetospeed() is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

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 8 on page 108 shows the codes to set the baud rate.

No errnos are defined for cfgetospeed().

Example

CBC3BC06

```
/* CBC3BC06
   This example determines the speed of stdout.
 */
#define _OPEN_SYS
#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 49
- "cfgetispeed() — Determine Input Baud Rate" on page 108
- "cfsetispeed() — Set Input Baud Rate in the Termios" on page 112
- "cfsetospeed() — Set Output Baud Rate in the Termios" on page 113
- "tcgetattr() — Get the Attributes for a Terminal" on page 869
- "tcsetattr() — Set the Attributes for a Terminal" on page 874

cfsetispeed() — Set Input Baud Rate in the Termios

Standards/Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	POSIX(ON)

Format

```
#define _OPEN_SYS
#include <termios.h>

int cfsetispeed(struct termios *termptr, speed_t speed);
```

General Description

Specifies a new input baud rate for the `termios` control structure, **termptr*. The cfsetispeed() function 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 OpenExtensions application valid speeds can be set with cfsetospeed() and passed to the system with tcsetattr(), the speed has no effect on the operation of the pseudoterminal. A pseudoterminal is a special file in the /dev/ directory that effectively functions as a keyboard and a display device.

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 8 on page 108 shows the codes to set the baud rate.

Returned Value

If successful, cfsetispeed() returns a value of `0`.

If unsuccessful, cfsetispeed() returns a value of -1 and sets errno to the following:

EINVAL
An unsupported value for *speed*.

Example

CBC3BC07

```
/* CBC3BC07
   This example specifies a new input baud rate.
 */
#define _OPEN_SYS
```

```
#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 49
- "cfgetispeed() — Determine Input Baud Rate" on page 108
- "cfgetospeed() — Determine Output Baud Rate" on page 110
- "cfsetospeed() — Set Output Baud Rate in the Termios" on page 113
- "tcgetattr() — Get the Attributes for a Terminal" on page 869
- "tcsetattr() — Set the Attributes for a Terminal" on page 874

cfsetospeed() — Set Output Baud Rate in the Termios

Standards/Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	POXIS(ON)

Format

```
#define _OPEN_SYS
#include <termios.h>

int cfsetospeed(struct termios *termptr, speed_t speed);
```

General Description

Specifies a new output baud rate for the `termios` control structure, **termptr*. The cfsetospeed() function 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 OpenExtensions application valid speeds can be set with cfsetospeed() and passed to the system with tcsetattr(), the speed has no effect on the operation of the pseudoterminal. However, the operation will have an effect if issued for an OCS remote terminal. A pseudoterminal is a special file in the /dev/ directory that effectively functions as a keyboard and a display device.

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 8 on page 108 shows the codes to set the baud rate.

Returned Value

If successful, cfsetospeed() returns a value of `0`.

If unsuccessful, cfsetospeed() returns a value of -1 and sets errno to the following:

EINVAL
The value of *speed* is not supported by the hardware or software.

Example

CBC3BC08

```
/* CBC3BC08
   This example specifies a new output baud rate.
 */
#define _OPEN_SYS
#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 49
- “cfgetispeed() — Determine Input Baud Rate” on page 108
- “cfgetospeed() — Determine Output Baud Rate” on page 110
- “cfsetispeed() — Set Input Baud Rate in the Termios” on page 112
- “tcgetattr() — Get the Attributes for a Terminal” on page 869
- “tcsetattr() — Set the Attributes for a Terminal” on page 874

chaudit() — Change Audit Flags for a File by Path

Standards/Extensions	C or C++	Dependencies
OpenExtensions	both	

Format

```
#define _OPEN_SYS
#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. The chaudit() function 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 a value of `0`.

If unsuccessful, chaudit() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EACCES
The calling process does not have permission to search some component of *pathname*.

ELOOP
A loop exists in symbolic links. The chaudit() function issues this error if it detects the number of symbolic links in the resolution of *pathname* is greater than POSIX_SYMLOOP (a value defined in the limits.h header file).

ENAMETOOLONG
pathname is longer than PATH_MAX characters or a 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
No file named *pathname* exists or *pathname* 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
pathname specifies a file that is on a read-only file system.

EINVAL
option is not AUDT_USER or AUDT_AUDITOR.

Example

CBC3BC09

```
/* CBC3BC09
   This example changes the audit flags.
 */
#define _OPEN_SYS
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _OPEN_SYS
#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 46
- “access() — Determine Whether a File Can be Accessed” on page 71
- “chmod() — Change the Mode of a File or Directory” on page 117
- “chown() — Change the Owner or Group of a File or Directory” on page 120
- “fchaudit() — Change Audit Flags for a File by Descriptor” on page 201
- “stat() — Get File Information” on page 803

chdir() — Change the Working Directory

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

Format

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

int chdir(const char *pathname);
```

General Description

Makes *pathname* the new working directory.

Returned Value

If successful, chdir() changes the working directory and returns a value of 0.

If unsuccessful, chdir() does not change the working directory, returns a value of -1, and sets errno to one of the following values under the conditions described:

EACCES
The process does not have search permission on one of the components of *pathname*.

ELOOP
A loop exists in symbolic links. The chdir() function issues this error if it detects the number of symbolic links in the resolution of *pathname* is greater than POSIX_SYMLOOP (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 _OPEN_SYS
#include <unistd.h>
#undef _OPEN_SYS
#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 28
- “unistd.h” on page 51
- “closedir() — Close a Directory” on page 129
- “getcwd() — Get Path Name of the Working Directory” on page 320
- “mkdir() — Make a Directory” on page 498
- “opendir() — Open a Directory” on page 531
- “readdir() — Read an Entry from a Directory” on page 632
- “rewinddir() — Reposition a Directory Stream to the Beginning” on page 672

chmod() — Change the Mode of a File or Directory

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

Format

```
#define _OPEN_SYS
#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*.

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.

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). The chmod() function 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 a value of `0`.

If unsuccessful, chmod() returns -1 and sets errno to one of the following values under the conditions described:

EACCES
The process does not have search permission on some component of the *pathname* prefix.

ELOOP
A loop exists in symbolic links. The chmod() function issues this error if it detects the number of symbolic links in the resolution of *pathname* is greater than POSIX_SYMLOOP (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
No file named *pathname* exists or the *pathname* argument is an empty string.

ENOTDIR
Some component of the *pathname* 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
pathname 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 _OPEN_SYS
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _OPEN_SYS
#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 46
- “sys/types.h” on page 46
- “chown() — Change the Owner or Group of a File or Directory” on page 120
- “fchmod() — Change the Mode of a File or Directory by Descriptor” on page 203
- “mkdir() — Make a Directory” on page 498
- “mkfifo() — Make a FIFO Special File” on page 499
- “open() — Open a File” on page 527
- “stat() — Get File Information” on page 803

chown() — Change the Owner or Group of a File or Directory

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

Format

```
#define _OPEN_SYS
#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. The *pathname* is the name of the file whose owner or group you want to change; the *owner* is the user ID (UID) of the new owner of the file; and the *group* is the group ID (GID) 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 these is true:

1. The process has appropriate privileges.

2. The effective user ID of the process is equal to the user ID of the file owner. (The *owner* is equal to the user ID of the file owner and the *group* 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 a value of `0`.

If unsuccessful, chown() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EACCES
The process does not have search permission on some component of the *pathname* prefix.

EINVAL
owner or *group* is not a valid user ID (UID) or group ID (GID).

ELOOP
A loop exists in symbolic links. The chown() function issues this error if it detects the number of symbolic links in the resolution of *pathname* is greater than POSIX_SYMLOOP (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 can be determined using pathconf().

ENOENT
No file named *pathname* exists, or the *pathname* argument is an empty string.

ENOTDIR
Some component of the *pathname* 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
pathname is on a read-only file system.

Example

CBC3BC12

```
/* CBC3BC12
   This example changes the owner and group of a file.
 */
#define _OPEN_SYS
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#undef _OPEN_SYS
#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 28
- “unistd.h” on page 51
- “chmod() — Change the Mode of a File or Directory” on page 117
- “fchown() — Change the Owner or Group by File Descriptor” on page 204
- “fstat() — Get Status Information about a File” on page 296
- “lstat() — Get Status of File, External Link, or Symbolic Link” on page 471
- “stat() — Get File Information” on page 803

clearenv() — Clear Environment Variables

Standards/Extensions	C or C++	Dependencies
POSIX.1a Language Environment	both	

Format (POSIX - C only)

```
#include <env.h>

int clearenv(void);
```

Format (non-POSIX)

```
#include <stdlib.h>

int clearenv(void);
```

General Description

Clears all environment variables from the environment table and frees the associated storage.

The clearenv() function also resets all behavior modified by IBM XL C/C++ for z/VM 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 file, 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 (that is, 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 *z/OS: XL C/C++ Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/cbcpx01_v2r5.pdf)*.

Special Behavior for POSIX C

The clearenv() function can change the value of the pointer `environ`; therefore, a copy of that pointer made before a call to clearenv() might not be valid after the call to clearenv(). For more information about using POSIX support, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, clearenv() returns a value of `0`.

If unsuccessful, clearenv() returns a nonzero value and sets errno to ENOMEM, which indicates that the process requires more space than is available.

Example

CBC3BC13

```
/* CBC3BC13
   This IBM C for
VM/ESA example needs to be run with POSIX(ON).
   It clears the process environment variable list.
 */
#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());
  }
}
```

CBC3BC13 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
 */
#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");
}
```

CBC3BC14 Output

```
     _EDC_BYTE_SEEK = Y
     _EDC_BYTE_SEEK is undefined
```

Related Information

- For information about using environment variables, see *z/OS: XL C/C++ Programming Guide*.
- “env.h” on page 24
- “stdlib.h” on page 42
- “getenv() — Get Value of Environment Variables” on page 326
- “setenv() — Add, Delete, and Change Environment Variables” on page 708

clearerr() — Reset Error and End-of-File

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 `stream`. Generally, the indicators for a stream remain set until the program calls clearerr() or rewind().

Returned Value

The clearerr() function has no returned 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 40
- “feof() — Test End-of-File Indicator” on page 219
- “ferror() — Test for Read/Write Errors” on page 220
- “fseek() — Change File Position” on page 292
- “rewind() — Set File Position to Beginning of File” on page 671

clock() — Determine Processor Time

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, you should 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`.

Note: If you use the system() function in your program, you should 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 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`.

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 pragma in the <time.h> header is used to compile your application:

```
      #pragma map ( clock(), "@@OCLCK")
```

Because of this pragma, when your application executes, it will attempt to access an XPG4 version of clock() that 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 nor _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 version of clock(), which returns a clock_t value in historical 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 49

close() — Close a File

Standards/Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2 OpenExtensions	both	POSIX(ON)

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 *fildes* file descriptor. 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 byte file system (BFS) 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 of FIFO file is discarded.

The close() function unlocks (removes) all outstanding record locks that a process has on the associated file.

Behavior for Sockets

The 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 input data is queued, the TCP connection is reset rather than being cleanly closed.

socket
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_IUCV, AF_INET and AF_INET6 stream sockets (SOCK_STREAM) using the SO_LINGER socket option, the socket does not immediately end if data is still present when a close is issued. See "setsockopt() — Set Options Associated with a Socket" on page 734 for a description of how to establish socket options.

For AF_IUCV or AF_UNIX, the close() function will block until all pending data is written. The IUCV path associated with the connection is then severed. For AF_IUCV, this behavior can be modified by enabling the SO_LINGER socket option. For more information, see "setsockopt() — Set Options Associated with a Socket" on page 734.

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 *fildes* is closed and the calling process was previously registered to receive a SIGPOLL 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 *fildes* to be dismantled.

If O_NONBLOCK is not set and no signals have been posted for the STREAM, and data is 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 with an I_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: OpenExtensions services do not supply any STREAMS devices or pseudodevices. For more information, see "open() — Open a File" on page 527.

If *fildes* refers to the manager side of a pseudoterminal, a SIGHUP signal is sent to the process group, if any, for which the subsidiary side of the pseudoterminal is the controlling terminal.

Returned Value

If successful, close() returns a value of `0`.

If unsuccessful, close() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EAGAIN
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 the following sequence of events occurs:

1. Thread is blocked in a read() or select() call on a given file or socket descriptor.
2. Another thread issues a simultaneous close() call for the same descriptor.

EBADF
fildes 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
The close() call 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
fildes does not exist. The minor number for the file is incorrect.

EWOULDBLOCK
(Only for AF_IUCV sockets with the SO_LINGER socket option enabled.) The close completed, but linger time expired before the remaining data was sent and acknowledged. Any unsent data was discarded.

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 51
- “accept() — Accept a New Connection on a Socket” on page 69
- “creat() — Create a New File or Rewrite an Existing One” on page 146
- “dup() — Duplicate an Open File Descriptor” on page 180
- “exec Functions” on page 192
- “fclose() — Close File” on page 206
- “fork() — Create a New Process” on page 259
- “fcntl() — Control Open File Descriptors” on page 207
- “open() — Open a File” on page 527
- “pipe() — Create an Unnamed Pipe” on page 539
- “getsockopt() — Get Options Associated with a Socket” on page 370
- “setsockopt() — Set Options Associated with a Socket” on page 734
- “socket() — Create a Socket” on page 791
- “shutdown() — Shut Down All or Part of a Duplex Connection” on page 748
- “unlink() — Remove a Directory Entry” on page 910

closedir() — Close a Directory

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

Format

```
#define _OPEN_SYS
#include <dirent.h>

int closedir(DIR *dir);
```

General Description

Closes the directory pointed to by *dir*, freeing the buffer that readdir() uses when reading the directory stream.

Returned Value

If successful, closedir() returns a value of 0.

If unsuccessful, closedir() returns a value of -1 and sets errno to one of the following values under the conditions described:

EBADF
dir does not refer to an open directory stream.

EINTR
The closedir() function was interrupted by a signal. The directory may or may not be closed.

Example

CBC3BC18

```
/* CBC3BC18
   This example closes a directory.
 */
#define _OPEN_SYS
#include <dirent.h>
#include <sys/types.h>
#undef _OPEN_SYS
#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 23
- “sys/types.h” on page 46
- “opendir() — Open a Directory” on page 531
- “readdir() — Read an Entry from a Directory” on page 632
- “rewinddir() — Reposition a Directory Stream to the Beginning” on page 672

clrmemf() — Clear Memory Files

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. The clrmemf() function can remove memory files regardless of whether they are open or not.

To avoid infringing on the user's name space, this nonstandard 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 (that is, 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 *level* argument indicates which memory files are to be removed. The level can be one of the following:

__LOWER
: Removes memory files that were created in other programs and called from this program using system().

__CURRENT
: Removes only the memory files created at the current level.

__CURRENT_LOWER
: 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 PICI C Environments

Only files created by the C environment from which the clrmemf() function is called will be cleared.

Returned Value

If successful, clrmemf() returns a value of `0`.

If unsuccessful, clrmemf() returns a nonzero value.

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 40
- "fopen() — Open a File" on page 254
- "system() — Execute a Command" on page 858

collequiv() — Return List of Equivalent Collating Elements

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. The collequiv() function 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 *z/OS: XL C/C++ Programming Guide*.

Returned Value

Returns the number of collating elements found. If the value of c is not in the valid range of collating elements in the current locale, collequiv() returns a value of -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 (ismccollel(*rp))
         printf("'%s' ", colltostr(*rp));
      else if (iswprint(*rp))
         printf("'%lc' ", *rp);
      else
         printf("'%x' ", *rp);
   }
}
```

Related Information

- “collate.h” on page 20
- “cclass() — Return Characters in a Character Class” on page 104
- “collorder() — Return List of Collating Elements” on page 133
- “collrange() — Calculate the Range List of Collating Elements” on page 134
- “colltostr() — Return a String for a Collating Element” on page 136
- “getmccoll() — Get Next Collating Element from String” on page 345
- “getwmccoll() — Get Next Collating Element from Wide String” on page 386
- “ismccollel() — Identify a Multicharacter Collating Element” on page 429
- “maxcoll() — Return Maximum Collating Element” on page 479
- “strtocoll() — Return Collating Element for String” on page 839

collorder() — Return List of Collating Elements

Standards/Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <collate.h>

int      collorder(collel_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[]) {
   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 20
- “cclass() — Return Characters in a Character Class” on page 104
- “collequiv() — Return List of Equivalent Collating Elements” on page 132
- “collrange() — Calculate the Range List of Collating Elements” on page 134
- “colltostr() — Return a String for a Collating Element” on page 136
- “getmccoll() — Get Next Collating Element from String” on page 345
- “getwmccoll() — Get Next Collating Element from Wide String” on page 386
- “ismccollel() — Identify a Multicharacter Collating Element” on page 429
- “maxcoll() — Return Maximum Collating Element” on page 479
- “strtocoll() — Return Collating Element for String” on page 839

collrange() — Calculate the Range List of Collating Elements

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

Returns a value of -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 (ismccollel(*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 104
- “collequiv() — Return List of Equivalent Collating Elements” on page 132
- “collorder() — Return List of Collating Elements” on page 133
- “colltostr() — Return a String for a Collating Element” on page 136
- “getmccoll() — Get Next Collating Element from String” on page 345
- “getwmccoll() — Get Next Collating Element from Wide String” on page 386
- “ismccollel() — Identify a Multicharacter Collating Element” on page 429
- “maxcoll() — Return Maximum Collating Element” on page 479
- “strtocoll() — Return Collating Element for String” on page 839

colltostr() — Return a String for a Collating Element

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 strtocoll().

An application 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
   multicharacter 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 20
- “cclass() — Return Characters in a Character Class” on page 104
- “collequiv() — Return List of Equivalent Collating Elements” on page 132
- “collorder() — Return List of Collating Elements” on page 133
- “collrange() — Calculate the Range List of Collating Elements” on page 134
- “getmccoll() — Get Next Collating Element from String” on page 345

- "getwmccoll() — Get Next Collating Element from Wide String" on page 386
- "ismccollel() — Identify a Multicharacter Collating Element" on page 429
- "maxcoll() — Return Maximum Collating Element" on page 479
- "strtocoll() — Return Collating Element for String" on page 839

confstr() — Get Configurable Variables

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

Provides a method for applications to get configuration-defined string values. It is similar in use and purpose to the sysconf() function, which provides numeric values.

The *name* argument represents the system variable being queried. It may be any one of the following symbolic constants, defined in <unistd.h>:

_CS_PATH
Requests the value of the PATH environment variable that will find all standard utilities.

OpenExtensions uses the following constant:

_CS_SHELL
Requests the fully qualified name of the default shell.

If *len* is not `0` and *name* has a configuration-defined value, confstr() copies that value into the buffer pointed to by *buf*. 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 *len* is `0` and *buf* is a null pointer, then confstr() still returns the integer value defined below but does not return a string. If *len* is `0` but *buf* is not a null pointer, the results are unspecified.

Returned Value

If *name* has a configuration-defined value, confstr() returns the size of the buffer 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 not valid, confstr() returns a value of `0` and sets errno to EINVAL.

If *name* does not have a configuration-defined value, confstr() returns a value of `0` and leaves errno unchanged.

Related Information

- "unistd.h" on page 51
- "fpathconf() — Determine Configurable Path Name Variables" on page 261
- "pathconf() — Determine Configurable Path Name Variables" on page 533
- "sysconf() — Determine System Configuration Options" on page 856

connect() — Connect a Socket

Standards/Extensions	C or C++	Dependencies
POSIX.1 XPG4.2 OpenExtensions	both	POSIX(ON)

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`.

socket
The socket descriptor.

address
The pointer to a socket address structure containing the address of the socket to which a connection will be attempted.

address_len
The size of the socket address pointed to by *address* in bytes.

The client application uses the connect() call on a stream socket 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 a value of `-1` and sets errno 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 a value of `-1` and sets errno 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(), writev(), 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 *address* parameter is a pointer to a buffer containing the name of the peer to which the application needs to connect. The *address_len* parameter is the size, in bytes, of the buffer pointed to by *address*.

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 netinet/in.h header file.

```
struct in_addr
{
        in_addr_t s_addr;
};

 struct sockaddr_in {
     unsigned char  sin_len;       /* length of sockaddr structure */
     unsigned char  sin_family;    /* addressing family */
     unsigned short sin_port;      /* port number */
     struct in_addr sin_addr;      /* IP address */
     unsigned char  sin_zero[8];   /* unassigned */

};
```

The *sin_len* field must be set to either `0` or `sizeof(struct sockaddr_in)`. Both values are treated the same.

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_addr.s_addr* field is set to the internet address of the interface 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_INET6 Domain

If the server is in the AF_INET6 domain, the format of the name buffer is expected to be sockaddr_in6, as defined in netinet/in.h:

```
 struct sockaddr_in6 {
     uint8_t         sin6_len;       /* length of sockaddr structure */
     sa_family_t     sin6_family;    /* addressing family */
     in_port_t       sin6_port;      /* port number */
     uint32_t        sin6_flowinfo;  /* ignored */
     struct in6_addr sin6_addr;      /* IP address */
     uint32_t        sin6_scope_id;  /* scope ID */
};
```

The *sin6_len* field must be set to either `0` or `sizeof(struct sockaddr_in6)`. Both values are treated the same.

The *sin6_family* must be set to AF_INET6.

The *sin6_port* field is set to the port to which the server is bound. It must be specified in network byte order.

The *sin6_flowinfo* field is currently unsupported so its contents are ignored.

The *sin6_addr.s6_addr* field is set to the internet address of the interface to which the server is bound. It must be specified in network byte order.

The *sin6_scope_id* field identifies a set of interfaces as appropriate for the scope of the address carried in the *sin6_addr* field. For link local addresses, the *sin6_scope_id* can be used to specify the outgoing interface index. The z/VM stack supports *sin6_scope_id* for link local addresses only.

Servers in the AF_IUCV Domain

If the server is in the AF_IUCV domain, the format of the name buffer is expected to be `sockaddr_iucv`, as defined in the saiucv.h header file.

```
struct sockaddr_iucv {
    unsigned char  siucv_len;       /* length of sockaddr struct */
    unsigned char  siucv_family;    /* addressing family */
    unsigned short siucv_port;      /* port number */
    unsigned long  siucv_addr;      /* address */
    unsigned char  siucv_nodeid[8]; /* nodeid to connect to */
    unsigned char  siucv_userid[8]; /* userid to connect to */
    unsigned char  siucv_name[8];   /* iucvname for connect */
};
```

The *siucv_len* field must be set to either `0` or `sizeof(struct sockaddr_iucv)`. Both values are treated the same.

The *siucv_family* field must be set to AF_IUCV.

The *siucv_port*, *siucv_addr*, and *siucv_nodeid* fields are reserved for future use. The *siucv_port* and *siucv_addr* fields must be zeroed. The *siucv_nodeid* field must be set to exactly 8 blank characters.

The *siucv_userid* field is set to the VM user ID of the server application to which the client application is requesting a connection. This field must be 8 characters long, padded with blanks to the right. It cannot contain the null character.

The *siucv_name* field is set to the application name by which the server socket is known. The name must exactly match the 8 characters passed in the bind() call executed by the server.

If the AF_IUCV socket is not yet bound when the connect function is invoked, it will be bound to an iucvname invented by the sockets library. Use the getsockname() function to see which name was chosen.

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 sys/un.h header file.

```
 struct sockaddr_un {
    unsigned char  sun_len;        /* length of sockaddr structure */
    unsigned char  sun_family;     /* addressing family */
             char  sun_path[108];  /* pathname */
};
```

The *sun_len* field must be set to either `0` or a value greater than or equal to SUN_LEN(&*sa*), where *sa* is the name of the *sockaddr_un* variable, but less than or equal to `sizeof(struct sockaddr_un)`. The SUN_LEN() macro, which is defined in sys/un.h, evaluates to an expression that returns the total length of the used portion of the `sockaddr_un` structure, when *sun_path* has been filled in with a null-terminated file name. The length returned by SUN_LEN() does not include the terminating null character. The SUN_LEN() macro may be used to derive a value for the *address_len* argument too. If `0` is specified for *sun_len*, the *address_len* value determines how long the path name is. If *sun_len* is nonzero, the lesser of *sun_len* and *address_len* is used. In either case, if a null character appears in the string before the given length, path name is considered to end there.

The *sun_family* field must be set to AF_UNIX.

The *sun_path* field contains the null-terminated path name, and *sun_len* contains the length of the path name. Generally, the client should specify the same absolute path name on the connect() function call as the server specified on the bind() function call.

If the AF_UNIX socket is not yet bound when the connect() function is invoked, it will be bound to the null path name string (that is, the string " "). In this case, no file is created in the Byte File System.

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, connect() returns a value of `0`.

If unsuccessful, connect() returns a value of -1 and sets errno to one of the following values under the conditions described:

EADDRNOTAVAIL
The specified address is not available from the local machine.

EAFNOSUPPORT
The address family is not supported.

EALREADY
The socket descriptor *socket* is marked nonblocking, and a previous connection attempt has not completed.

EBADF
The *socket* parameter is not a valid socket descriptor.

ECONNREFUSED
The connection request was rejected by the destination host. For AF_IUCV or AF_UNIX connections, this means that the combination of IUCV directory authorizations for the server and the client were not sufficient to allow the connection to be established, or that the maximum number of connections has already been reached for the server or the client.

EFAULT
Using *address* and *address_len* would result in an attempt to copy the address into a portion of the caller's virtual machine 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 *address_len* parameter is not a valid length.

EIO
A network or a transport failure has occurred.

EISCONN
The socket descriptor *socket* is already connected.

ENETUNREACH
For an AF_INET or AF_INET6 connection, this indicates that the network cannot be reached from this host. For an AF_IUCV connection, this means that the specified user ID is not logged on or is no longer accepting connections. For an AF_UNIX connection, this means that the user ID which created the specified path name is not logged on or is no longer accepting connections.

ENOTSOCK
The descriptor refers to a file, not a socket.

EOPNOTSUPP
The *socket* parameter is not of type SOCK_STREAM.

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:

ENOTDIR
A component of the path prefix of the path name in *address* is not a directory.

ENAMETOOLONG
A component of a path name exceeded NAME_MAX characters, or an entire path name exceeded PATH_MAX characters.

EACCES
The process does not have search permission on a component of the path prefix, or it does not have write access to the named socket.

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 path name in *address*.

ENOENT
A component of the path name does not name an existing file or the path name is an empty string.

Examples

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 requested name 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 92.

AF_INET Domain Example

The following example illustrates the connect() call to connect to a server in the AF_INET domain.

```
int s;
struct sockaddr_in inet_server;
int rc;
int connect(int s, struct sockaddr *name, int namelen);

/* Connect to server bound to a specific interface in the
internet domain */
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));
```

AF_INET6 Domain Example

The following example illustrates the connect() call to connect to a server in the AF_INET6 domain.

```
int                  s;  /* Socket descriptor. */
int                  rc;
char                 v6addr[16] = "C2C1::129:5:24:1";  /* IPv6 IP address. */
struct sockaddr_in6  inet6_server;

/* Connect to server bound to specific interface in the AF_INET6 domain. */
/* (Not specifying flow info and scope ID.)                              */
memset(&inet6_server,0,sizeof(inet6_server));
myname.sin6_len = sizeof(inet6_server);
myname.sin6_family = AF_INET6;
myname.sin6_addr = inet_pton(AF_INET6,(char *)&v6addr,&(inet6_server.sin6_addr));
myname.sin6_port = htons(1024);
rc = connect(s,struct sockaddr *)&inet6_server,sizeof(inet6_server));
```

AF_IUCV Domain Example

The following example illustrates the connect() call to connect to a server in the AF_IUCV domain.

```
int s;
struct sockaddr_iucv servername;
int rc;
```

```
%define SERVER_USERID "VMUSER1"  /* Server's VM user ID */
%define SERVER_NAME "APPL1"  /* Server's application name */

/* Connect to a server bound to a name in the IUCV domain */
memset(&servername, 0, sizeof servername);
servername.siucv_family = AF_IUCV;
memset(servername.siucv_nodeid, ' ', sizeof servername.siucv_nodeid);
memset(servername.siucv_userid, ' ', sizeof servername.siucv_userid);
memset(servername.siucv_name, ' ', sizeof servername.siucv_name);
memcpy(servername.siucv_userid, SERVER_USERID, sizeof SERVER_USERID -1);
memcpy(servername.siucv_name, SERVER_NAME, sizeof SERVER_NAME -1);
rc = connect(s, (struct sockaddr *) &servername, sizeof (servername);
```

AF_UNIX Domain Example

The following example illustrates the connect() call to connect to a server in the AF_UNIX domain.

```
int s;
struct sockaddr_un servername;
int rc;
#define SERVER_PATH "/tmp/testserver"

servername.sun_len = 0;  /* Library will use connect() argument instead */
servername.sun_family = AF_UNIX;

memcpy(servername.sun_path, SERVER_PATH, sizeof SERVER_PATH);
rc = connect(s, (struct sockaddr *) &servername, SUN_LEN(&servername();
```

Related Information

- “accept() — Accept a New Connection on a Socket” on page 69
- “bind() — Bind a Name to a Socket” on page 92
- “getsockname() — Get Name of a Socket” on page 368
- “htons() — Translate an Unsigned Short Integer into Network Byte Order” on page 396
- “inet_addr() — Translate an Internet Address into Network Order” on page 408
- “listen() — Prepare Server for Incoming Client Requests” on page 452
- “select() — Monitor Activity on Files/Sockets and Message Queues” on page 680
- “selectex() — Monitor Activity on Files/Sockets and Message Queues” on page 685
- “socket() — Create a Socket” on page 791

cos(), cosf(), cosl()— Calculate Cosine

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

Format

```
#include <math.h>

double cos(double x);
float cos(float x);                /* C++ only */
long double cos(long double x);    /* C++ only */
float cosf(float x);
long double cosl(long double x);
```

General Description

Calculates the cosine of *x*, expressed in radians.

Note: These functions work in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 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 a value of `0` and sets errno to ERANGE. If the correct value would cause an underflow, cos() returns `0` and the value of the ERANGE macro 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 32
- "acos(), acosf(), acosl() — Calculate Arccosine" on page 73
- "asin(), asinf(), asinl() — Calculate Arcsine" on page 79
- "atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate Arctangent" on page 83
- "cosh(), coshf(), coshl()— Calculate Hyperbolic Cosine" on page 145
- "sin() — Calculate Sine" on page 785
- "sinh() — Calculate Hyperbolic Sine" on page 786
- "tan() — Calculate Tangent" on page 861
- "tanh() — Calculate Hyperbolic Tangent" on page 862

cosh(), coshf(), coshl()— Calculate Hyperbolic Cosine

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

Format

```
#include <math.h>

double cosh(double x);
float cosh(float x);                  /* C++ only */
long double cosh(long double x);      /* C++ only */
float coshf(float x);
long double coshl(long double x);
```

General Description

Calculates the hyperbolic cosine of *x*, expressed in radians.

Note: These functions work in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

Returned Value

If the result is too large, 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 32
- "acos(), acosf(), acosl() — Calculate Arccosine" on page 73
- "asin(), asinf(), asinl() — Calculate Arcsine" on page 79

- “atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate Arctangent” on page 83
- “cos(), cosf(), cosl()— Calculate Cosine” on page 143
- “sin() — Calculate Sine” on page 785
- “sinh() — Calculate Hyperbolic Sine” on page 786
- “tan() — Calculate Tangent” on page 861
- “tanh() — Calculate Hyperbolic Tangent” on page 862

creat() — Create a New File or Rewrite an Existing One

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

Format

```
#define _OPEN_SYS
#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. For further information, see “open() — Open a File” on page 527.

The *mode* argument specifies the file permission bits for creating the file. You can use the following symbols 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 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.

Returned Value

If successful, creat() returns a file descriptor for the open file.

If unsuccessful, creat() returns a value of -1 and sets errno to one or the following values under the conditions described:

EACCES
A file descriptor was not created for one of these reasons:

- 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
The open() function 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. The creat() function issues this error if it detects the number of symbolic links detected in the resolution of *pathname* is greater than POSIX_SYMLOOP.

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 _OPEN_SYS
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#undef _OPEN_SYS
#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 25
- “sys/stat.h” on page 46
- “sys/types.h” on page 46
- “close() — Close a File” on page 127
- “open() — Open a File” on page 527
- “unlink() — Remove a Directory Entry” on page 910

cs() — Compare and Swap

Standards/Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdlib.>

int cs(cs_t *oldptr, cs_t *curptr, cs_t newword);
```

General Description

A built-in function that 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 nonstandard 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 *z/Architecture Principles of Operation (https://publibfp.dhe.ibm.com/epubs/pdf/a227832d.pdf)*.

Returned Value

Returns a value of `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, cs() returns a value of 1.

Related Information

- *z/Architecture Principles of Operation (https://publibfp.dhe.ibm.com/epubs/pdf/a227832d.pdf)*
- "stdlib.h" on page 42
- "cds() — Compare Double and Swap" on page 106

csid() — Character Set ID for Multibyte Character

Standards/Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdlib.h>

int csid(const char *c)
```

General Description

Determines the character set identifier for the specified multibyte character, *c*, that begins in the initial shift state.

To avoid infringing on the user's name space, this nonstandard 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 (that is, 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 a value of -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 42

csnap() — Request a Condensed Dump

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. It also displays other environmental control blocks required by IBM Service. Under Language Environment, these consist of the Common Anchor Area (CAA) and the IBM XL C/C++ for z/VM CAA information. The output, which is written to `DDname CEEDUMP`, is identified with *dumptitle*.

To avoid infringing on the user's name space, this nonstandard 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 (that is, 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 successful, csnap() returns a value of `0`.

If unsuccessful, csnap() returns a nonzero value.

Example

```
#include <ctest.h>

int main(void) {

   int rc;
   rc = csnap("Sample csnap output");
}
```

Related Information

- "ctest.h" on page 22

- "cdump() — Request Main Storage Dump" on page 106
- "ctrace() — Request a Traceback" on page 155

__csplist() — Retrieve CSP Parameters

Standards/Extensions	C or C++	Dependencies
Language Environment	C only	

Format

```
#include <csp.h>
__csplist;
```

General Description

A macro used to access the parameter list passed from Cross System Product (CSP) to your 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 program.

`argc` will always be 1. For information about the PLIST compiler option, see the *z/OS: XL C/C++ User's Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/cbcux01_v2r5.pdf)*.

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 21

ctermid() — Generate Path Name for Controlling Terminal

Standards/Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	POSIX(ON)

Format

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

char *ctermid(char *string);
```

General Description

Returns a string argument that can be used as a path name to refer to the controlling terminal for the current process. The *string* argument 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.

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 returned path name can be used to access the controlling terminal, if the process has a controlling terminal.

Returned Value

The ctermid() function is always successful.

No errnos are defined for ctermid().

Example

CBC3BC32

```
/* CBC3BC32
   This example refers to the controlling terminal for the current process.
 */
#define _OPEN_SYS
#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 40
- “unistd.h” on page 51
- “ttyname() — Get the Name of a Terminal” on page 897

ctest() — Start Debug Tool

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. The parameter *command* is a character pointer to a list of valid Debug Tool commands that ctest() uses to invoke the 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, 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 (that is, 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 more information on the Debug Tool, see *Debug Tool User's Guide and Reference*.

Returned Value

If successful, ctest() returns a value of `0`.

If unsuccessful, ctest() returns a nonzero value.

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 22

ctime() — Convert Time to a Character String

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))`

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 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 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 (if time() is called from a non-POSIX program, or if TZ is not defined).

  For more information about using POSIX support, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

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 30

- “time.h” on page 49
- “asctime() — Convert Time to Character String” on page 78
- “gmtime() — Convert Time to Broken-Down UTC Time” on page 391
- “localtime() — Convert Time and Correct for Local Time” on page 456
- “mktime() — Convert Local Time” on page 505
- “strftime() — Convert to Formatted Time” on page 821
- “time() — Determine Current Time” on page 885

ctrace() — Request a Traceback

Standards/Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <ctest.h>

int ctrace(char *dumptitle);
```

General Description

Requests a traceback. The output, which is written to `DD:CEEDUMP`, is identified with *dumptitle*. ctrace() invokes the `CEE3DMP` Language Environment callable service with the following options: `TRACEBACK, NOFILE, NOBLOCK, NOVARIABLE, NOSTORAGE, STACKFRAME(ALL), NOCOND, NOENTRY`.

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 nonstandard 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 (that is, 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

If successful, ctrace() returns a value of `0`.

If unsuccessful, ctrace() returns a nonzero value.

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              Language Environment for MVS & VM
06/16/95 6:12:47 PMPage:    1

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  CEEBBEXT      000079D8  +0000013C  CEEBBEXT      000079D8  +0000013C              Call
```

Output for C

```
CEE3DMP: Sample ctrace output Language Environment for MVS & VM
06/16/95 6:12:47 PMPage:    1

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  CEEBBEXT      000079D8  +0000013C  CEEBBEXT      000079D8  +0000013C              Call
```

Related Information

- *z/OS: Language Environment Programming Reference (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea300_v2r5.pdf)*
- “ctest.h” on page 22
- “cdump() — Request Main Storage Dump” on page 106
- “csnap() — Request a Condensed Dump” on page 150

cuserid() — Return Character Login of the User

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

Format

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

char *cuserid(char *s);
```

General Description

Generates a character representation of the name associated with the real or effective user ID of the process.

If s is a null pointer, cuserid() generates this representation in an area that subsequent calls to cuserid() may overwrite. A pointer to the area is returned.

If *s* is not a null pointer, cuserid() assumes *s* points to an array of at least {`L_cuserid`} bytes and deposits the representation in this array. The symbolic constant {`L_cuserid`} is defined in `<stdio.h>` and has a value greater than `0`.

Returned Value

Returns one of the following values:

- 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, cuserid() returns the address of a buffer local to the calling thread containing the login name.
- If *s* is not a null pointer, cuserid() returns *s*.
- If *s* is not a null pointer and the login name cannot be found, cuserid() places the null byte `'\0'` at **s*.

Related Information

- "stdlib.h" on page 42
- "geteuid() — Get the Effective User ID" on page 327
- "getlogin() — Get the User Login Name" on page 344
- "getpwnam() — Access the User Database by User Name" on page 361
- "getpwuid() — Access the User Database by User ID" on page 362
- "getuid() — Get the Real User ID" on page 378

dbm_clearerr() — Clear Database Error Indicator

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

Clears the error condition of the database, allowing dbm_ operations to proceed. The *db* argument is a database handle returned by a previous call to dbm_open().

Note: This function does not correct any database problems due to previous failures. The database may be in an inconsistent or damaged state.

Special Behavior for OpenExtensions Services

In a multithreaded environment, the database error indicator is global to all threads using the database handle. Thus, clearing the database error indicator affects all threads using the database handle.

Returned Value

The returned value is unspecified by X/Open.

If successful, dbm_clearerr() returns a value of `0`.

If unsuccessful, dbm_clearerr() returns a value of -1 and sets errno to the following value:

EINVAL
An invalid database descriptor was specified.

Related Information

- “ndbm.h” on page 34
- “dbm_close() — Close a Database” on page 158
- “dbm_delete() — Delete Database Record” on page 158
- “dbm_error() — Check Database Error Indicator” on page 159
- “dbm_fetch() — Get Database Content” on page 160
- “dbm_firstkey() — Get First Key in Database” on page 161
- “dbm_nextkey() — Get Next Key in Database” on page 162
- “dbm_open() — Open a Database” on page 163
- “dbm_store() — Store Database Record” on page 164

dbm_close() — Close a Database

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

Format

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

void dbm_close(DBM * db);
```

General Description

Closes a database. The *db* argument is a database handle returned by a previous call to dbm_open().

Special Behavior for OpenExtensions Services

A dbm_close() function call removes access to the database for all threads that use the database handle.

Returned Value

The dbm_close() function has no returned value.

Related Information

- “ndbm.h” on page 34
- “dbm_clearerr() — Clear Database Error Indicator” on page 157
- “dbm_delete() — Delete Database Record” on page 158
- “dbm_error() — Check Database Error Indicator” on page 159
- “dbm_fetch() — Get Database Content” on page 160
- “dbm_firstkey() — Get First Key in Database” on page 161
- “dbm_nextkey() — Get Next Key in Database” on page 162
- “dbm_open() — Open a Database” on page 163
- “dbm_store() — Store Database Record” on page 164

dbm_delete() — Delete Database Record

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

Format

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

int dbm_delete(DBM *db, datum key);
```

General Description

Deletes a record and its key from the database. The *db* argument is the handle to a database 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`.

Note: After calling dbm_delete() during a pass through the keys by dbm_firstkey() and dbm_nextkey(), an application must call dbm_firstkey() to reset positioning. Otherwise, unpredictable results may occur, for example, the same key may return multiple times or not at all.

The dbm_delete() operation does not physically reclaim file space or reduce file size. The space is available for reuse, however, subject to hashing.

Special Behavior for OpenExtensions Services

In a multithreaded environment, changes made to the database by a dbm_delete() operation affect all threads using the database handle. Thus, all other threads must also call dbm_firstkey() to reset their positioning prior to using dbm_nextkey(). The dbm_delete() operation does not affect buffer pointers to previous data for a prior dbm_fetch() operation by another thread for the same *key*. 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 value of `0`.

If unsuccessful, dbm_delete() returns a value of -1 and sets the error value in errno. It may also set the database error indicator.

Related Information

- “ndbm.h” on page 34
- “dbm_clearerr() — Clear Database Error Indicator” on page 157
- “dbm_close() — Close a Database” on page 158
- “dbm_error() — Check Database Error Indicator” on page 159
- “dbm_fetch() — Get Database Content” on page 160
- “dbm_firstkey() — Get First Key in Database” on page 161
- “dbm_nextkey() — Get Next Key in Database” on page 162
- “dbm_open() — Open a Database” on page 163
- “dbm_store() — Store Database Record” on page 164

dbm_error() — Check Database Error Indicator

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

Returns the error condition of the database. The *db* argument is a database handle returned by a previous call to dbm_open().

Special Behavior for OpenExtensions Services

In a multithreaded environment, the database error indicator is global to all threads using the database handle. Thus, another thread may perform a database operation that results in setting the database error indicator.

Returned Value

Returns a value of zero if the error condition is not set and a nonzero value if the error condition is set.

Related Information

- "ndbm.h" on page 34
- "dbm_clearerr() — Clear Database Error Indicator" on page 157
- "dbm_close() — Close a Database" on page 158
- "dbm_delete() — Delete Database Record" on page 158
- "dbm_fetch() — Get Database Content" on page 160
- "dbm_firstkey() — Get First Key in Database" on page 161
- "dbm_nextkey() — Get Next Key in Database" on page 162
- "dbm_open() — Open a Database" on page 163
- "dbm_store() — Store Database Record" on page 164

dbm_fetch() — Get Database Content

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

Format

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

datum dbm_fetch(DBM * db, datum key);
```

General Description

Reads a record from the database. The *db* argument is a database handle returned by a previous call to dbm_open(). The *key* argument is a datum that the application has initialized to the value matching 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 character pointer to an array of data that is `dsize` bytes in length. This 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 OpenExtensions Services

In a multithreaded 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 thread operations on the database, with the exception of a dbm_close() operation, which invalidates the *datum*.

Returned Value

If successful, dbm_fetch() returns 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 sets the error value in errno. It may also set the database error indicator.

Related Information

- "ndbm.h" on page 34
- "dbm_clearerr() — Clear Database Error Indicator" on page 157
- "dbm_close() — Close a Database" on page 158
- "dbm_delete() — Delete Database Record" on page 158
- "dbm_error() — Check Database Error Indicator" on page 159
- "dbm_firstkey() — Get First Key in Database" on page 161
- "dbm_nextkey() — Get Next Key in Database" on page 162
- "dbm_open() — Open a Database" on page 163
- "dbm_store() — Store Database Record" on page 164

dbm_firstkey() — Get First Key in Database

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

Format

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

datum dbm_firstkey(DBM * db);
```

General Description

Returns the first key in the database. The *db* argument is a database handle returned by a previous call to dbm_open(). Since the keys are arbitrary binary data, their order of return by dbm_firstkey() and dbm_nextkey() does not reflect any lexical ordering or record insertion. You can retrieve all keys from the database by resetting positioning to the beginning using dbm_firstkey(), then looping using dbm_nextkey() function calls until a null `dptr` is returned in the *datum*.

An example of looping is the following:

```
for (key = dbm_firstkey(db); key.dptr !=NULL; key = dbm_nextkey(db))
```

The returned `dptr` is valid only until the next `dbm_` operation by this thread.

Special Behavior for OpenExtensions Services

In a multithreaded 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, if other threads have modified the database, for example, by using dbm_store() or dbm_delete(), unpredictable results may occur for threads executing dbm_nextkey(). For example, the same key may return multiple times or not at all. The application must call dbm_firstkey() to reset positioning to the beginning if another thread has modified the database.

Returned Value

If successful, dbm_firstkey() returns the `datum` containing a pointer to the key `dptr` and the key length `dsize`.

If unsuccessful, dbm_firstkey() returns a null pointer in `dptr` and sets the error value in errno. It may also set the database error indicator.

Related Information

- "ndbm.h" on page 34
- "dbm_clearerr() — Clear Database Error Indicator" on page 157
- "dbm_close() — Close a Database" on page 158
- "dbm_delete() — Delete Database Record" on page 158
- "dbm_error() — Check Database Error Indicator" on page 159
- "dbm_fetch() — Get Database Content" on page 160
- "dbm_nextkey() — Get Next Key in Database" on page 162
- "dbm_open() — Open a Database" on page 163
- "dbm_store() — Store Database Record" on page 164

dbm_nextkey() — Get Next Key in Database

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

Format

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

datum dbm_nextkey(DBM * db);
```

General Description

Returns the next key in the database. The *db* argument is a database handle returned by a previous call to dbm_open(). Since the keys are arbitrary binary data, the order of return by dbm_firstkey() and dbm_nextkey() does not reflect any lexical ordering or record insertion. You can retrieve all keys from the database by resetting positioning to the beginning using dbm_firstkey(), then looping using dbm_nextkey() function calls until a null `dptr` is returned in the *datum*.

An example of looping is the following:

```
for (key = dbm_firstkey(db); key.dptr !=NULL; key = dbm_nextkey(db))
```

The returned `dptr` is valid only until the next `dbm_` operation by this thread.

Special Behavior for OpenExtensions Services

In a multithreaded 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_firstkey() and dbm_nextkey() operations. However, if other threads have modified the database, for example, by using dbm_store() or dbm_delete(), unpredictable results may occur for threads executing dbm_nextkey(). For example, the same key may return multiple times or not at all. The application must call dbm_firstkey() to reset positioning to the beginning prior to executing dbm_nextkey() if another thread has modified the database.

Returned Value

If successful, dbm_nextkey() returns the `datum` containing a pointer to the key `dptr` and the key length `dsize`.

If unsuccessful, dbm_nextkey() returns a null pointer in `dptr` and sets the error value in errno. It may also set the database error indicator.

Related Information

- "ndbm.h" on page 34
- "dbm_clearerr() — Clear Database Error Indicator" on page 157
- "dbm_close() — Close a Database" on page 158
- "dbm_delete() — Delete Database Record" on page 158
- "dbm_error() — Check Database Error Indicator" on page 159
- "dbm_fetch() — Get Database Content" on page 160
- "dbm_firstkey() — Get First Key in Database" on page 161
- "dbm_open() — Open a Database" on page 163
- "dbm_store() — Store Database Record" on page 164

dbm_open() — Open a Database

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

Format

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

DBM *dbm_open(const char *file, int open_flags, mode_t file_mode);
```

General Description

Opens a database. The *file* argument is the path name of the database, not including the file name 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 the database can store 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 OpenExtensions Services

In a multithreaded environment, the dbm_ functions have both POSIX process-wide and thread-specific characteristics. OpenExtensions Services provide the following multithreaded 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 key's 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 is modified, all threads should call dbm_firstkey() to reset positioning using 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 are permitted for read access only by a single thread, multiple threads within a process, or multiple processes. No protection is provided for database modification, which can cause unpredictable results, including database destruction.

Returned Value

If successful, dbm_open() returns a pointer to the database descriptor.

If unsuccessful, dbm_open() returns a null pointer and sets the error value in errno.

Related Information

- “ndbm.h” on page 34
- “dbm_clearerr() — Clear Database Error Indicator” on page 157
- “dbm_close() — Close a Database” on page 158
- “dbm_delete() — Delete Database Record” on page 158
- “dbm_error() — Check Database Error Indicator” on page 159
- “dbm_fetch() — Get Database Content” on page 160
- “dbm_firstkey() — Get First Key in Database” on page 161
- “dbm_nextkey() — Get Next Key in Database” on page 162
- “dbm_store() — Store Database Record” on page 164
- “open() — Open a File” on page 527

dbm_store() — Store Database Record

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

Format

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

int dbm_store(DBM * db, datum key, datum content, int store_mode);
```

General Description

Writes a record to a database. The *db* argument is a 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 *content* argument is a datum that describes the data record being stored. The *content* datum contains a `dptr` pointer to the data and the data length in `dsize`.

The *store_mode* argument controls whether dbm_store() replaces an 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 *key* and *content* pair if the *key* already exists in the database. If the *key* does not already exist, add the new *key* and *content* pair.

DBM_REPLACE
Replace the *key* and *content* pair in the database with the new pair if the *key* already exists. If the *key* does not already exist, add the new *key* and *content* pair.

Note: After calling dbm_store(), during a pass through the keys by dbm_firstkey() and dbm_nextkey(), the application must call dbm_firstkey() to reset positioning. Otherwise, unpredictable results may occur, for example, the same key may return multiple times or not at all.

The number of records that the database can store 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 OpenExtensions Services

In a multithreaded environment, changes made to the database by a dbm_store() operation affect all threads using the database handle. Thus, all other threads must also call dbm_firstkey() to reset their positioning prior to executing dbm_nextkey(). The dbm_store() operation does not affect buffer pointers to previous data for a prior dbm_fetch() operation by another thread for the same *key*. 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 value of `0`. If you specify `DBM_INSERT` and the *key* already exists, dbm_store() returns a value of 1.

If unsuccessful, dbm_store() returns a value of -1 and sets errno to one of the following values under the conditions described. It may also set the database error indicator.

EFBIG
The seek/write operation failed, attempting to write a 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
The *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 to be less than or equal to 1012 bytes.

Related Information

- “ndbm.h” on page 34
- “dbm_clearerr() — Clear Database Error Indicator” on page 157
- “dbm_close() — Close a Database” on page 158
- “dbm_delete() — Delete Database Record” on page 158
- “dbm_error() — Check Database Error Indicator” on page 159
- “dbm_fetch() — Get Database Content” on page 160
- “dbm_firstkey() — Get First Key in Database” on page 161
- “dbm_nextkey() — Get Next Key in Database” on page 162
- “dbm_open() — Open a Database” on page 163

difftime() — Compute Time Difference

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 49

- “asctime() — Convert Time to Character String” on page 78
- “ctime() — Convert Time to a Character String” on page 153
- “gmtime() — Convert Time to Broken-Down UTC Time” on page 391
- “localtime() — Convert Time and Correct for Local Time” on page 456
- “mktime() — Convert Local Time” on page 505
- “strftime() — Convert to Formatted Time” on page 821
- “time() — Determine Current Time” on page 885

dirname() — Report the Parent Directory of a Path Name

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

Format

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

char *dirname(char *path);
```

General Description

Takes the path name, *path*, and returns a pointer to the path name of the parent directory of that file. Trailing '/' characters in the path are not counted as part of the path.

The dirname() function may modify the string pointed to by *path*.

Examples:

```
Input String       Output String
"/usr/lib"         "/usr"
"/usr/"            "/"
"usr"              "."
"/"                "/"
"."                "."
".."               "."
```

Returned Value

Returns a pointer to a string that is the parent directory of *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 ".".

No errnos are defined for dirname().

Related Information

- “libgen.h” on page 28
- “basename() — Return the Last Component of a Path Name” on page 91

div() — Calculate Quotient and Remainder

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

Format

```
#include <stdlib.h>

div_t div(int numerator, int denominator);
div_t div(long numerator, long denominator);  /* C++ only */
```

General Description

Calculates the quotient and remainder of the division of *numerator* by *denominator*.

Special Behavior for C++

For C++ applications, div() is also overloaded for the type long.

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 42
- "ldiv() — Compute Quotient and Remainder of Integral Division" on page 446

dlclose() — Close a dlopen() object

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R6

Format

```
#define _UNIX03_SOURCE
#include <dlfcn.h>

void  dlclose(void *handle);
```

General Description

Informs the system that the Dynamic Link Library (DLL) referenced by a *handle* returned from a previous dlopen() invocation is no longer needed by the application. Once a DLL has been closed, an application should assume that its symbols and the symbols of any dependent DLLs are no longer available to dlsym().

Returned Value

NULL is returned if the referenced DLL was successfully closed. If the DLL could not be closed, or if *handle* does not refer to an open DLL, a non-zero value will be returned.

Application Usage

1. A conforming application should use a *handle* returned from a dlopen() invocation only within a given scope, bracketed by the dlopen() and dlclose() operations. The value of a *handle* must be treated as an opaque object by the application, used only in calls to dlsym() and dlclose().
2. DLLs that are loaded explicitly, that is with dlopen(), and are not freed with a corresponding call to dlclose(), are freed automatically at enclave termination in LIFO sequence.
3. Non-local C++ static destructors defined in a DLL are executed only once, when the DLL program object is deleted from memory.
4. More detailed diagnostic information is available through dlerror() ,the _EDC_DLL_DIAG environment variable, and the Language Environment DLL Failure control block (CEEDLLF) chain.
5. This function is not available under SPC, MTF and CSP environments.

Example

The following example illustrates use of dlopen() and dlclose():

```
...
/* Open a dynamic library and then close it ... */

#include <dlfcn.h>

void *mylib;
int eret;

mylib = dlopen("mylib.so", RTLD_LOCAL | RTLD_LAZY);
...
eret = dlclose(mylib);
...
```

Related Information

- “dlerror() — Get Diagnostic Information” on page 169
- “dlopen() — Gain Access to a Dynamic Link Library (DLL)” on page 170
- “dlsym() — Obtain the Address of a Symbol from a dlopen() Object” on page 172

dlerror() — Get Diagnostic Information

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R6

Format

```
#define _UNIX03_SOURCE
#include <dlfcn.h>

void  dlerror(void);
```

General Description

Returns a null-terminated character string (with no trailing <newline>) that describes the last error that occurred while processing a DLL by dlopen(), dlsym(), or dlclose(). NULL is returned if no errors have occurred since the last invocation of dlerror().

Note: dlerror() is thread safe, so the information returned describes the last error that occurred on that thread

Returned Value

A null-terminated character string is returned if successful, otherwise NULL is returned.

Application Usage

1. Messages returned by dlerror() reside in a static buffer that is overwritten on each new call to dlerror() on that thread.
2. Application code should not write to this buffer.
3. Programs wishing to preserve an error message should make their own copies of that message.
4. This function is not available under SPC, MTF and CSP environments.

Example

The following example prints out the last dynamic linking error:

```
...
#include <dlfcn.h>

char *errstr;

errstr = dlerror();
if (errstr != NULL)
printf ("A dynamic linking error occurred: (%s)\n", errstr);
...
```

Related Information

- "dlclose() — Close a dlopen() object" on page 168
- "dlopen() — Gain Access to a Dynamic Link Library (DLL)" on page 170
- "dlsym() — Obtain the Address of a Symbol from a dlopen() Object" on page 172

dlopen() — Gain Access to a Dynamic Link Library (DLL)

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R6

Format

```
#define _UNIX03_SOURCE
#include <dlfcn.h>

void  *dlopen(const char *file, int mode);
```

General Description

Makes the Dynamic Link Library (DLL) specified by *file* available to the calling program.

If the *file* argument contains a single slash ("/"), it is used as the UNIX file system pathname for the DLL. If the environment variable LIBPATH is set, each directory listed will be searched for the DLL. Otherwise, the current directory will be searched.

Note: Searching for a DLL in the UNIX file system is case-sensitive.

If the *file* argument begins with two slashes ("//"), then an attempt is made to load the DLL from the caller's CMS search order. The DLL name must be eight characters or less, and is converted to uppercase.

If the *file* argument doesn't begin with one or two slashes ("/" or "//"), and doesn't contain a single slash ("/") anywhere in the name, then it is ambiguous as to where the DLL resides.

- If the POSIX(ON) runtime option is specified, then the UNIX file system is searched first for the DLL, and if not found, the accessed CMS search order is searched.
- If the POSIX(OFF) runtime option is specified, then the accessed CMS search order is searched first for the DLL, and if not found, the UNIX file system is searched.

For more details on how DLLs are located, see the "Loading DLLs" topic in *z/OS: Language Environment Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf)*.

A successful call returns a handle which the caller may use on subsequent calls to dlsym() and dlclose(). The value of this handle should not be interpreted in any way by the caller.

Only a single copy of a DLL is brought into the address space, even if invoked multiple times for the same DLL, and even if different values of the *file* parameter are used to reference the same DLL.

The *mode* parameter describes how dlopen() operates on a file with respect to the processing of dependent DLLs and the scope of visibility of the symbols provided within *file*. If a *file* is specified in multiple invocations, *mode* is interpreted at each invocation. The *mode* is a bitwise-OR of the values specified.

Mode Values

When a DLL is loaded, it may contain implicit references to symbols in another "dependent" DLL, whose addresses are not known until that DLL is loaded. These implicit references must be relocated before the symbols can be accessed, which means loading the DLL containing the references. The *mode* parameter governs when these relocations (and loads) take place and may have the following values:

Value
Description

RTLD_LAZY
When possible, the loading of dependent DLLs, and resolution of symbols contained therein, may be deferred until the first reference to one of those symbols. This is the default behavior.

Note: Once *RTLD_NOW* has been specified, all relocations will have been completed causing additional *RTLD_NOW* operations to be redundant and any further *RTLD_LAZY* operations irrelevant.

RTLD_NOW
Load all dependent DLLs for the DLL being loaded and resolve all symbols before returning. This may include zero or more levels of nested dependent DLLs, all of which are loaded at this time.

RTLD_GLOBAL
Allows symbols in the DLL being loaded to be visible when resolving symbols through the global symbol object that was opened with dlopen(*NULL*,*0*). All dependent DLLs are always implicitly loaded as if RTLD_GLOBAL had been specified. This is the default behavior.

RTLD_LOCAL
Prevents symbols in the DLL being loaded to be visible when resolving symbols through the global symbol object that was opened with dlopen(*NULL*,*0*). All dependent DLLs of this DLL continue to be implicitly loaded as if RTLD_GLOBAL had been specified.

If a subsequent call is made for this same DLL with a *mode* of *RTLD_GLOBAL*, then the DLL will maintain the *RTLD_GLOBAL* status regardless of any previous or future specification of *RTLD_LOCAL*, as long as the DLL remains loaded (see dlclose()).

If the value of *file* is NULL, dlopen() returns a "global symbol object" *handle*. This object will provide access (via dlsym()) to the symbols exported from:

- The main application, and dependent DLLs for the main application that were loaded at program start-up, and
- The set of DLLs loaded using dlopen() with the *RTLD_GLOBAL* flag. This set of DLLs can change dynamically as other DLLs are opened and closed.

Symbols introduced by the call to dlopen() for a DLL, and available through dlsym(), are those which are exported by the DLL. Typically such symbols will be those identified by a #pragma export in C, or with the EXPORTALL compile option. For details on how to specify exported data and functions, for C/C++ as well as other languages, see the "Building a Simple DLL" topic in *z/OS: Language Environment Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf)*.

Returned Value

NULL is returned if:

- *file* cannot be found or opened for reading
- *file* is not in correct DLL executable format
- an error occurred during the process of loading *file*, or relocating its symbolic references
-

Application Usage

1. For details about how to create and use DLLs, see *z/OS: Language Environment Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf)*.
2. The AMODE of the application must be the same as the AMODE of the DLL.
3. Non-local C++ static constructors defined in a DLL are executed only once, when the DLL program object is physically loaded into memory.
4. More detailed diagnostic information is available through dlerror(), the _EDC_DLL_DIAG environment variable, and the Language Environment DLL Failure control block (CEEDLLF) chain.
5. This function is not available under SPC, MTF and CSP environments.

Example

The following example illustrates use of dlopen() and dlclose():

```
...
/* Open a dynamic library and then close it ... */

#include <dlfcn.h>

void *mylib;
int eret;

mylib = dlopen("mylib.so", RTLD_LOCAL | RTLD_LAZY);
...
eret = dlclose(mylib);
...
```

Related Information

- “dlclose() — Close a dlopen() object” on page 168
- “dlerror() — Get Diagnostic Information” on page 169
- “dlsym() — Obtain the Address of a Symbol from a dlopen() Object” on page 172

dlsym() — Obtain the Address of a Symbol from a dlopen() Object

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 3	both	z/OS V1R6

Format

```
#define _UNIX03_SOURCE
#include <dlfcn.h>

void  *dlsym(void *__restrict__ handle, const char *__restrict__ name);
```

General Description

Obtains the address of a symbol defined within a Dynamic Link Library (DLL) made accessible through a dlopen() call. The *handle* argument is the value returned from a call to dlopen() (which has not been released by a call to dlclose()), and *name* is the symbol's name as a character string.

The DLL that was loaded by dlopen() will be searched for the named symbol. If the symbol is not found in that DLL, then the dependent DLLs of that DLL will be searched, followed by any dependents of those, and continuing in a breadth-first manner until the named symbol is found or all dependent DLLs have been searched. This search order determines how duplicate symbols in different DLLs will be found, although the order in which dependent DLLs at the same level are searched is indeterminate.

Also note that a search of dependent DLLs by dlsym() will not result in unloaded dependent DLLs being loaded. Only the dependent DLLs loaded as part of the call to dlopen() will be searched. If the full set of dependent DLLs need to be available to subsequent calls to dlsym(), make sure the DLL is opened with the *RTLD_NOW* load flag. It is indeterminate which dependent DLLs are loaded when *RTLD_LAZY* is specified

The only exception to this is the global symbol object obtained via a dlopen(*NULL*,*0*) call, in which case all DLLs (excluding those opened with *RTLD_LOCAL*) are searched in the order in which they were loaded.

Notes:

- The named symbol can be either an exported data item or function.
- DLLs are enclave level resources.

Returned Value

NULL is returned:

- If *handle* does not refer to a valid DLL opened by dlopen(),
- or the named symbol (*name*) cannot be found within any of the DLLs associated with *handle*.

Application Usage

1. C++ symbol names should be passed to dlsym() in mangled form; dlsym() does not perform any name mangling on behalf of the calling application.
2. More detailed diagnostic information is available through dlerror(), the _EDC_DLL_DIAG environment variable, and the Language Environment DLL Failure control block (CEEDLLF) chain.
3. This function is not available under SPC, MTF and CSP environments.

Example

The following example shows how dlopen() and dlsym() can be used to access either function or data objects. For simplicity, error checking has been omitted.

```
void    *handle;
int     *iptr, (*fptr)(int);

/* open the needed object */
handle = dlopen("/usr/home/me/libfoo.so", RTLD_LOCAL | RTLD_LAZY);

/* find the address of function and data objects */
fptr = (int (*)(int))dlsym(handle, "my_function");
iptr = (int *)dlsym(handle, "my_object");

/* invoke function, passing value of integer as a parameter */
(*fptr)(*iptr);
```

Related Information

- “dlclose() — Close a dlopen() object” on page 168
- “dlerror() — Get Diagnostic Information” on page 169
- “dlopen() — Gain Access to a Dynamic Link Library (DLL)” on page 170

dllfree() — Free the Supplied DLL

This function is deprecated; use “dlclose() — Close a dlopen() object” on page 168 instead.

dllload() — Load the DLL and Connect it to the Application

This function is deprecated; use “dlopen() — Gain Access to a Dynamic Link Library (DLL)” on page 170 instead.

dllqueryfn() — Obtain a Pointer to a DLL Function

This function is deprecated; use “dlerror() — Get Diagnostic Information” on page 169 instead.

dllqueryvar() — Obtain a Pointer to a DLL Variable

This function is deprecated; use “dlsym() — Obtain the Address of a Symbol from a dlopen() Object” on page 172 instead.

dn_comp() — Resolver Domain Name Compression

Standards

Standards / Extensions	C or C++	Dependencies
BSD 4.3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int dn_comp(const char *exp_dn, u_char *comp_dn, int length,
            u_char **dnptrs, u_char **lastdnptr);
```

General Description

The dn_comp() function compresses the domain name *exp_dn* and stores it in the message at *comp_dn*. The size of the storage pointed to by *comp_dn* is given by *length*. The compression uses an array of

pointers *dnptrs* to a list of previously-compressed names in the current message. The first pointer in this array points to the beginning of the message and the array ends with NULL. The limit to the array is specified by *lastdnptr*.

A side effect of dn_comp() is to update the array of pointers for labels inserted into the message as the name is compressed. If *dnptr* is NULL, names are not compressed. If *lastdnptr* is NULL, the array is not updated. *lastdnptr* is not changed by dn_comp().

Returned Value

If successful, dn_comp() returns the size of the compressed name.

If unsuccessful, dn_comp() returns -1 to report the error, when the name to be compressed was not found before the end of the *exp_dn* buffer was reached.

There are no documented errno values.

Related Information

- “arpa/nameser.h” on page 18
- “netinet/in.h” on page 34
- “resolv.h” on page 38
- “sys/types.h” on page 46
- “dn_expand() — Resolver Domain Name Expansion” on page 175
- “dn_find() — Resolver Domain Name Find” on page 176
- “dn_skipname() — Resolver Domain Name Skipping ” on page 177
- “res_init() — Domain Name Resolver Initialization” on page 663
- “res_mkquery() — Make Resolver Query for Domain Name Servers (DNS)” on page 665
- “res_query() — Resolver Query for Domain Name Servers (DNS)” on page 666
- “res_querydomain() — Build Domain Name and Resolver Query ” on page 667
- “res_search() — Resolver Query for Domain Name Servers (DNS) With Optional Rules” on page 668
- “res_send() — Send Resolver Query for Domain Name Servers (DNS)” on page 669

dn_expand() — Resolver Domain Name Expansion

Standards

Standards / Extensions	C or C++	Dependencies
BSD 4.3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int dn_expand(const u_char *msg, const u_char *eomorig,
              const u_char *comp_dn, char *exp_dn, int length);
```

General Description

The dn_expand() function expands the compressed domain name *comp_dn* to a full domain name. The compressed name is contained in a query or reply message; *msg* is a pointer to the beginning of the

message. The expanded name is placed in the buffer indicated by *exp_dn* which is of size *length*. *eomorig* points to the end of the message.

Returned Value

If successful, dn_expand() returns the size of the expanded name.

If unsuccessful, dn_expand() returns -1 to report the error, when the name to be expanded was not found before the end of the *comp_dn* buffer was reached.

There are no documented errno values.

Related Information

- "arpa/nameser.h" on page 18
- "netinet/in.h" on page 34
- "resolv.h" on page 38
- "sys/types.h" on page 46
- "dn_comp() — Resolver Domain Name Compression" on page 174
- "dn_find() — Resolver Domain Name Find" on page 176
- "dn_skipname() — Resolver Domain Name Skipping " on page 177
- "res_init() — Domain Name Resolver Initialization" on page 663
- "res_mkquery() — Make Resolver Query for Domain Name Servers (DNS)" on page 665
- "res_query() — Resolver Query for Domain Name Servers (DNS)" on page 666
- "res_querydomain() — Build Domain Name and Resolver Query " on page 667
- "res_search() — Resolver Query for Domain Name Servers (DNS) With Optional Rules" on page 668
- "res_send() — Send Resolver Query for Domain Name Servers (DNS)" on page 669

dn_find() — Resolver Domain Name Find

Standards

Standards / Extensions	C or C++	Dependencies
BSD 4.3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int dn_find(u_char *exp_dn, u_char *msg, u_char **listptr, u_char **lastdnptr);
```

General Description

The dn_find() function will search for the expanded name exp_dn using the listptr list of pointers to previously-compressed names in the current message.

listptr is equivalent to *dnptrs*+4. (For a description of *dnptrs*, see "dn_comp() — Resolver Domain Name Compression" on page 174.) The limit to the list is specified by *lastdnptr*. *msg* points to the start of the message.

Returned Value

If successful, dn_find() returns the offset of the expanded name *exp_dn* found in the message.

If unsuccessful, dn_find() returns -1 to report the error, when the name was not found before the end of the list was reached.

There are no documented errno values.

Related Information

- "arpa/nameser.h" on page 18
- "netinet/in.h" on page 34
- "resolv.h" on page 38
- "sys/types.h" on page 46
- "dn_comp() — Resolver Domain Name Compression" on page 174
- "dn_expand() — Resolver Domain Name Expansion" on page 175
- "dn_skipname() — Resolver Domain Name Skipping " on page 177
- "res_init() — Domain Name Resolver Initialization" on page 663
- "res_mkquery() — Make Resolver Query for Domain Name Servers (DNS)" on page 665
- "res_query() — Resolver Query for Domain Name Servers (DNS)" on page 666
- "res_querydomain() — Build Domain Name and Resolver Query " on page 667
- "res_search() — Resolver Query for Domain Name Servers (DNS) With Optional Rules" on page 668
- "res_send() — Send Resolver Query for Domain Name Servers (DNS)" on page 669

dn_skipname() — Resolver Domain Name Skipping

Standards

Standards / Extensions	C or C++	Dependencies
BSD 4.3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int dn_skipname(const u_char *comp_dn, u_char *eom);
```

General Description

The dn_skipname() function skips the compressed domain name *comp_dn* in the current message and returns the position in the message that follows the *comp_dn* compressed domain name. *eom* points to the end of the message.

Returned Value

If successful, dn_skipname() returns the position in the message that follows the *comp_dn* compressed domain name.

If unsuccessful, dn_skipname() returns -1 to report the error, when the name to be skipped was not found before the end of the *comp_dn* buffer was reached.

There are no documented errno values.

Related Information

- “arpa/nameser.h” on page 18
- “netinet/in.h” on page 34
- “resolv.h” on page 38
- “sys/types.h” on page 46
- “dn_comp() — Resolver Domain Name Compression” on page 174
- “dn_expand() — Resolver Domain Name Expansion” on page 175
- “dn_find() — Resolver Domain Name Find” on page 176
- “res_init() — Domain Name Resolver Initialization” on page 663
- “res_mkquery() — Make Resolver Query for Domain Name Servers (DNS)” on page 665
- “res_query() — Resolver Query for Domain Name Servers (DNS)” on page 666
- “res_querydomain() — Build Domain Name and Resolver Query ” on page 667
- “res_search() — Resolver Query for Domain Name Servers (DNS) With Optional Rules” on page 668
- “res_send() — Send Resolver Query for Domain Name Servers (DNS)” on page 669

drand48() — Generate Pseudo-Random Number

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(), mrand48() and nrand48() functions generate uniformly distributed pseudorandom numbers using a linear congruential algorithm and 48-bit integer arithmetic. These functions have been extended so that the returned value will bein the proper floating-point format (hexadecimal or IEEE) based on the floating-point mode of the invoking thread.

- The drand48() and erand48() functions return non-negative, double-precision, floating-point values, uniformly distributed over the interval `[0.0,1.0]`. These functions have been extended so that the returned value will be in the proper floating-poing format (hexadecimal or IEEE) based on the floating-point mode of the calling thread.
- The lrand48() and nrand48() functions return non-negative, long integers, uniformly distributed over the interval `[0,2**31]`.
- The mrand48() and jrand48() functions 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)mod(2**48)     n>=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)`. The drand48(), lrand48(), and mrand48() functions share this storage. You may reinitialize the value of `X(n)` in this storage by calling the lcong48(), seed48() or srand48() function. Likewise, you may change the values of a and c by calling lcong48(). Thereafter, whenever you call seed48() or srand48() to change `X(n)`, the initial values of a and c are also reestablished.

Special Behavior for OpenExtensions Services

You can make drand48() 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 VM/ESA serializes access to the storage for `X(n)`, a, and c by functions in the `drand48` family when a multithreaded application calls them.

If you request thread-specific behavior and call drand48() from thread `t`, drand48() generates the next 48-bit integer value in a sequence of 48-bit integer values, `X(t,i)`, for the thread `t`, according to the linear congruential formula:

```
X(t,n+1) = (a(t)X(t,n) + c(t))mod(2**48)     n>=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 that is specific to the thread `t` to save the most recent 48-bit integer value of the sequence, `X(t,i)`, generated by drand48(), lrand48(), or mrand48(). You may reinitialize the value `X(t,n)` in this storage by calling lcong48(), seed48(), or srand48() from the thread `t`. Likewise, you may change the values of `a(t)` and `c(t)` for thread `t` by calling lcong48() from the thread. Thereafter, whenever you call seed48() or srand48() from the thread `t` to change `X(t,n)`, the initial values of `a(t)` and `c(t)` are also reestablished.

Returned Value

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 OpenExtensions Services

If you request thread-specific behavior for the `drand48` family and call `drand48` on thread `t`, drand48() 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 40
- “erand48() — Generate Pseudo-Random Number” on page 187
- “jrand48() — Generate Pseudorandom Number” on page 436
- “lcong48() — Initialize Pseudorandom Number” on page 444
- “lrand48() — Generate Pseudorandom Number” on page 467
- “mrand48() — Generate Pseudorandom Number” on page 510
- “nrand48() — Pseudo-Random Number Generator” on page 523
- “seed48() — Initialize Pseudorandom Number” on page 679

- “srand48() — Initialize Pseudorandom Number” on page 801

dup() — Duplicate an Open File Descriptor

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

Format

```
#define _OPEN_SYS
#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 207.

Returned Value

If successful, dup() returns a new file descriptor.

If unsuccessful, dup() returns a value of -1 and sets errno to one of the following values under the conditions described.

EBADF
fildes is not a valid open file descriptor.

EEXTLINK
The dup() function is not supported for external links.

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 _OPEN_SYS
#include <errno.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _OPEN_SYS
#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 51
- “close() — Close a File” on page 127
- “creat() — Create a New File or Rewrite an Existing One” on page 146
- “dup2() — Duplicate an Open File Descriptor to Another” on page 181
- “exec Functions” on page 192
- “fcntl() — Control Open File Descriptors” on page 207
- “open() — Open a File” on page 527
- “pipe() — Create an Unnamed Pipe” on page 539

dup2() — Duplicate an Open File Descriptor to Another

Standards/Extensions	C or C++	Dependencies
POSIX.1	both	

Format

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

int dup2(int fd1, int fd2);
```

General Description

Returns a file descriptor with the value *fd2*, which now refers to the same file as *fd1*. 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 a value of -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, dup2() returns a value of -1 and sets errno to one of the following values under the conditions described.

EBADF
fd1 is not a valid file descriptor, or *fd2* is less than 0 or greater than OPEN_MAX.

EEXTLINK
The dup2() function is not supported for external links.

EINTR
The dup2() function was interrupted by a signal.

Example

CBC3BD06

```
/* CBC3BD06
   This example duplicates an open file descriptor, using dup2().
 */
#define _OPEN_SYS
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _OPEN_SYS
#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 51

- “close() — Close a File” on page 127
- “creat() — Create a New File or Rewrite an Existing One” on page 146
- “dup() — Duplicate an Open File Descriptor” on page 180
- “exec Functions” on page 192
- “fcntl() — Control Open File Descriptors” on page 207
- “open() — Open a File” on page 527
- “pipe() — Create an Unnamed Pipe” on page 539

ecvt() — Convert Double to String

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

Converts *x* to a null-terminated string of *ndigit* digits (where the total number of digits in the output string is determined by the precision of a double) and returns a pointer to the string.

The high-order digit is nonzero, unless the value is zero. 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 zero.

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 215 and “gcvt() — Convert Double to String” on page 311.

Returned Value

If successful, ecvt() returns the character equivalent of *x* as specified above.

If ecvt() is not able to allocate the return buffer or the converted value is out of range or is not representable, ecvt() returns NULL.

Related Information

- “stdlib.h” on page 42
- “fcvt() — Convert Double to String” on page 215
- “gcvt() — Convert Double to String” on page 311

endhostent() — Work with a Host Entry

Standards/Extensions	C or C++	Dependencies
POSIX.1 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

Closes the HOSTS SITEINFO file, which contains information about known hosts.

You can use the X_SITE environment variable to specify a file other than HOSTS SITEINFO.

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 335
- "gethostbyname() — Get a Host Entry by Name" on page 337
- "gethostent() — Get Next Host Entry" on page 339
- "sethostent() — Open Host Information Data Set" on page 712

endnetent() — Close Network Information Data Set

Standards/Extensions	C or C++	Dependencies
POSIX.1 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

Closes the HOSTS ADDRINFO file, which contains information about known networks.

You can use the X_ADDR environment variable to specify a file other than HOSTS ADDRINFO.

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 347
- "getnetbyname() — Get a Network Entry by Name" on page 348
- "getnetent() — Get Next Network Entry" on page 350
- "setnetent() — Open Network Information Data Set" on page 728

endprotoent() — Work with a Protocol Entry

Standards/Extensions	C or C++	Dependencies
POSIX.1 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

Closes the ETC PROTO file, 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 358
- "getprotoent() — Get Next Protocol Entry" on page 360
- "setprotoent() — Open Protocol Information File" on page 731

endservent() — Close Network Services Information Data Sets

Standards/Extensions	C or C++	Dependencies
POSIX.1 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

Closes the ETC SERVICES file, 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 365
- "getservbyport() — Get a Service Entry by Port" on page 366
- "getservent() — Get Next Service Entry" on page 367
- "setservent() — Open Network Services Information File" on page 732

endutxent() — Close the utmpx Database

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

Format

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

void endutxent(void);
```

General Description

Closes the utmpx database for the current thread. The database may be opened by getutxent(), getutxid(), getutxline(), or pututxline().

Because endutxent() processes thread-specific data, you can use endutxent() safely from a multithreaded 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 calls endutxent().

Programs should not reference the data passed back by getutxline(), getutxid(), getutxent(), or pututxline(), after endutxent() has been called (the storage has been freed).

After getuxline(), 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.

Note: You can cause all OpenExtensions 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. The endutxent() function 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 has no returned value.

Related Information

- "__utmpxname() — Change the utmpx Database Name" on page 915
- "getutxent() — Read Next Entry in utmpx Database" on page 379
- "getutxid() — Search by ID utmpx Database" on page 380
- "getutxline() — Search by Line utmpx Database" on page 381
- "pututxline() — Write Entry to utmpx Database" on page 616
- "setutxent() — Reset to Start of utmpx Database" on page 741

erand48() — Generate Pseudo-Random Number

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 pseudo-random numbers using a linear congruential algorithm and 48-bit integer arithmetic.

- The drand48() and erand48() functions return nonnegative, double-precision, floating-point values, uniformly distributed over the interval `[0.0,1.0]`. These functions have been extended so that the returned value will be in the proper floating-point format (hexadecimal or IEEE) based on floating-point mode of the invoking thread.
- The lrand48() and nrand48() functions return non-negative, long integers, uniformly distributed over the interval `[0,2**31]`.
- The mrand48() and jrand48() functions 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)mod(2**48)     n>=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)
```

You may change the values of a and c by calling the lcong48() function. Calling either seed48() or srand48() restores a and c.

Special Behavior for OpenExtensions Services

You can make erand48() 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, IBM XL C/C++ for z/VM serializes access to the storage for X(n), a, and c by functions in the drand48 family when a multithreaded application calls them.

If you request thread-specific behavior and call erand48() from thread t, erand48() generates the next 48-bit integer value in a sequence of 48-bit integer values, X(t,i), for the thread t, according to the linear congruential formula:

```
X(t,n+1) = (a(t)X(t,n) + c(t))mod(2**48)     n>=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 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)
```

You may change the values of a(t) and c(t) by calling the lcong48() function from the thread t. Calling either seed48() or srand48() from the thread restores the initial values of a(t) and c(t).

Returned Value

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 OpenExtensions Services

If you request thread-specific behavior for the drand48 family and call erand48() on thread t, erand48() 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 42
- “drand48() — Generate Pseudo-Random Number” on page 178
- “jrand48() — Generate Pseudorandom Number” on page 436
- “lcong48() — Initialize Pseudorandom Number” on page 444
- “lrand48() — Generate Pseudorandom Number” on page 467
- “mrand48() — Generate Pseudorandom Number” on page 510
- “nrand48() — Pseudo-Random Number Generator” on page 523
- “seed48() — Initialize Pseudorandom Number” on page 679
- “srand48() — Initialize Pseudorandom Number” on page 801

erf() - erfc() — Calculate Error and Complementary Error Functions

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

LANGLVL(EXTENDED), LANGLVL(SAA), or LANGLVL(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 - erf(x)`, it is used in place of erf() for large values of *x*.

Returned Value

Both erf() and erfc() return the calculated value. If the correct value would cause underflow, the functions return `0` and the value of the macro ERANGE is stored in errno.

Special Behavior for IEEE

erf() and erfc() are always successful.

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 32
- "gamma() — Calculate Gamma Function" on page 310
- "j0(), j1(), jn() — Bessel Functions of the First Kind" on page 438
- "y0(), y1(), yn() — Bessel Functions of the Second Kind" on page 983

__errno2() — Return Reason Code Information

Standards/Extensions	C or C++	Dependencies
OpenExtensions	both	

Format

```
int  __errno2(void);
```

General Description

Returns the reason code of the last OpenExtensions system service called by the IBM XL C/C++ for z/VM library. It provides OpenExtensions programs access to diagnostic information returned to C from an underlying OpenExtensions callable service.

The __errno2() function is provided to help you determine more about the cause of failure of your application program. You should use it for diagnostic purposes *only*, and only when it is recommended by IBM service personnel. 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 the appendix of *z/VM: OpenExtensions 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);
}
```

__etoa() — Convert EBCDIC String to ISO8859-1

Standards/Extensions	C or C++	Dependencies
OpenExtensions	both	

Format

```
#include <unistd.h>

int __etoa(char *string);
```

General Description

Converts an EBCDIC character string, pointed to by *string*, to its ISO8859-1 equivalent using the codeset page associated with the current locale.

The __etoa() function changes all characters in the input string up to, but not including, the null from the current locale to an ISO8859-1 representation.

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 a value of -1 and sets errno to one of the following values under the conditions described. (This function may internally call iconv_open() and iconv(), which return errnos that are propagated without modification.)

EINVAL
The current locale does not describe a single-byte character set.

ENOMEM
Insufficient storage space prevents completion of the conversion process.

Related Information

- "sys/msg.h" on page 45

__etoa_l() — Convert EBCDIC Bytes to ISO8859-1

Standards/Extensions	C or C++	Dependencies
OpenExtensions	both	

Format

```
#include <unistd.h>

int __etoa_l(char *bufferptr, int leng);
```

General Description

Converts the *leng* of EBCDIC bytes in the buffer, *bufferptr*, to the ISO8859-1 equivalent using the codeset page associated with the current locale.

The __etoa_l() function treats the input buffer as a sequence of bytes, and converts all bytes in the input buffer, 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 a value of -1 and sets errno to one of the following values under the conditions described. (This function may internally call iconv_open() and iconv(), which return errnos that are propagated without modification.)

EINVAL
The current locale does not describe a single-byte character set.

ENOMEM
Insufficient storage space prevents completion of the conversion process.

Related Information

- "sys/msg.h" on page 45

exec Functions

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

Format

```
#define _OPEN_SYS
#include <unistd.h>
int execl(const char *path, const char *arg, …, NULL);
int execle(const char *path, const char *arg, …, NULL,
           char *const envp[]);
int execlp(const char *file, const char *arg, …,NULL);
int execv(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 inclusion of unistd.h you should 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 byte file system (BFS) or from an accessed CMS mini-disk or shared file system directory.

The exec is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

For information on specifying names for CMS record files and BFS files, see *z/OS: XL C/C++ Programming Guide*.

A successful exec function never returns control because the calling process is overwritten with the new process.

The *path* argument 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 BFS.

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. A NULL pointer after the last character string must 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. A NULL pointer after the last character string must 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* to which the *external variable **environ* points.

The variable ARG_MAX, obtained from OpenExtensions 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. For more information about this flag, see “fcntl() — Control Open File Descriptors” on page 207. 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.

Signals set to be ignored in the caller, `SIG_IGN`, are set to be ignored in the new process image. You should 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.

The new process also inherits the following from the caller:

- Process ID
- Parent process ID
- Process group ID
- Session membership
- Time left until an alarm clock signal
- Working directory
- Root directory
- File mode creation mask
- Process signal mask

- Pending signals
- `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime`. For more about these qualities, see "times() — Get Process and Child Process Times" on page 886.

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.

For more information on interactions with CMS program objects, see *z/VM: OpenExtensions Callable Services Reference*.

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 a value of -1 and sets errno to one of the following values under the conditions described:

E2BIG
The combined argument list and environment list of the new process has more bytes than the system-defined length. For information about the system-defined length, see "sysconf() — Determine System Configuration Options" on page 856.

EACCES
The process did not have appropriate permissions to run the specified file, for one of these reasons:

- The process did not have permission to search a directory named in your *path*.
- The process did not have execute permission for the file to be run.
- The system cannot run files of this type.

ECMSERR
Insufficient storage prevented completion of the operation.

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. The exec functions issue this error if they detect the number of symbolic links in the resolution of the *path* or *file* is greater than POSIX_SYMLOOP (a value defined in the limits.h header file)

EMFILE
Currently too many files are open in this process.

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().

ENFILE
Currently too many files are open in the system.

ENOENT
One or more *pathname* components in *path* or *file* does not exist. The exec functions also issue this error if *path* or *file* is a null string.

ENOEXEC
The new process image file has the appropriate access permission but has an unrecognized format.

ENOMEM
The new process requires more memory than is permitted by the operating system.

ENOTDIR
A directory component of *path* or *file* is not really a directory.

Example

CBC3BE03

```
/* CBC3BE03
   This example runs a program, using the execl() function.
 */
#define _OPEN_SYS
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

main() {

  execl("/bin/false", NULL);
  perror("The execl() call must have failed");
  exit(255);
}
```

Output

There is no output for this example.

Related Information

- “limits.h” on page 28
- “signal.h” on page 38
- “unistd.h” on page 51
- “alarm() — Set an Alarm” on page 76
- “chmod() — Change the Mode of a File or Directory” on page 117
- “_exit() — End a Process and Bypass the Cleanup” on page 195
- “fcntl() — Control Open File Descriptors” on page 207
- “setuid() — Set the Effective User ID” on page 739
- “sigaction() — Examine or Change a Signal Action” on page 749
- “sigpending() — Examine Pending Signals” on page 772
- “sigprocmask() — Examine or Change a Thread” on page 773
- “stat() — Get File Information” on page 803
- “times() — Get Process and Child Process Times” on page 886
- “umask() — Set or Display the File Mode Creation Mask” on page 902

_exit() — End a Process and Bypass the Cleanup

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

Format

```
#define _OPEN_SYS
#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.

This function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

The *status* argument specifies a return status for the process that is ending. Ending the process has the following results:

- The _exit() function 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 930.
- 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. 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. The _exit() function does not cause C runtime library cleanup to be performed; therefore, stream buffers are not necessarily flushed.

Special Behavior for C++

If _exit() is called in a C++ 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

The _exit() function is always successful and does not have a returned value. No value is stored in *errno* for this function.

Example

CBC3BE05

```
/* CBC3BE05
   This example ends a process.
 */
#define _OPEN_SYS
#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

- "unistd.h" on page 51
- "stdlib.h" on page 42
- "abort() — Stop a Program" on page 66
- "atexit() — Register Program Termination Function" on page 85
- "close() — Close a File" on page 127
- "sigaction() — Examine or Change a Signal Action" on page 749
- "signal() — Handle Interrupts" on page 767
- "wait() — Wait for a Child Process to End" on page 927

exit() — End Program

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, and destroys C++ objects with static storage duration, all in last-in-first-out (LIFO) order. C++ objects with static storage duration are destroyed in the reverse order of the completion of their constructor. (Automatic objects are not destroyed as a result of calling exit().)
2. Flushes all buffers, and closes all open files.
3. Deletes all files opened with tmpfile().
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. For more information about using POSIX support, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

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 z/VM C++ program, the program terminates without leaving the current block, and therefore destructors are not called for local (automatic) variables. Destructors for initialized statis objects will be called in the reverse order of the completion of their constructors.

Functions registered with atexit() are called in the reverse order of their registration. A function registered with atexit(), before an object obj1 of static storage duration is initialized, will not be called until obj1's destruction has completed. A function registered with atexit(), after an object obj2 of static storage duration is initialized, will be called before obj2's destruction starts.

Returned Value

The exit() function has 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

- For information about using runtime user exits, see *z/OS: XL C/C++ Programming Guide*.
- “stdlib.h” on page 42
- “abort() — Stop a Program” on page 66
- “atexit() — Register Program Termination Function” on page 85
- “_exit() — End a Process and Bypass the Cleanup” on page 195
- “signal() — Handle Interrupts” on page 767
- “wait() — Wait for a Child Process to End” on page 927
- “waitpid() — Wait for a Specific Child Process to End” on page 930

exp(), expf(), expl()— Calculate Exponential Function

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

Format

```
#include <math.h>

double exp(double x);
float exp(float x);                        /* C++ only */
long double exp(long double x);            /* C++ only */
float expf(float x);
long double expl(long double x);
```

General Description

Calculates the exponent of *x*, defined as *e***x*, where *e* equals 2.17128128...).

Note: These functions work in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

Returned Value

Returns the calculated value if successful. If an overflow occurs, exp() returns HUGE_VAL.

If an underflow occurs, exp() returns a value of 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 32
- "log(), logf(), logl() — Calculate Natural Logarithm" on page 460

- "log10(), log10f(), log10l() — Calculate Base 10 Logarithm" on page 462
- "pow() — Raise to Power" on page 543

expm1() — Calculate Exponential Minus One

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

Format

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

double expm1(double x);
```

General Description

Calculates the function $e^x - 1.0$.

Note: These functions work in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

Returned Value

If successful, expm1() returns the above function calculated on *x*.

The expm1() function will fail under the following conditions:

- 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 32
- "exp(), expf(), expl()— Calculate Exponential Function" on page 199
- "ilogb() — Integer Unbiased Exponent" on page 406
- "log1p() — Calculate Natural Log of x plus 1" on page 461

fabs(), fabsf(), fabsl() — Calculate Floating-Point Absolute Value

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

Format

```
#include <math.h>

double fabs(double x);
float fabs(float x);                  /* C++ only */
long double fabs(long double x);      /* C++ only */
float fabsf(float x);
long double fabsl(long double x);
```

General Description

The built-in fabs() functions calculate the absolute value of a floating-point argument.

Note: These functions work in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 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 32
- "abs(), absf(), absl() — Calculate Integer Absolute Value" on page 68
- "labs() — Calculate Long Absolute Value" on page 443

fchaudit() — Change Audit Flags for a File by Descriptor

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	

Format

```
#define _OPEN_SYS
#include <sys/stat.h>

int fchaudit(int fildes, unsigned int flags, unsigned int option);
```

General Description

Changes the audit flags of a file. The *fildes* parameter is thefile descriptor for the open file whose audit flags you should change. The *flags* parameter specifies what you should change the audit flags 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. You can use the bitwise inclusive OR of any or all of these to set more than one type of auditing.

The *option* parameter specifies whether you should change the user audit flags or the security auditor audit flags:

AUDT_USER (0)
Changes user audit flags. The user must be the file owner or have appropriate authority to change the user audit flags for a file.

AUDT_AUDITOR (1)
Changes security-auditor audit flags. The user must have security auditor authority to change the security auditor audit flags for a file.

Returned Value

If successful, fchaudit() returns a value of `0`.

If unsuccessful, fchaudit() returns a value of -1 and sets errno to one of the following values under the conditions described:

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
fildes 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.
 */
#define _OPEN_SYS
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#undef _OPEN_SYS
#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 46
- "access() — Determine Whether a File Can be Accessed" on page 71
- "chaudit() — Change Audit Flags for a File by Path" on page 114
- "fchmod() — Change the Mode of a File or Directory by Descriptor" on page 203
- "fchown() — Change the Owner or Group by File Descriptor" on page 204
- "fstat() — Get Status Information about a File" on page 296

fchmod() — Change the Mode of a File or Directory by Descriptor

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

Format

```
#define _OPEN_SYS
#include <sys/stat.h>

int fchmod(int fildes, mode_t mode);
```

General Description

Sets the S_ISUID, S_ISGID, and file permission bits of the open file identified by *fildes*, 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, see "chmod() — Change the Mode of a File or Directory" on page 117.

Returned Value

If successful, fchmod() marks for update the `st_ctime` field of the file and returns a value of `0`.

If unsuccessful, fchmod() returns a value of -1 and sets errno to one of the following values under the conditions described:

EBADF
fildes 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 _OPEN_SYS
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _OPEN_SYS
#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 46
- "chmod() — Change the Mode of a File or Directory" on page 117
- "chown() — Change the Owner or Group of a File or Directory" on page 120
- "fchown() — Change the Owner or Group by File Descriptor" on page 204
- "mkdir() — Make a Directory" on page 498
- "mkfifo() — Make a FIFO Special File" on page 499
- "open() — Open a File" on page 527
- "stat() — Get File Information" on page 803

fchown() — Change the Owner or Group by File Descriptor

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

Format

```
#define _OPEN_SYS
#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. The *fildes* argument 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:

- The process has appropriate privileges.

- The effective user ID of the process matches the user ID (UID) of the owner of the file; the *owner* argument matches the user ID of the file owner; the *group* argument is the effective group ID or the 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 a value of `0`.

If unsuccessful, fchown() returns a value of -1 and sets errno to one of the following values under the conditions described:

EBADF
fildes is not a valid open file descriptor.

EPERM
Either the effective user ID does not match the owner of the file, or the calling process does not have appropriate privileges; and POSIX_CHOWN_RESTRICTED 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 _OPEN_SYS
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#undef _OPEN_SYS
#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 51
- "chmod() — Change the Mode of a File or Directory" on page 117
- "chown() — Change the Owner or Group of a File or Directory" on page 120
- "fchmod() — Change the Mode of a File or Directory by Descriptor" on page 203
- "mkdir() — Make a Directory" on page 498
- "mkfifo() — Make a FIFO Special File" on page 499
- "open() — Open a File" on page 527
- "stat() — Get File Information" on page 803

fclose() — Close File

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 fclose() function frees the storage pointed to by the FILE pointer. An attempt to use the FILE pointer to a closed file not 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

If successful, fclose() returns a value of `0`. If a failure occurs in flushing buffers or in outputting data, fclose() returns EOF. The fclose() function will still try to close the file.

Special Behavior for XPG4

The fclose() function sets errno to one of the following values:

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
No free space is left on the output device.

ENXIO
A request was made of a non-existent device, or the request was outside the device.

EPIPE
The fclose() function 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.

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

- *z/OS: XL C/C++ Programming Guide*
- "stdio.h" on page 40
- "fopen() — Open a File" on page 254
- "freopen() — Redirect an Open File" on page 280

fcntl() — Control Open File Descriptors

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

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 *fildes* argument 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 controlled operations are determined by *cmd*. The *arg* parameter is a variable with a meaning that depends on the value of the *cmd* parameter.

socket
: The socket descriptor.

cmd
: The command to perform.

arg
: The data associated with *cmd*.

The *action* argument can be one of the following symbols:

F_CLOSFD
: Closes a range of file descriptors. You must specify a third `int` argument to indicate the upper limit for the range of the file descriptors to be closed, while *fildes* specifies the lower limit. If you specify `-1` for the third argument, all file descriptors greater than or equal to the lower limit are closed.

F_DUPFD
: Duplicates the file descriptor. You must specify a third `int` argument; fcntl() 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 *fildes* and shares any locks. The FD_CLOEXEC and FD_CLOFORK flags are turned off in the new file descriptor so that the file is kept open if an exec function is called.

F_DUPFD2
: Duplicates the file descriptor. You must specify a third `int` argument 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 *fildes* and shares any locks. The FD_CLOEXEC and FD_CLOFORK flags are turned off in the new file descriptor so that the file is kept open if an exec function is called.

F_GETFD
: Obtains the file descriptor flags for *fildes*. The fcntl() function returns these flags as its result. For a list of supported file descriptor flags, see “File Flags” on page 210.

F_SETFD
Sets the file descriptor flags for *fildes*. You must specify a third `int` argument, giving the new file descriptor flag settings. The fcntl() function returns a value of `0` if it successfully sets the flags.

F_GETFL
Obtains the file status flags and file access mode flags for *fildes*. The fcntl() function returns these flags as its result. For a list of supported file descriptor flags, see "File Flags" on page 210.

Behavior for Sockets: This command gets the status flags of socket descriptor *socket*. With the _OPEN_SYS feature test macro, you can query the FNDELAY flag. With the _XOPEN_SOURCE_EXTENDED 1 feature test macro, you can query the O_NDELAY flag. The FNDELAY and O_NDELAY flags mark *socket* as being in nonblocking mode. If data is not present on calls that can block, such as read(), readv(), and recv(), the call returns a value of `-1` and sets errno to EWOULDBLOCK.

F_SETFL
Sets the file status flags for *fildes*. You must specify a third `int` argument, giving the new file descriptor flag settings. The fcntl() function does not change the file access mode and ignores file access bits in the third argument. The fcntl() function returns a value of `0` if it successfully sets the flags.

Behavior for Sockets: This command sets the status flags of socket descriptor *socket*. With the _OPEN_SYS feature test macro, you can set the FNDELAY flag. With the _XOPEN_SOURCE_EXTENDED 1 feature test macro, you can set the O_NDELAY flag.

F_GETLK
Obtains locking information for a file. For details, see "File Locking" on page 211.

F_SETLK
Sets or clears a file segment lock. For details, see "File Locking" on page 211.

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. For details, see "File Locking" on page 211.

F_SETTAG
Sets the file tag for the file referred to by file descriptor *fildes*.

The third argument *ftag* is the address of a populated file_tag structure.

If the *ftag* argument supplied to fcntl(F_SETTAG) does not have the ft_deferred bit set ON, fcntl() will immediately set the file's File Tag with the provided *ftag*'s ft_ccsid and ft_txtflag values.

If the *ftag* argument supplied to fcntl(F_SETTAG) has the ft_deferred bit set ON, fcntl() will not set the file's File Tag until first write to the file. The CCSID used to tag the file will be the current Program CCSID at the time of first write, regardless of the *ftag* ft_ccsid value, however the ft_txtflag value will be used.

If the ft_ccsid of the specified file_tag differs from the Program CCSID, automatic file conversion will occur, provided:

- The ft_txtflag is set to ON.
- The BPXPRMxx member AUTOCVT() is ON or environment variable _BPXK_AUTOCVT is ON.

If AUTOCVT(OFF) and _BPXK_AUTOCVT=OFF, the file will be tagged with the specified file_tag's ft_ccsid and ft_txtflag values, but automatic conversion will not occur.

If the *ftag* argument supplied to fcntl(F_SETTAG) has the ft_deferred bit set ON, pipes and FIFOs are tagged from the write end with the Program CCSID of the first writer.

F_CONTROL_CVT
Controls or queries the conversion status of the open file referred to by file descriptor *fildes*. Conversion control is generally used to provide CCSID information for untagged files or untagged programs.

Character set conversion between a program and a file, pipe or other I/O stream can be enabled or changed with F_CONTROL_CVT. A pair of CCSID's is specified or defaulted, one for the program and one for the data. As the program reads and writes data, the system will convert from one CCSID to the other.

The third f_cnvrt argument is the required address of an f_cnvrt structure. This structure is defined in <fcntl.h> and includes the following members:

Table 9. Struct f_cnvrt Element Descriptions

Element	Data Type	Description
pccsid	short	The Program CCSID - This is output from query and input to setting conversion ON. A value of 0 on input indicates to use the previously set value or the current Program CCSID.
fccsid	short	The File CCSID - This is output from query and input to setting conversion ON. A value of 0 on input indicates to use the CCSID from the File Tag as stored in the file, specified on mount, or set by a prior call.
cvtcmd	int	Conversion Control Command. The following conversion controls are available: • Query Conversion - Returns whether or not conversion is in effect and the Program and File CCSIDs being used. On input, *cvtcmd* is set to QUERYCVT, and on output, it is changed to either SETCVTON or SETCVTOFF to indicate that conversion is currently ON or OFF respectively. The current CCSIDs are also returned in their respective positions in the f_cnvrt structure. • Set Conversion OFF - Turns OFF any conversion that may be in effect. On input, *cvtcmd* is set to SETCVTOFF and the rest of the f_cnvrt is ignored. There is no output. A program can use this to override an automatic conversion that might be established by the environment within which it is invoked. If conversion is currently in effect, the CCSIDs being used will be remembered while conversion is turned OFF, so that the prior conversion may be resumed without the program having to remember what the prior CCSIDs were. • Set Conversion ON - Turns ON conversion and optionally specifies the CCSIDs to use in place of the Program or File CCSIDs that are currently in effect. A value of 0 for the Program CCSID indicates that the current Program CCSID be used. A value of 0 for the file CCSID indicates that no change should be made to the File CCSID. This does not affect the stored File Tag or the current Program CCSID. It only changes the values being used to control conversion on this data stream. • On input, *cvtcmd* is set to either SETCVTON to unconditionally turn on conversion or to SETAUTOCVTON to turn ON conversion only if AUTOCVT(YES) was specified in BPXPRMxx or _BPXK_AUTOCVT=ON.

The call fails if a conversion table is not installed for the resulting CCSID pair.

Attention: : Flipping the autoconversion mode off and on, or changing the CCSID values during the execution of a program in which file conversion and/or tagging takes place, or setting the CCSIDs to values that are not compatible with the program or file, can be quite unpredictable.

File Flags

Several types of flags are 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_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 " on page 628 and "write() — Write Data on a File or Socket" on page 975.

O_SYNC
Force synchronous update. 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. (For information, see fsync().) 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 accessed the file 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 a read lock already exists 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 the following 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(fildes,F_SETLK,&lock_it);
```

This example sets up a `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 a value of `-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 and sets errno 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 a `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 no locks 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.

Large Files for HFS

Applications that are compiled with the option LANGLVL(EXTENDED) and also define the Feature Test Macro (FTM) _LARGE_FILES before any headers are included will enable this function to operate on HFS files that are larger than 2 gig-1 in size. File size and offset fields will be enlarged to 63 bits in width so any other function operating on this file will have to be enabled with the same FTM.

Usage of the F_GETFL command will return the setting of O_LARGEFILE status flag when fcntl() has been enabled to operate on large files

Returned Value

If successful, the value returned will depend on the *action* that was specified.

If unsuccessful, fcntl() returns a value of -1 and sets errno to one of the following values under the conditions described:

EAGAIN
The process tried to set a lock with F_SETLK, but the lock is in conflict with a lock already set by some other process on an overlapping part of the file.

EBADF
fildes 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 *socket* parameter is not a valid socket descriptor.

In an F_DUPFD2 operation, the third argument is negative or greater than or equal to OPEN_MAX, which is the highest file descriptor value allowed for the process.

EDEADLK
The system detected the potential for deadlock in a F_SETLKW operation.

EINTR
The fcntl() function was interrupted by a signal during a F_SETLKW operation.

EINVAL
In an F_DUPFD operation, the third argument is negative or greater than or equal to OPEN_MAX, the highest file descriptor value allowed for the process. The OPEN_MAX value can be determined using pathconf().

In a locking operation, *fildes* refers to a file with a type that does not support locking, or the `struct flock` pointed to by the third argument has an incorrect form.

If an F_CLOSFD operation, the third argument, which specifies the upper limit, is less than *fildes* but is not equal to `-1`.

Behavior for Sockets: The *arg* parameter is not a valid flag, or the *cmd* parameter is not a valid command.

EMFILE
In an F_DUPFD operation, the process has already reached its maximum number of file descriptors, or no file descriptors are available 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.

EOVERFLOW
One of the values to be returned cannot be represented correctly.

The cmd argument is F_GETLK, F_SETLK or F_SETLKW and the smallest or, if l_len is nonzero, the largest offset of any byte in the requested segment cannot be represented correctly in an object of type `off_t`.

EPERM
The operation was F_CLOSFD, but all the requested file descriptors were not closed.

Example

CBC3BF06

```
/* CBC3BF06
   This example illustrates one use of fcntl().
 */
#define _OPEN_SYS
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>
#undef _OPEN_SYS
#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 temorarily 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 25
- “sys/types.h” on page 46
- “unistd.h” on page 51
- “close() — Close a File” on page 127
- “dup() — Duplicate an Open File Descriptor” on page 180
- “dup2() — Duplicate an Open File Descriptor to Another” on page 181
- “exec Functions” on page 192
- “fsync() — Write Changes to Direct-Access Storage” on page 299
- “getsockopt() — Get Options Associated with a Socket” on page 370
- “ioctl() — Control Device” on page 417
- “lseek() — Change the Offset of a File” on page 469
- “open() — Open a File” on page 527
- “setsockopt() — Set Options Associated with a Socket” on page 734
- “socket() — Create a Socket” on page 791

fcvt() — Convert Double to String

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().

z/VM C/C++ formatted output functions, including the fcvt() function, convert IEEE floating-point infinity and NaN (Not a Number) argument values to special infinity and NaN floating-point number output sequences. See fprintf Family of Formatted Output Functions 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 183 in all respects other than the number of digits in the return value.

Returned Value

If successful, fcvt() returns the character equivalent of *x* as specified above.

If it is unable to allocate the return buffer or the conversion is unsuccessful, fcvt() returns NULL.

Related Information

- "stdlib.h" on page 42
- "ecvt() — Convert Double to String" on page 183
- "gcvt() — Convert Double to String" on page 311

fdelrec() — Delete a VSAM Record

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*. You can only use fdelrec() after calling fread() and before performing any other operation on that file pointer. For example, if you need to acquire the file position using ftell() or ftell() or fgetpo(), 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 nonstandard 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 (that is, 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.

You can use fdelrec() 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 entry sequenced data sets.

Returned Value

If successful fdelrec() returns a value of 0.

If unsuccessful, fdelrec() returns a nonzero value.

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

- *z/OS: XL C/C++ Programming Guide*
- "stdio.h" on page 40
- "flocate() — Locate a VSAM Record" on page 246
- "fupdate() — Update a VSAM Record" on page 307

fdopen() — Associate a Stream with an Open File Descriptor

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

Format

```
#define _OPEN_SYS
#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 OpenExtensions implementation of buffering, see *z/OS: XL 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).

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 (EOF) (reading and writing).

All of 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, fdopen() returns NULL and sets errno to one of the following values under the conditions described.

EINVAL
The specified mode is incorrect or does not match the mode of the open file descriptor.

EBADF
fildes 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 _OPEN_SYS
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h*gt;
#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 40
- "fileno() — Get the File Descriptor from an Open Stream" on page 242
- "fopen() — Open a File" on page 254
- "open() — Open a File" on page 527

feof() — Test End-of-File Indicator

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 you attempt to read past the EOF. Thus, reading the last character in the file does not turn the flag on. A subsequent read attempt reaches the EOF.

For byte file system files, a simultaneous reader cannot see the extensions automatically. You must use clearerr() 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. Calling rewind(), fsetpos(), fseek(), or clearerr() clears the EOF flag for this stream.

Returned Value

Returns a nonzero value if and only if the EOF flag is set for *stream*; otherwise, feof() returns a value of `0`.

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 40
- "clearerr() — Reset Error and End-of-File" on page 124
- "fseek() — Change File Position" on page 292
- "fsetpos() — Set File Position" on page 294
- "rewind() — Set File Position to Beginning of File" on page 671

ferror() — Test for Read/Write Errors

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

Format

```
#include <stdio.h>

int ferror(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 a parameter that is not valid is given to an I/O function, IBM XL C/C++ for z/VM 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 *stream*; otherwise, ferror() returns a value of 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 (ferror(stream))
   {
      printf("write error\n");
      clearerr(stream);
   }
   if (fclose(stream))
      printf("fclose error\n");
}
```

Related Information

- "stdio.h" on page 40
- "clearerr() — Reset Error and End-of-File" on page 124
- "rewind() — Set File Position to Beginning of File" on page 671

fetch() — Get a Load Module

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 an IBM XL C/C++ for z/VM 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 nonstandard function has two names. One name, the external entry point name, is prefixed with two underscore characters, and the other 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 (that is, the name that begins with two underscore characters __fetch()) or compile with LANGLVL(EXTENDED). When you use LANGLVL(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, you should use the system() library function. Alternatively, when creating the module, you can reset the entry point so that the linkage is provided by IBM XL C/C++ for z/VM.

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 or written in any language conforming to the Language Environment. For IBM XL C/C++ for z/VM modules, the source of the fetched module must, in general, contain `#pragma linkage(..., fetchable)` (the exception is described below). For more information about the need for `#pragma linkage`, see also "fetchep() — Share Writable Static" on page 231.

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 CMS you can accomplish this by using NUCXLOAD on the reentrant module. 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.

Note: 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 POSIX, however, the fetchable, dynamically loaded modules cannot be in the byte file system (BFS). Note also that under POSIX external variables are propagated. For more information about using POSIX support, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Under CMS, the fetch() library function will support the fetching of reentrant modules that are members of a LOADLIB. (For more information on creating a fetchable member of a CMS LOADLIB, see the sections on the LINKLOAD EXEC in *XL C/C++ for z/VM: User's Guide*.) As well, fetch() supports the fetching of reentrant modules that have been loaded into a nucleus extension by a CMS `NUCXLOAD` command. For more information on using NUCXLOAD, see the information about preparing a reentrant program in *z/OS: XL C/C++ Programming Guide* or *z/VM: CMS Commands and Utilities Reference*.

Unless your program is naturally reentrant, each reentrant module has a different copy of writable static. You should 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. Use the prelinker with the option `NOAUTO` if you intend to use LINKLOAD to create a single reentrant object module.
3. Using the object module created in step 2, generate a fetchable member. Under CMS, use the LINKLOAD EXEC to create a member of a CMS load library or use `LOAD/GENMOD` to create a module that can be acted on with `NUCXLOAD`. 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 223 for the program flow of a fetchable module.

Figure 3. Program Flow of a Fetchable Module

To dynamically fetch a set of functions with shared writable static, you can use the library function fetchep(). For more details, see "fetchep() — Share Writable Static" on page 231.

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.

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.

The fetch() function 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(). (For more information, see "fetchep() — Share Writable Static" on page 231.) For example, you cannot pass the address of a function to a fetched routine and invoke it from the fetched routine because the IBM XL C/C++ for z/VM 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.
- Under CMS, you should use unique names for the load module. There must be no existing object modules (of type TEXT) and no existing loadlib members with the same name.

The search order for a CMS LOAD is:

1. Nucleus extensions
2. Shared segments
3. OS load

- A program that invokes fetch() many times without releasing all of the load modules may run out of memory.
- You cannot release() a module that contains any code that is written in another Language Environment-conforming language.

Returned Value

Returns a pointer to a stub that will call the entry point to the fetched load module. If the load fails, fetch() returns NULL.

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() IBM XL C/C++ for z/VM 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(), you must 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 C++ program cannot call fetch(). If you attempt to call fetch() from a z/VM C++ program, the compiler will issue an error message. There are three alternatives to fetch() under z/VM C++:

- You can replace fetch() with DLL (Dynamic Link Library) calls.
- You can provide a C DLL module to fetch modules.
- A z/VM C++ program may statically call a z/VM C function that, in turn, fetches another module.

Refer to "Examples of Alternatives to fetch() Under C++" on page 226 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);
}
```

Commands to compile, link, and run under CMS using LINKLOAD (under Language Environment):

```
GLOBAL TXTLIB  &VSCLIB;
GLOBAL LOADLIB SCEERUN YOURLIB
CC F1A (RENT
CPLINK F1A (NOAUTO
LINKLOAD CPOBJ (LIB YOURLIB MBR F1A NEW

CC F1
CMOD F1
F1
```

Finally, commands to compile, link, and run under CMS using nucleus extensions;

```
GLOBAL TXTLIB EDCBASE IBMLIB CMSLIB
GLOBAL LOADLIB EDCLINK
CC F1A (RENT
CPLINK F1A (NOAUTO
COPYFILE CPOBJ TEXT A F1A TEXT A (REP
CMOD F1A
NUCXDROP F1A
NUCXLOAD F1A

CC F1
CMOD F1
F1
```

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.
```

Next, the compile, Link, and Run under CMS using nucleus extensions:

```
GLOBAL TXTLIB &VSCLIB;
GLOBAL LOADLIB SCEERUN

COBOL2 COB1A
LOAD COB1A (RLDSAVE RESET COB1A
GENMOD COB1A (NOSTR FROM COB1A

CC COB1
CMOD COB1
COB1
```

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

- *z/OS: XL C/C++ Programming Guide*
- "stdlib.h" on page 42
- "fetchep() — Share Writable Static" on page 231
- "release() — Delete a Load Module" on page 657

fetchep() — Share Writable Static

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. The fetchep() function registers an entry point and 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.

You should call fetchep() within a fetched module but not from the same level as the fetch() call. If you call fetchep() 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* you give as input to fetchep() is a function address external to the current module or is an invalid function address, using the pointer returned from the call will result in undefined behavior.

If writable static is required, then you must use this directive:

```
#pragma linkage(entry_point, FETCHABLE)
```

In addition, you must follow the steps for fetching a reentrant module, as described in "fetch() — Get a Load Module" on page 221. 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 nonstandard function has two names. One name, the external entry point name, is prefixed with two underscore characters, and the other 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 fetchep() is __ftchep(), **NOT** __fetchep().

To use this function, you must either invoke the function using its external entry point name (that is, the name that begins with two underscore characters __ftchep(), or compile with LANGLVL(EXTENDED). When you use LANGLVL(EXTENDED) any relevant information in the header is also exposed.

A fetch pointer points to an internal executable control block known as the Fetch Control Block (FECB).

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;
:
}
```

Related Information

- “stdlib.h” on page 42
- “fetch() — Get a Load Module” on page 221
- “release() — Delete a Load Module” on page 657

fflush() — Write Buffer to File

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.

Calling the ungetc() and ungetwc() functions causes fflush() to back up the file position when characters are pushed back. For details, see the ungetc() and ungetwc() functions, respectively. You can set the _EDC_COMPAT environment variable 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 you use fflush() after ungetwc() has pushed a wide character on a text stream, the position will be backed up by one wide character from the position the file was at when you issued ungetwc(). For a wide-oriented binary stream, the position will be backed up based on the number of bytes used to represent the wide character in the ungetc buffer. For this reason, using 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: The _EDC_COMPAT environment variable also changes the behavior of fflush() after ungetwc(). It will discard any pushed-back wide character and leave the position at the point where the ungetwc() was issued. For details on the _EDC_COMPAT environment variable, see *z/OS: XL 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 affect when output data is flushed. For more information about the buffering of C streams, see *z/OS: XL C/C++ Programming Guide*.

stream remains open after the fflush() call. Because a read operation cannot immediately follow or precede a write operation, you can use fflush() to allow exchange between these two modes. You can also use fflush() to refresh the buffer when working with a reader and a simultaneous writer/updater.

Returned Value

If successful, fflush() returns the a value of `0`.

If unsuccessful, fflush() returns EOF. When flushing all open files, a failure to flush any of the files returns EOF. However, flushing will continue on any other open files that fflush() can flush 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 40
- "setbuf() — Control Buffering" on page 703
- "setvbuf() — Control Buffering" on page 742
- "ungetc() — Push Character onto Input Stream" on page 907
- "ungetwc() — Push a Wide Character onto a Stream" on page 908

ffs() — Find First Set Bit in an Integer

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

Format

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

int ffs(int i);
```

General Description

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

Returns the index of the first bit set. If *i* is `0`, ffs() returns a value of `0`.

No errnos are defined for ffs().

Related Information

- "strings.h" on page 44

fgetc() — Read a Character

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`.

The fgetc() function 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition must also occur, 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.

Note: You should use feof() or ferror() to determine whether an error or an EOF condition occurred. An 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 40
- “feof() — Test End-of-File Indicator” on page 219
- “ferror() — Test for Read/Write Errors” on page 220
- “fgetwc() — Get Next Wide Character” on page 239
- “fputc() — Write a Character” on page 272
- “getc() - getchar() — Read a Character” on page 312

fgetpos() — Get File Position

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() 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 backs up the file position indicator 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 907. For details on how ungetwc() affects fgetpos() behavior for a wide-oriented stream, see "ungetwc() — Push a Wide Character onto a Stream" on page 908.

Note: You can set the _EDC_COMPAT environment variable at open time such that fgetpos() will ignore any pushed back characters. For further details on _EDC_COMPAT, see the information about environment variables in *z/OS: XL C/C++ Programming Guide*.

Large Files for HFS

Applications that are compiled with the option LANGLVL(EXTENDED) and also define the Feature Test Macro (FTM) _LARGE_FILES before any headers are included will enable this function to operate on HFS files that are larger than 2 gig-1 in size. File size and offset fields will be enlarged to 63 bits in width so any other function operating on this file will have to be enabled with the same FTM.

Returned Value

If successful, the fgetpos() function returns a value of 0.

If unsuccessful, fgetpos() returns a nonzero value and sets errno to a nonzero value.

Special Behavior for XPG4.2

The fgetpos() function returns a value of 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 40
- “fseek() — Change File Position” on page 292
- “fsetpos() — Set File Position” on page 294
- “ftell() — Get Current File Position” on page 300
- “ungetc() — Push Character onto Input Stream” on page 907
- “ungetwc() — Push a Wide Character onto a Stream” on page 908

fgets() — Read a String from a Stream

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 (\n), or until the end of the stream, whichever comes first. The fgets() function stores the result in *string* and adds a null character (\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`.

The fgets() function 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition must also occur, unless an EOF has been reached.

Returned Value

If successful, fgets() returns a pointer to the *string* buffer.

If unsuccessful, fgets() returns NULL.

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 an intervening flush or reposition operation occurs 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.

Note: You should use ferror() and feof() to determine whether an error or an EOF condition occurred. An 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 return NULL since fgets() would reach EOF without reading any data.

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 40
- “feof() — Test End-of-File Indicator” on page 219
- “ferror() — Test for Read/Write Errors” on page 220
- “fgetc() — Read a Character” on page 235
- “fgetws() — Get a Wide-Character String” on page 241
- “fputs() — Write a String” on page 273
- “gets() — Read a String” on page 364
- “puts() — Write a String” on page 615

fgetwc() — Get Next Wide Character

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().

The fgetwc() function 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition must also occur, 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.

Note: You should use the ferror() and feof() functions to determine whether an error or an EOF condition occurred. An 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 40
- “wchar.h” on page 53

- "fgetc() — Read a Character" on page 235
- "fgetws() — Get a Wide-Character String" on page 241
- "fputwc() — Output a Wide Character" on page 274

fgetws() — Get a Wide-Character String

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*. The fgetws() function does not read any additional wide characters after a new-line wide character (which is retained) or after the EOF; it writes a null wide character written immediately after the last wide character read into the array.

The fgetws() function advances the file position unless an error occurs 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().

The fgetws() function 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition must also occur, unless an EOF has been reached.

Returned Value

If successful, fgetws() returns the new value of *wcs*.

If fgetws() encounters EOF and has not read any wide characters into the array, the contents of the array remain unchanged and fgetws() returns a null pointer. 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, fgetws() returns NULL and sets errno to EILSEQ.

If *n* is less than or equal to 0, it indicates a domain error; fgetws() sets errno 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 an intervening flush or reposition operation occurs 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, you should use either the feof() or the ferror() function. If EOF is reached after fgetws() has read data into the string buffer, fgetws() returns a pointer to the

string buffer to indicate success. A subsequent call would return NULL because fgetws() would reach EOF without reading any data.

Note: You should use ferror() and feof() to determine whether an error or an EOF condition occurred. An 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 40
- “wchar.h” on page 53
- “fgets() — Read a String from a Stream” on page 238
- “fgetwc() — Get Next Wide Character” on page 239
- “fputws() — Output a Wide-Character String” on page 276

fileno() — Get the File Descriptor from an Open Stream

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

Format

```
#define _OPEN_SYS
#include <stdio.h>
int fileno(const FILE *stream);
```

General Description

Returns the file descriptor number associated with a specified C or C++ for z/VM I/O stream. The *stream* argument points to a `FILE` structure controlling a C or C++ for z/VM 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: `stdin`, `stdout`, and `stderr` are macros, not constants.

Returned Value

Returns the file descriptor number associated with an open byte file system stream (that is, one opened with fopen() or freopen()). CMS files are not supported, so fileno() of a CMS file returns a value of -1.

If unsuccessful, fileno() returns a value of -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 a CMS file

Example

CBC3BF21

```
/* CBC3BF21
   This example illustrates one use of fileno().
 */
#define _OPEN_SYS
#include <errno.h>
#include <stdio.h>
#undef _OPEN_SYS

main() {
  FILE *stream;
  char bfs_file[]="./bfs_file", cms_fl[]="//cms.fl";

  printf("fileno(stdin) = %d\n", fileno(stdin));

  if ((stream = fopen(bfs_file, "w")) == NULL)
    perror("fopen() error for BFS file");
  else {
    printf("fileno() of the BFS file is %d\n", fileno(stream));
    fclose(stream);
    remove(bfs_file);
  }

  if ((stream = fopen(cms_fl, "w")) == NULL)
    perror("fopen() error for CMS file");
  else {
    errno = 0;
    printf("fileno() returned %d for the CMS file\n", fileno(stream) );
    printf(" errno=%s\n", strerror(errno));
```

```
    fclose(stream);
    remove(cms_fl);
  }
}
```

Output

```
fileno(stdin) = 0
fileno() of the BFS file is 3
fileno() returned -1 for the CMS file,
errno=Bad file descriptor
```

Related Information

- "stdio.h" on page 40
- "fdopen() — Associate a Stream with an Open File Descriptor" on page 217
- "fopen() — Open a File" on page 254
- "freopen() — Redirect an Open File" on page 280
- "open() — Open a File" on page 527

fldata() — Retrieve File Information

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 specified in fopen() or freopen() in *filename* and other information in the structure *info*. 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. 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 nonstandard 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 (that is, 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 the information about opening files in *z/OS: XL C/C++ Programming Guide*.

If *fldata* is the first reference to a standard stream, a call to the fldata() function opens the stream.

Table 10 on page 245 describes the fields in the fldata_t data structure. For further details, see *z/OS: XL C/C++ Programming Guide*.

Special Behavior for POSIX C

Under OpenExtensions, if an exec occurred 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. For more

information about using POSIX support, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, fldata() returns a value of `0`.

If unsuccessful, fldata() returns a nonzero value.

Table 10. 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.
`__recfmPO: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 a byte file system 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`.
`__reserve2:9`	`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`.

Table 10. Elements Returned in fldata_t Data Structure (continued)		
Element	**Data Type**	**General Description**
`__vsamkeylen`	`unsigned long`	Length of VSAM key (if any).
`__vsamRKP`	`unsigned long`	Key position.
`__dsname`	`char *`	Real file name of file opened by ddname if the file is a DASD data set or memory file; otherwise, this field is NULL. The `char *__dsname` field is allocated internally by the library functions and must be saved before the next call to the fldata() function.
`__reserve4`	`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 40
- “fopen() — Open a File” on page 254
- “freopen() — Redirect an Open File” on page 280

flocate() — Locate a VSAM Record

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 nonstandard 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 (that is, 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 *key* argument points to a key used for positioning. The *key_len* argument specifies the length of the search key; its 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.

The *options* argument specifies the position options described in Table 11 on page 247.

Table 11. Position Options Parameter for flocate()

Parameter	Function
`__KEY_FIRST`	Positions to the first record in the file. Subsequent reads are in the forward direction. *key* and *key_len* are ignored.
`__KEY_LAST`	Positions to the last record in the file. Subsequent reads are in backward order. *key* and *key_len* 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. You should *not* use this option with RRDS. 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`. You should *not* use this option with KSDS because the RBA of a given record may change over time due to inserts, deletions, or updates of other records.

Table 11. Position Options Parameter for flocate() (continued)	
Parameter	**Function**
__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. You should *not* use this option with RRDS. 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. You should *not* use this option with KSDS because the RBA of a given record may change over time due to 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

If successful, flocate() returns a value of 0. If a record was not found or the position is beyond the EOF, then flocate() returns EOF.

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

- For information on performing VSAM I/O operations, see *z/OS: XL C/C++ Programming Guide*.
- "stdio.h" on page 40
- "fdelrec() — Delete a VSAM Record" on page 216
- "fgetpos() — Get File Position" on page 236
- "fseek() — Change File Position" on page 292
- "fsetpos() — Set File Position" on page 294
- "ftell() — Get Current File Position" on page 300
- "fupdate() — Update a VSAM Record" on page 307

floor(), floorf(), floorl() — Round Down to Integral Value

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

Format

```
#include <math.h>

double floor(double x);
float floor(float x);                  /* C++ only */
long double floor(long double x);      /* C++ only */
float floorf(float x);
long double floorl(long double x);
```

General Description

Calculates the largest integer that is less than or equal to *x*.

Note: These functions work in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 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 32
- "ceil(), ceilf(), ceill()— Round Up to Integral Value" on page 107

• "fmod(), fmodf(), fmodl() — Calculate Floating-Point Remainder" on page 250

fmod(), fmodf(), fmodl() — Calculate Floating-Point Remainder

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

Format

```
#include <math.h>

double fmod(double x, double y);
float fmod(float x, float y);                    /* C++ only */
long double fmod(long double x, long double y);  /* C++ only */
float fmodf(float x, float y);
long double fmodl(long double x, long 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: These functions work in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

Returned Value

If *y* is 0, fmod() returns a value of 0. However, errno is neither set to ERANGE to set an out-of-range error, nor to EDOM to set an out-of-domain error.

If x/y causes an overflow, then fmod() returns a value of 0.

Special Behavior for IEEE

If successful, the function returns the floating-point remainder of x/y.

If *y* is 0, the function sets errno to EDOM and returns NaNQ. No other errors will occur.

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 32

fmtmsg() — Display a Message in the Specified Format

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

Displays messages in a specified format instead of in the traditional format of the 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 *classification* component is not part of a display message, 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. You may use any one identifier from a subclass in combination with a single identifier from a different subclass. You should not use two or more identifiers from the same subclass together, with the exception of identifiers from the display subclass. (You may use both display subclass identifiers so that messages can display 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 will display. Identifiers are: MM_PRINT to display the message on the standard error stream and MM_CONSOLE to display the message on the system console. You may use one or both identifiers.

Status Subclassifications

Indicates whether the application will recover from the condition. Identifiers are: MM_RECOVER (recoverable) and MM_NRECOV (non-recoverable).

An additional identifier, MM_NULLMC, indicates that you have not supplied a classification component for the message.

label

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 __MM_MXLABELLN defines the maximum length of *label*.

severity
Indicates the seriousness of the condition. Identifiers for the levels of severity are:

MM_HALT
Indicates that the application has encountered a severe fault and is halting. Produces the string "HALT."

MM_ERROR
Indicates that the application has detected a fault. Produces the string "ERROR."

MM_WARNING
Indicates a condition that is out of the ordinary or might be a problem and should be watched. Produces the string "WARNING."

MM_INFO
Provides information about a condition that is not in error. Produces the string "INFO."

MM_NOSEV
Indicates that no severity level is supplied for the message.

Other
Provides an unknown severity. Produces the string "SV=n", where n is the *severity* value specified.

text
Describes the error condition that produced the message. The character string is not limited to a specific size. If the character string is empty, the text produced is unspecified.

action
Describes the first step to be taken in the error-recovery process. The fmtmsg() function precedes the action string with the prefix "TO FIX." The action string is not limited to a specific size.

tag
References online documentation for the message. The tag should include the label and a unique identifying number. A sample tag is "XSI:cat:146". The constant __MM_MXTAGLN defines the maximum length of *tag*.

The MSGVERB environment variable (for message verbosity) tells fmtmsg() which message components 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 its 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, or its value is the null string, or its value is not in the correct format, or 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

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 264

fnmatch() — Match File Name or Path Name

Standards / Extensions	C or C++	Dependencies
XPG4 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 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 FNM_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 FNM_PATHNAME is set and either of these characters would match a slash, the function returns FNM_ESLASH. If the FNM_PATHNAME flag is not set, the slash character is treated as an ordinary character.

If FNM_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 FNM_NOESCAPE is set, a backslash character will be treated as an ordinary character.

If FNM_PERIOD is set in *flags*, then a leading period in *string* will match a period in *pattern*. The location of "leading" is indicated by the value of FNM_PATHNAME:

- If FNM_PATHNAME is set, a period is “leading” if it is the first character in *string* or if it immediately follows a slash.
- If FNM_PATHNAME is not set, a period is “leading” only if it is the first character of *string*.

If FNM_PERIOD is not set, then no special restrictions are placed on matching a period. If FNM_PERIOD is set, and a pattern wildcard would match a leading period as defined by the above rules, then fnmatch() returns FNM_EPERIOD.

Returned Value

If *string* matches *pattern*, fnmatch() returns a value of zero. If no match is made, 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 26
- “glob() — Generate Path names Matching a Pattern” on page 389
- “wordexp() — Perform Shell Word Expansions” on page 972

fopen() — Open a File

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 12 on page 255 and Table 13 on page 256.

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 filenaming rules exist, which allow you to create an application that references both VM and BFS files specifically. For details on how fopen() determines the type of file from the *filename* and *mode* strings, see the information about opening files in *z/OS: XL C/C++ Programming Guide*.

File Mode

Restriction: When running with POSIX(OFF) and specifying a mode parameter that includes `t`, for example, `rt`, `rt+`, `r+t`, `wt`, `wt+`, `w+t`, `at`, `at+`, or `a+t`, the fopen() request will fail with a message indicating a non-valid mode was specified.

Table 12. 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. 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. fopen() 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. If the file already exists, its contents are destroyed.
a+	Open a text file in append mode for reading or updating at the end of the file (fopen() 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 `wb` mode is specified for a ddname that has `DISP=MOD`, the behavior is the same as if ab had been specified. 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 (fopen() 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. fopen() 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 `w+b` (or `wb+`) mode is specified for a ddname that has `DISP=MOD`, the behavior is the same as if ab+ had been specified. 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 (fopen() 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. fopen() 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 a VM 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 C or C++ for z/VM, 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. For more information about the byte stream model, see *z/OS: XL C/C++ Programming Guide*.

C and C++ for z/VM have a *Record I/O file* extension. These files are binary in nature—no data interpretation—and require the additional qualifier: `type=record`. For more information about writing to record I/O files, see *z/OS: XL C/C++ Programming Guide*.

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 13. 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 *z/OS: XL C/C++ Programming Guide* .
`byteseek`	Indicator to allow byte seeks for a binary file. For more information, see the ftell() and fseek() functions.
`lrecl=`*value*	Specifies the length, in bytes, for fixed-length records and the maximum length for variable-length records. To check whether your `lrecl` parameter is valid and is within its limits, see *z/OS: XL C/C++ Programming Guide*.
`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

Table 13. Keyword Parameters for File Mode (continued)	
Parameter	**Description**
`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
`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
space	Space attributes for MVS™ data sets. Within the parameter, you cannot have any imbedded blanks. This parameter is ignored under CMS.
`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.
`os`	Used on CMS to open for reading a data set by a ddname that is on an OS disk.
`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 byte file system file names (for details, see *z/OS: XL C/C++ Programming Guide*).
`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.

Large Files for HFS

Applications that are compiled with the option LANGLVL(EXTENDED) and also define the Feature Test Macro (FTM) _LARGE_FILES before any headers are included will enable this function to operate on HFS files that are larger than 2 gig-1 in size. File size and offset fields will be enlarged to 63 bits in width so any other function operating on this file will have to be enabled with the same FTM.

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.

Special Behavior for Large Files for HFS

The following is the possible value of errno:

Error Code
Description

EOVERFLOW
The named file is a regular file and the size of the file cannot be represented correctly in an object of type `off_t`.

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

- “stdio.h” on page 40
- “fclose() — Close File” on page 206
- “fldata() — Retrieve File Information” on page 244
- “freopen() — Redirect an Open File” on page 280

fork() — Create a New Process

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

Attention: On VM/ESA this is not a full function fork(). You should only use this function during a porting exercise until you can convert to something like spawn() or threading functions.

Format

```
#define _POSIX_SOURCE
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

Recommendation: Although POSIX.1 does not require the inclusion of <sys/types.h>, XPG4 has it as an optional header; therefore, you should include it for portability.

General Description

Creates a new process. The new process (the *child process*) is a new CMS process running in the same virtual machine as 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 886.
- 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.

You can use CMS memory files from an OpenExtensions application. Using an exec function from the program clears a memory file when the process address space is cleared.

Special Behavior for OpenExtensions Services

If the specified file name represents an external link, the program is loaded from the caller's CMS 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 loadbfs service. In other cases, the program is loaded from the BFS.

Restrictions:

1. The fork() call in the child process must be followed by an exec().
2. You must issue the CMS Command OPENVM SET FORK ON before running an application using fork(). If the CMS FORK flag is not turned on, the application will receive a return value of -1, a return code of ENOSYM, and a reason code of JRFunctNotSupported.
3. You must run the application as a POSIX(ON) application. If the Language Environment POSIX flag is not turned on, the application will receive a return value of -1, a return code of ENOSYS, and a reason code of JRFunctNotSupported.
4. The child process is *not* allowed to issue any function that will cause the child process to be blocked. Doing so will result in an X'AE6' abnormal end code, for example, a pipe read() or a pause(), before the child issues the exec(). Any attempt to exit the child process before the exec() is issued will result in an X'AE5' abnormal end code.
5. Any local variables in the application that are changed in the child process before the exec() is issued will be changed in the parent process as well. This is due to the fact that the child and parent processes are still using the same program storage. The exec() function causes the child process to begin using its own program storage.
6. Any global and/or environment variables in the application that are changed in the child process before the exec() is issued will be changed in the parent process as well. This is due to the fact that the child and parent processes are still using the same program storage. The exec() function causes the child process to begin using its own program storage.

Returned Value

If successful, fork() returns a value of `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, returns a value of -1 to the parent, and sets errno to one of the following values under the conditions described:

EAGAIN
Insufficient resources are available to create another process, or the process has already reached the maximum number of processes you can run.

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());
    "execl(/bin/true);"
  }
```

```
  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 0
```

Related Information

- “sys/types.h” on page 46
- “alarm() — Set an Alarm” on page 76
- “exec Functions” on page 192
- “fcntl() — Control Open File Descriptors” on page 207
- “kill() — Send a Signal to a Process” on page 439
- “putenv() — Change or Add an Environment Variable” on page 614
- “semop() — Perform Semaphore Operations” on page 692
- “shmat() — Attach Shared Memory Segment” on page 743
- “sysconf() — Determine System Configuration Options” on page 856
- “times() — Get Process and Child Process Times” on page 886
- “wait() — Wait for a Child Process to End” on page 927

fpathconf() — Determine Configurable Path Name Variables

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

Format

```
#define _OPEN_SYS
#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*).

The fpathconf() function 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. The fpathconf() function 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). The fpathconf() function 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, fpathconf() returns an errno of EINVAL (see description below).

_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; *pathname* must refer to a character special file for a terminal. For details, see the callable service tcsetattr().

_PC_ACL
Returns 1 if an access control mechanism is supported by the security product for the file identified by the file descriptor.

_PC_ACL_ENTRIES_MAX
Returns the maximum number of ACL entries in an ACL for the file or directory identified by the file descriptor.

Returned Value

If a particular variable has no limit, fpathconf() returns a value of -1 but does not change errno.

If successful, fpathconf() returns the value of the variable requested in *varcode*.

If unsuccessful, fpathconf() returns a value of -1 and sets errno to one of the following values under the conditions described.

EBADF
fildes is not a valid open file descriptor.

EINVAL
varcode is not a valid variable code, or the given variable cannot be associated with the specified file.

- If *varcode* refers to MAX_CANON, MAX_INPUT, or _POSIX_VDISABLE, and *pathname* does not refer to a character special file, fpathconf() returns a value of -1 and sets errno to EINVAL.
- If *varcode* refers to NAME_MAX, PATH_MAX, or POSIX_NO_TRUNC, and *pathname* does not refer to a directory, fpathconf() returns the requested information.
- If *varcode* refers to PC_PIPE_BUF and *pathname* refers to a pipe or a FIFO, the value returned applies to the referenced object itself. If *pathname* refers to a directory, the value returned applies to any FIFOs that exist or can be created within the directory. If *pathname* refers to any other type of file, fpathconf() 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 _OPEN_SYS
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _OPEN_SYS
#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 51

- "open() — Open a File" on page 527
- "pathconf() — Determine Configurable Path Name Variables" on page 533

fprintf() - printf() - sprintf() — Format and Write Data

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.

The fprintf() and printf() functions 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition must also occur unless an EOF has been reached.

The *format-string* consists of ordinary characters, escape sequences, and format specifications. The ordinary characters are copied in order of their appearance. Format specifications, beginning with a percent sign (%), 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.

When the *format-string* includes the use of the optional prefix *ll* to indicate the size expected is a long long datatype then the corresponding value in the argument list should be a long long datatype if correct output is expected.

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.

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, string, 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 14 on page 265 are used for the justification of output and printing of signs, blanks, decimal points, octal, and hexadecimal prefixes, and the semantics for `wchar_t` precision unit.

Note: More than one *flag* can appear in a format specification. This is an optional field.

Table 14. Flag Characters for fprintf() Family

Flag	Meaning	Default
-	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 (-).
blank(' ')	Prefix the output value with a blank if the output value is signed and positive. The + flag overrides the *blank* 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. The decimal point is sensitive to the LC_NUMERIC category of the same current locale.	Decimal point appears only if digits follow it.
	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 `ls` format, the # flag causes precision to be measured in wide characters.	Precision indicates the maximum number of bytes to be output.

Table 14. Flag Characters for fprintf() Family (continued)

Flag	Meaning	Default
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, 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.

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.

The interpretation of the *precision* value and the default when the *precision* is omitted depend upon the *type*, as shown in Table 15 on page 266.

Table 15. Precision Argument in fprintf() family

Type	Meaning	Default
i d u o x X	*Precision* specifies the minimum number of digits to be output. If the number of digits in the argument is less than *precision*, the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds *precision*.	If *precision* is 0 or omitted entirely, or if the period (.) appears without a number following it, the *precision* is set to 1.
f e E	*Precision* 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 *precision* is 6. If *precision* is 0 or the period appears without a number following it, no decimal point is output.

Table 15. Precision Argument in fprintf() family (continued)

Type	Meaning	Default
g G	*Precision* specifies the maximum number of significant digits output.	All significant digits are output.
c	No effect.	The character is output.
lc	No effect.	The wide character is output.
s	*Precision* specifies the maximum number of characters to be output. Characters in excess of *precision* are not output.	Characters are output until a null character is encountered.
S ls	*Precision* specifies the maximum number of bytes to be output. Bytes in excess of *precision* are not output; however, multibyte 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.

ll
A prefix with the integer types d, `i`, o, u, x, X means the integer is 64 bits long.

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 16 on page 267 shows the meaning of the type characters used in the precision argument.

Table 16. 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.

Table 16. Type Characters and their Meanings (continued)

Type	Argument	Output Format
f	Double	Signed value having the form [-]*dddd.dddd*, where *dddd* 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. The decimal point is sensitive to the LC_NUMERIC category of the current locale.
e	Double	Signed value having the form [-]*d.dddd*e[*sig n*]*ddd*, where *d* is a single-decimal digit, *dddd* is one or more decimal digits, *ddd* is 2 or more decimal digits, and *sign* is + or -.
E	Double	Identical to the *e* format, except that E introduces the exponent, not e.
g	Double	Signed value output in f or e format. The *g* format is used only when the exponent of the value is less than -4 or greater than *precision*. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it.
G	Double	Identical to the g format, except that G introduces the exponent (where appropriate), not *g*.
D(n,p)	Decimal type argument	Fixed-point value consisting of an optional sign (+ or -); a series of one or more decimal digits possibly containing a decimal point. Note that packed decimal is supported under C only, not C++.
c lc	Character Wide Character	Single character. The argument of wchar_t type is converted to an array of bytes representing a multibyte character as if by call to wcrtomb().
s ls	String Wide String	Characters output up to the first null character (\0) or until *precision* is reached. The argument is a pointer to an array of wchar_t 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 wcrtomb(), with the conversion state described by the mbstate_t object initialized to 0. The result written out will not include the terminating null character. If no precision is specified, the array contains a null wide character. If a precision is specified, its sets the maximum number of characters written, including shift sequences. A partial multibyte character cannot be written.
n	Pointer to integer	Number of characters successfully output so far to the *stream* 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 newline), 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 newline), 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 <= n <= 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 z/VM 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(n) or NANQ(n) 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.

Examples

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("1234567890123%n4567890123456789\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);
}
```

CBC3BF30 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.
 */
#include <stdio.h>
#include <decimal.h>

decimal(10,2) pd01 = -12.34d;
decimal(12,4) pd02 = 12345678.9876d;
decimal(31,10) pd03 = 1234567890135792468 01.9876543210d;

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 %%20.2D(12,4)  = %20.2D(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);
}
```

CBC3BF31 Output

```
pd01 %D(10,2)      = -12.34
pd02 %D( 12 , 4 )  = 12345678.9876
pd01 %010.2D(10,2) = -000012.34
pd02 %20.2D(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);
}
```

CBC3BF32 Output

```
string:
baltimore
l
35
1.732051

character count = 24
```

Related Information

- For information about locales and character sets, see *z/OS: XL C/C++ Programming Guide*.
- “locale.h” on page 30
- “stdio.h” on page 40
- “wchar.h” on page 53
- “fscanf() - scanf() - sscanf() — Read and Format Data” on page 283
- “localeconv() — Query Numeric Conventions” on page 454
- “setlocale() — Set Locale” on page 721
- “wcrtomb() — Convert a Wide Character to a Multibyte Character” on page 933

fputc() — Write a Character

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`, writes *c* to the output *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`.

The fputc() function 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition must also occur, 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 40
- "fgetc() — Read a Character" on page 235
- "putc() - putchar() — Write a Character" on page 613

fputs() — Write a String

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 about truncation, see *z/OS: XL C/C++ Programming Guide*.

The fputs() function is not supported for files opened with `type=record`.

The fputs()function 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition must also occur, unless an EOF has been reached.

Returned Value

If successful, fputs() returns the number of bytes written.

If unsuccessful, fputs() returns EOF.

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 '/0' 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 40
- "fgets() — Read a String from a Stream" on page 238
- "gets() — Read a String" on page 364
- "puts() — Write a String" on page 615

fputwc() — Output a Wide Character

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, *wc*, to a multibyte character, writes it to the output *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 LC_CTYPE category of the current locale affects the behavior of this wide-character function. Using non-wide-character functions with fputwc() results in undefined behavior.

The fputwc() function 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition kust also occur, 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, 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.

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, fputwc() returns WEOF and sets the error indicator for the stream. If an encoding error occurs during conversion from the wide character to a multibyte character, fputwc() returns WEOF and stores the value of the macro EILSEQ in errno.

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%lx\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 40
- “wchar.h” on page 53

fputws() — Output a Wide-Character String

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, *wcs*,to a multibyte character string and writes it to the *stream* as a multibyte character string. It does not write the terminating null byte.

The LC_CTYPE category of the current locale affects the behavior of this wide-character function. Using non-wide-character functions with fputws() results in undefined behavior.

The fputws() function 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition must also occur, unless an EOF has been reached.

Returned Value

If successful, fputws() returns a non-negative value.

If a *stream* error occurs, fputws() returns a value of -1 and sets the error indicator for the stream. If an *encoding* error occurs, fputws() returns a value of -1 and stores the value of the macro EILSEQ in errno. 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 40
- “wchar.h” on page 53

fread() — Read Items

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 an error occurs 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, you should 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.

The fread() function 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition must also occur, unless an EOF has been reached.

Returned Value

Returns the number of complete items successfully read. If *size* or *count* is `0`, fread() returns a value of `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.

Note: You should use ferror() and feof() to determine whether an error or an EOF condition occurred. An 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+1] = '\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 40
- “fopen() — Open a File” on page 254
- “freopen() — Redirect an Open File” on page 280
- “fwrite() — Write Items” on page 308

free() — Free a Block of Storage

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*, 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. The free() function will not set *ptr* to NULL.

This function is also available to C applications in a stand-alone Systems Programming Environment.

Attention: Attempting to free a block of storage not allocated with calloc(), malloc(), realloc(), or previously freed storage, can affect the subsequent reservation 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

The free() function has 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

- For information on using the System Programming C Facilities, see *z/OS: XL C/C++ Programming Guide*.
- “spc.h” on page 39
- “stdlib.h” on page 42
- “calloc() — Reserve and Initialize Storage” on page 103
- “malloc() — Reserve Storage Block” on page 477
- “realloc() — Change Reserved Storage Block Size” on page 637

freeaddrinfo() — Free Addrinfo Storage

Standards

Standards / Extensions	C or C++	Dependencies
RFC2553	both	

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <sys/socket.h>
#include <netdb.h>

void *freeaddrinfo(struct addrinfo *ai);
```

General Description

The freeaddrinfo() function frees one or more addrinfo structures returned by getaddrinfo(), along with any additional storage associated with those structures. *ai* points to the first addrinfo structure in a list of one or more addrinfo structures. If the ai_next field of the structure is not null, the entire list of structures is freed.

Returned Value

This function does not return a value.

Related Information

- "connect() — Connect a Socket" on page 138
- "gai_strerror() — Address and Name Information Error Description" on page 309
- "getaddrinfo() — Get Address Information" on page 313
- "socket() — Create a Socket" on page 791
- "netdb.h" on page 34
- "sys/socket.h" on page 46

freopen() — Redirect an Open File

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*, opens the file specified by *filename*, and then associates the stream with it.

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. For a description of the *mode* parameter, see "fopen() — Open a File" on page 254.

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 *z/OS: XL 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 Language Environment MSGFILE, which is the default for `stderr`
- Byte File System files.
- HFS files

Note: Using the empty string method is included in the SAA C definition, but not in the ANSI C standard.

Large Files for HFS

Applications that are compiled with the option LANGLVL(EXTENDED) and also define the Feature Test Macro (FTM) _LARGE_FILES before any headers are included will enable this function to operate on HFS files that are larger than 2 gig-1 in size. File size and offset fields will be enlarged to 63 bits in width so any other function operating on this file will have to be enabled with the same FTM.

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.

Special Behavior for Large Files for HFS

The following is the possible value of errno:

Error Code
Description

EOVERFLOW
The named file is a regular file and the size of the file cannot be represented correctly in an object of type off_t.

Example

This example illustrates the IBM XL C/C++ for z/VM 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 40
- "fclose() — Close File" on page 206
- "fopen() — Open a File" on page 254

frexp() — Extract Mantissa and Exponent of the Floating-Point Value

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

Format

```
#include <math.h>

double frexp(double x, int *expptr);
float frexp(float x, int *expptr);                /* C++ only */
long double frexp(long double x, int *expptr);    /* C++ only */
float frexpf(float x, int *expptr);
long double frexpl(long 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 *expptr* points.

Note: These functions work in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

Returned Value

Returns the normalized fraction *m*. If *x* is `0`, frexp() returns a value of `0` for both the fraction and exponent. The fraction has the same sign as *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 32
- "ldexp(), ldexpf(), ldexpl() — Multiply by a Power of Two" on page 445
- "modf() — Extract Fractional and Integral Parts of a FLoating-Point Value" on page 507

fscanf() - scanf() - sscanf() — Read and Format Data

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. You cannot use the fscanf() function for a file opened with `type=record`.

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. You *cannot* use scanf() 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.

The fscanf() and scanf() functions 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition must also occur, 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 that are introduced by the percent sign (%). 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 too many arguments are included for the conversion specifications, the extra arguments are evaluated but otherwise ignored. The results are undefined if not enough arguments are provided for the conversion specifications.

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).

A detailed discussion of each field of the format specification follows:

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, IBM XL C/C++ for z/VM permits a new conversion specifier, it could match a section of your format string, be interpreted incorrectly, and result in undefined behavior. For a list of conversion specifiers, see Table 17 on page 285.

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 should use the `long` version of the following *conversion specifier*, while the prefix h indicates that you should use the `short` version. 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). You can use the `l` and h modifiers with the d, `i`, o, x, and u *conversion specifiers*. You can also use the `l` modifier with the e, `f`, and g *conversion specifiers*. You can use the L modifier with the e, `f` and g *conversion specifiers*.

Note: 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 17 on page 285.

Table 17. Conversion Specifiers in fscanf() and scanf()

Conversion Specifier	Type of Input Expected	Type of Argument
d	Decimal integer	Pointer to `int`
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`

Table 17. Conversion Specifiers in fscanf() and scanf() (continued)

Conversion Specifier	Type of Input Expected	Type of Argument
Note: 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 preceding 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 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 uppercase or lowercase, and then a white-space character (space, tab, or newline), 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 uppercase or lowercase, then an optional NaN ordinal sequence, and then a white-space character (space, tab, or newline), 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 <= n <= 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. Note that packed decimal is supported under C only.	Pointer to decimal

Table 17. Conversion Specifiers in fscanf() and scanf() (continued)

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. 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 mbrtowc() 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. `lc` is the multibyte character constant.	Pointer to `char` large enough for input field. `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. 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. `ls` expects a multibyte string constant.	Pointer to character array large enough for input field, plus a terminating null character (`\0`) that is automatically appended. `ls` uses a pointer to `wchar_t` string.
n	No input read from *stream* or buffer.	Pointer to `int`, into which is stored the number of characters successfully read from the *stream* 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 17. Conversion Specifiers in fscanf() and scanf() (continued)

Conversion Specifier	Type of Input Expected	Type of Argument
[ ]	A non-empty sequence of bytes from a set of expected bytes (the *scanset*), which form the conversion specification. The conversion continues reading bytes until a failure to match or until an input failure. Consider the following situations: [^*bytes*]. In this case, the scanset contains all bytes that do not appear between the circumflex and the right square bracket. [ ]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, *not*]abc) [a–z] The – is in the scanset; the characters b through y are *not* in the scanset. 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.	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.

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, you should 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), you should use the specification %*a*c, 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. They might stop reading a particular input field 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 an input failure occurs before any conversion or if EOF is reached before any conversion. Thus a returned value of `0` means that no fields were assigned: a matching failure occurred before any conversion. Also, if an input failure occurs, the file error indicator is set, which is not the case for a matching failure.

Note: You should use ferror() and feof() to determine whether an error or an EOF condition occurred. An 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);
   }
}
```

CBC3BF42 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);
      }
}
```

CBC3BF43 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);
}
```

CBC3BF44 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);
}
```

CBC3BF46 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);
}
```

CBC3BS32 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

- For information about locales and character sets, see *z/OS: XL C/C++ Programming Guide*.
- "locale.h" on page 30
- "stdio.h" on page 40
- "fprintf() - printf() - sprintf() — Format and Write Data" on page 264
- "localtime() — Convert Time and Correct for Local Time" on page 456
- "setlocale() — Set Locale" on page 721

fseek() — Change File Position

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 907 and "ungetwc() — Push a Wide Character onto a Stream" on page 908.

The _EDC_COMPAT environment variable causes fseek() to ignore the effects of ungetc() or ungetwc(). For more information on environment variables, see *z/OS: XL C/C++ Programming Guide*.

Binary Streams

ANSI states that binary streams use relative byte offsets for both ftell() and fseek(). Under IBM XL C/C++ for z/VM, 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 SEEKSET.

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 *z/OS: XL 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 byte file system (BFS) files, for which the file is extended with nulls only if you subsequently write new data. This is true also under POSIX, using BFS files, where the file is extended with nulls only 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 BFS 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().

Note: You should not calculate your own position since this may cause 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.

For wide-oriented streams, all the above restrictions apply.

Recommendation: You should not reposition within a wide-oriented file or perform updates 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 and 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 *z/OS: XL 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 fseek() or fsetpos() is not valid, the call is treated as a flush and the ungetc characters are discarded.

Large Files for HFS

Applications that are compiled with the option LANGLVL(EXTENDED) and also define the Feature Test Macro (FTM) _LARGE_FILES before any headers are included will enable operations on HFS files that are larger than 2 gig-1 in size. When this choice has been made then fseek() should be replaced by the neutral function fseeko(). fseek() will fail if it is used against a large file.

Returned Value

If successful, fseek() returns a value of `0`.

If unsuccessful, fseek() returns a nonzero value. On devices that cannot seek, such as terminals and printers, the returned value is nonzero.

Special Behavior for XPG4.2

The fseek() function returns a value of -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 40
- “ftell() — Get Current File Position” on page 300
- “ungetc() — Push Character onto Input Stream” on page 907
- “ungetwc() — Push a Wide Character onto a Stream” on page 908

fsetpos() — Set File Position

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.

Recommendation: You should not reposition within a wide-oriented file or perform updates 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 and 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 successful, fsetpos() returns a value of 0.

If unsuccessful, fsetpos() returns a nonzero value and sets errno.

Special Behavior for XPG4.2

The fseek() function returns a value of -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 40
- “fgetpos() — Get File Position” on page 236
- “ftell() — Get Current File Position” on page 300
- “rewind() — Set File Position to Beginning of File” on page 671
- “ungetc() — Push Character onto Input Stream” on page 907

- "ungetwc() — Push a Wide Character onto a Stream" on page 908

fstat() — Get Status Information about a File

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

Format

```
#define _OPEN_SYS
#include <sys/stat.h>

int fstat(int fildes,struct stat *info);
```

General Description

Gets status information about the file specified by the open file descriptor *fildes* and stores it in the area of memory indicated by *info*. 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 803.

For external links to CMS files, fstat() functions as usual with the exception of how it determines the file size. For fixed format files, the file size returned is the product of the number of records and the logical record length. For variable format files, the size returned is the same as above if the file does not exceed one data block in length. If it is longer, the size is calculated as the product of the number of data blocks and the block size.

Large Files for HFS

Applications that are compiled with the option LANGLVL(EXTENDED) and also define the Feature Test Macro (FTM) _LARGE_FILES before any headers are included will enable this function to operate on HFS files that are larger than 2 gig-1 in size. File size and offset fields will be enlarged to 63 bits in width so any other function operating on this file will have to be enabled with the same FTM.

Returned Value

If successful, fstat() returns a value of `0`.

If unsuccessful, fstat() returns a value of -1 and sets errno to one of the following values under the conditions described:

EBADF
fildes is not a valid open file descriptor.

EINVAL
info contains a null.

EIO
Added for XPG4.2: An I/O error occurred while reading from the file system.

EOVERFLOW
Added for XPG4.2: 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 _OPEN_SYS
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#undef _OPEN_SYS
#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:   11
   mode:   03000080
  links:   1
    uid:   25
    gid:   500
created:   Fri Jun 16 16:03:16 1995
```

Related Information

- “sys/stat.h” on page 46
- “sys/types.h” on page 46
- “fcntl() — Control Open File Descriptors” on page 207
- “lstat() — Get Status of File, External Link, or Symbolic Link” on page 471
- “open() — Open a File” on page 527
- “stat() — Get File Information” on page 803

fstatvfs() — Get File System Information

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

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/statvfs.h>
int fstatvfs(int fildes, struct statvfs *fsinfo);
```

General Description

Obtains information about the file system containing the file referenced by *fildes* 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 806.

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 a value of `0`.

If unsuccessful, fstatvfs() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EBADF
fildes 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 46
- "chmod() — Change the Mode of a File or Directory" on page 117
- "chown() — Change the Owner or Group of a File or Directory" on page 120

- “creat() — Create a New File or Rewrite an Existing One” on page 146
- “dup() — Duplicate an Open File Descriptor” on page 180
- “fcntl() — Control Open File Descriptors” on page 207
- “link() — Create a Link to a File” on page 450
- “mknod() — Make a FIFO or Charcter Special File” on page 501
- “open() — Open a File” on page 527
- “pipe() — Create an Unnamed Pipe” on page 539
- “read() — Read From a File ” on page 628
- “time() — Determine Current Time” on page 885
- “unlink() — Remove a Directory Entry” on page 910
- “utime() — Set File Access and Modification Times” on page 912
- “write() — Write Data on a File or Socket” on page 975

fsync() — Write Changes to Direct-Access Storage

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

Format

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

int fsync(int fildes);
```

General Description

Transfers all data for the file indicated by the open file descriptor *fildes* to the storage device associated with *fildes*. The fsync() function does not return until the transfer has completed or until an error is detected.

Returned Value

If successful, fsync() returns a value of 0.

If unsuccessful, fsync() returns a value of -1 and sets errno to one of the following values under the conditions described:

EBADF
fildes is not a valid open file descriptor.

EEXTLINK
fildes is not supported for external links.

EINVAL
The file is not a regular file.

EROFS
Refers to a read-only file system.

Example

CBC3BF48

```
/* CBC3BF48 */
#define _OPEN_SYS
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _OPEN_SYS
#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 51
- “open() — Open a File” on page 527
- “write() — Write Data on a File or Socket” on page 975

ftell() — Get Current File Position

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 IBM XL C/C++ for z/VM, 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 *z/OS: XL C/C++ Programming Guide*.

Text Streams

The ftell() function 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 907 and "ungetwc() — Push a Wide Character onto a Stream" on page 908.

Large Files for HFS

Applications that are compiled with the option LANGLVL(EXTENDED) and also define the Feature Test Macro (FTM) _LARGE_FILES before any headers are included will enable operations on HFS files that are larger than 2 gig-1 in size. When this choice has been made, then ftell() should be replaced by the neutral function ftello(); ftell() will fail if it is used against a large file.

Returned Value

If successful, ftell() returns the calculated value.

If unsuccessful, ftell() returns a value of -1 and sets errno to a positive value.

Special Behavior for XPG4.2

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

Error Code
Description

ESPIPE
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 40
- "fgetpos() — Get File Position" on page 236
- "fopen() — Open a File" on page 254
- "fseek() — Change File Position" on page 292
- "fsetpos() — Set File Position" on page 294

ftime() — Get the Date and Time

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

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

Returns a value of `0` unless overflow occurs, in which case it returns a value of -1.

Note: 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`.

Related Information

- "limits.h" on page 28
- "sys/timeb.h" on page 46
- "ctime() — Convert Time to a Character String" on page 153
- "time() — Determine Current Time" on page 885

ftok() — Generate an Interprocess Communication (IPC) Key

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

Format

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

key_t ftok(const char *path, int id);
```

General Description

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 path name of an exiting 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, ftok() returns a different key. Only the low order 8-bits of *id* are significant and must be nonzero.

Returned Value

If successful, ftok() returns a key.

If unsuccessful, ftok() returns a value of -1 and sets errno to one of the following values under the conditions described:

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 detected in resolving *path*.

ENAMETOOLONG
The length of the *path* argument exceeds PATH_MAX, or a path name component is longer than NAME_MAX, or the path name 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 45
- “msgget() — Get Message Queue” on page 513
- “semget() — Get a Set of Semaphores” on page 690
- “shmget() — Get a Shared Memory Segment” on page 746
- “stat() — Get File Information” on page 803

ftruncate() — Truncate a File

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

Format

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

int ftruncate(int fildes, off_t length);
```

General Description

Truncates the file indicated by the open file descriptor *fildes* to the indicated *length*. *The fildes* argument 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*, the file is extended with sparse data.

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.

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.

If successful, ftruncate() marks the `st_ctime` and `st_mtime` fields of the file.

If unsuccessful, ftruncate() leaves the file unchanged.

Large Files for HFS

Applications that are compiled with the option LANGLVL(EXTENDED) and also define the Feature Test Macro (FTM) _LARGE_FILES before any headers are included will enable this function to operate on HFS files that are larger than 2 gig-1 in size. File size and offset fields will be enlarged to 63 bits in width so any other function operating on this file will have to be enabled with the same FTM.

Returned Value

If successful, ftruncate() returns a value of `0`.

If unsuccessful, ftruncate() returns a value of -1 and sets errno to one of the following values under the conditions described:

EBADF
fildes is not a valid open file descriptor.

EEXTLINK
This function does not support external links.

EINVAL
fildes 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.

Example

CBC3BF49

```
/* CBC3BF49 */
#define _OPEN_SYS
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _OPEN_SYS
#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 51
- "open() — Open a File" on page 527

ftw() — Traverse a File Tree

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

Format

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

int ftw(const char *path,
        int (*fn)(const char *, const struct stat *, int),
        int ndirs);
```

General Description

Recursively descends the directory hierarchy rooted in *path*. For each object in the hierarchy, ftw() calls the function, *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 *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 and uses at most one file descriptor for each level in the tree.

The *ndirs* argument should be in the range of 1 to {OPEN_MAX}. It specifies the maximum number of directory streams, file descriptors, or both, available for use by ftw() while traversing the tree.

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(). When ftw() returns, it closes any directory streams and file descriptors it uses, not counting any opened by the application-supplied *fn*.

Returned Value

If the tree is exhausted, ftw() returns a value of 0.

If the function pointed to by *fn* returns a nonzero value, ftw() stops its tree traversal and returns whatever value was returned by *fn()*.

If ftw() detects an error, it returns -1 and sets errno to one of the following values under the conditions described. All other errnos returned by ftw() are unchanged.

EACCES
Search permission is denied for any component of *path*, or read permission is denied for *path*.

ELOOP
Too many symbolic links were detected.

ENOENT
A component of *path* does not name an existing file or *path* is an empty string.

ENOTDIR
A component of *path* is not a directory.

EINVAL
The value of the *ndirs* argument is invalid.

ENAMETOOLONG
Path name resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

ENAMETOOLONG
The length of *path* exceeds {PATH_MAX}, or a path name component is longer than {PATH_MAX}.

Related Information

- “ftw.h” on page 26
- “longjmp() — Restore Stack Environment” on page 465
- “lstat() — Get Status of File, External Link, or Symbolic Link” on page 471
- “malloc() — Reserve Storage Block” on page 477
- “opendir() — Open a Directory” on page 531
- “siglongjmp() — Restore the Stack Environment and Signal Mask” on page 764
- “stat() — Get File Information” on page 803

fupdate() — Update a VSAM Record

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*. For information on performing VSAM I/O operations, see *z/OS: XL C/C++ Programming Guide* .

You can *only* use the fupdate() function with a VSAM data set opened in update mode (`rb+/r+b`, `ab+/a+b`, or `wb+/w+b`) with the `type=record` option.

You can only use fupdate() after performing an fread() call 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 nonstandard 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 (that is, 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

A call to fupdate() can change the size of the record. 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

A call to fupdate() cannot change the size of a record. 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.

An update cannot change the key of reference (the prime key if opened as a cluster, the alternative index key if opened as a path). 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

If successful, fupdate() returns the size of the updated record.

If unsuccessful, fupdate() returns a value of `0`.

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

- For information on performing VSAM I/O operations, see *z/OS: XL C/C++ Programming Guide*.
- "stdio.h" on page 40
- "fdelrec() — Delete a VSAM Record" on page 216
- "flocate() — Locate a VSAM Record" on page 246

fwrite() — Write Items

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, you should 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.

The fwrite() function 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition must also occur, unless an EOF has been reached.

Returned Value

Returns the number of bytes 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 40
- “fopen() — Open a File” on page 254
- “fread() — Read Items” on page 277
- “freopen() — Redirect an Open File” on page 280

gai_strerror() — Address and Name Information Error Description

Standards

Standards / Extensions;	C or C++	Dependencies
RFC2553	both	

Format

```
#define _OPEN_SYS_SOCK_IVP6
#include <netdb.h>

char *gai_strerror(int ecode);
```

General Description

The gai_strerror() function returns a pointer to a text string describing the error value returned by a failure return from either the getaddrinfo() or getnameinfo() function. If the *ecode* is not one of the EAI_xxx values from the <netdb.h> then gai_strerror() returns a pointer to a string indicating an unknown error.

Subsequent calls to gai_strerror() will overwrite the buffer containing the text string.

Returned Value

When successful, gai_strerror() returns a pointer to a string describing the error. Upon failure, gai_strerror() will return NULL and set errno to one of the following:

Error Code
Description

ENOMEM
Insufficient memory to allocate buffer for text string describing the error.

Related Information

- "getaddrinfo() — Get Address Information" on page 313
- "getnameinfo() — Get Name Information" on page 346
- "netdb.h" on page 34

gamma() — Calculate Gamma Function

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

Format

```
#include <math.h>

double gamma(double x);
```

Compiler Option

LANGLVL(SAA), LANGLVL(SAAL2), or LANGLVL(EXTENDED)

General Description

Computes the natural logarithm of the absolute value of `G(x)` (`ln(|G(x)|)`), where:

$$G(x) = \int_0^\infty e^{-t} \times t^{x-1} dt$$

The *x* argument must be a positive real value.

Returned Value

If *x* is a negative value, gamma() sets errno to EDOM. If the result causes an overflow, gamma() returns HUGE_VAL and sets errno to ERANGE.

Example

CBC3BG01

```
/* CBC3BG01
   This example uses gamma() 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(gamma(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 32
- "erf() - erfc() — Calculate Error and Complementary Error Functions" on page 189
- "j0(), j1(), jn() — Bessel Functions of the First Kind" on page 438
- "y0(), y1(), yn() — Bessel Functions of the Second Kind" on page 983

gcvt() — Convert Double to String

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().

z/VM 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 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 264) in the array pointed to by *buf* and returns *buf*. The gcvt() function produces *ndigit* significant digits (where the total number of digits in the output string is 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 the value is less than `0`. Trailing zeros are suppressed where the value is not a whole number. A radix character is included in the returned string if the value is not a whole number.

The radix character is determined by the current locale. If setlocale() ("setlocale() — Set Locale" on page 721) 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.

Returned Value

If successful, gcvt() returns the character equivalent of *x* as specified above.

If unsuccessful, gcvt() returns NULL.

Related Information

- "stdlib.h" on page 42
- "ecvt() — Convert Double to String" on page 183
- "fcvt() — Convert Double to String" on page 215

getc() - getchar() — Read a Character

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

Format

```
#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)`.

Recommendation: The getc() and fgetc() functions are identical; however, getc() and getchar() are provided in a highly efficient macro form. For performance purposes, you should use the macro forms 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)`

The getc() and getchar() functions are not supported for files opened with `type=record`.

The getc() and getchar() functions 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition must also occur, unless an EOF has been reached.

If the application is not multithreaded, 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 C: 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 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.

Note: You should use ferror() or feof() to determine whether an error or an EOF condition occurred. An 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 40
- “fgetc() — Read a Character” on page 235
- “gets() — Read a String” on page 364
- “putc() - putchar() — Write a Character” on page 613
- “ungetc() — Push Character onto Input Stream” on page 907

getaddrinfo() — Get Address Information

Standards

Standards / Extensions	C or C++	Dependencies
RFC2553	both	

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *nodename,
                const char *servname,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

General Description

The getaddrinfo() function translates the name of a service location (for example, a host name) and/or service name and returns a set of socket addresses and associated information to be used in creating a socket with which to address the specified service.

The *nodename* and *servname* arguments are either pointers to null-terminated strings or null pointers. One or both of these two arguments must be specified as a non-null pointer.

The format of a valid name depends on the protocol family or families. If a specific family is not given and the name could be interpreted as valid within multiple supported families, the function attempts to resolve the name in all supported families. When no errors are detected, all successful results are returned.

If the *nodename* argument is not null, it can be a descriptive name or it can be an address string. If the specified address family is AF_INET, AF_INET6, or AF_UNSPEC, valid descriptive names include host names. If the specified address family is AF_INET or AF_UNSPEC, address strings using standard dot notation as specified in inet_addr() are valid. If the specified address family is AF_INET6 or AF_UNSPEC, standard IPv6 text forms described in inet_pton() are valid.

If *nodename* is not null, the requested service location is named by *nodename*; otherwise, the requested service location is local to the caller.

If *servname* is null, the call returns network-level addresses for the specified *nodename*. If *servname* is not null, it is a null-terminated character string identifying the requested service. This can be either a descriptive name or a numeric representation suitable for use with the address family or families. If the specified address family is AF_INET, AF_INET6, or AF_UNSPEC, the service can be specified as a string specifying a decimal port number.

If the argument *hints* is not null, it refers to a structure containing input values that may direct the operation by providing options and by limiting the returned information to a specific socket type, address family and/or protocol. In the *hints* structure every member other than *ai_flags*, *ai_family*, *ai_socktype*, and *ai_protocol* must be zero or a null pointer. A value of AF_UNSPEC for *ai_family* means that the caller will accept any protocol family. A value of zero for *ai_socktype* means that the caller will accept any socket type. A value of zero for ai_protocol means that the caller will accept any protocol. If *hints* is a null pointer, the behavior will be as if it referred to a structure containing the value zero for the *ai_flags*, *ai_socktype*, and *ai_protocol* fields, and AF_UNSPEC for the *ai_family* field.

The *ai_flags* member to which the *hints* argument points can be set to 0 or be the bitwise inclusive OR of one or more of the following values:

- AI_PASSIVE
- AI_CANONNAME
- AI_NUMERICHOST
- AI_NUMERICSERV

- AI_V4MAPPED
- AI_ALL
- AI_ADDRCONFIG

If the AI_PASSIVE bit is set in the *ai_flags* member of the *hints* structure, then the caller plans to use the returned socket address structure in a call to bind(). In this case, if the *nodename* argument is a null pointer, then the IP address portion of the socket address structure will be set to INADDR_ANY for an IPv4 address or IN6ADDR_ANY_INIT for an IPv6 address. If the AI_PASSIVE bit is not set in the *ai_flags* member of the *hints* structure, then the returned socket address structure will be ready for a call to connect() (for a connection-oriented protocol) or either connect(), sendto(), or sendmsg() (for a connectionless protocol). In this case, if the *nodename* argument is a null pointer, then the IP address portion of the socket address structure will be set to the loopback address.

If the AI_CANONNAME bit is set in the *ai_flags* member of the *hints* structure, then upon successful return the *ai_canonname* member of the first addrinfo structure in the linked list will point to a null-terminated string containing the canonical name of the specified nodename.

If the AI_NUMERICHOST bit is set in the *ai_flags* member of the *hints* structure, then a non-null *nodename* string must be a numeric host address string. Otherwise an error code of EAI_NONAME is returned. This flag prevents any type of name resolution service (for example, the DNS) from being called.

If the AI_NUMERICSERV flag is specified, then a non-null *servname* string must be a numeric port string. Otherwise an error code EAI_NONAME is returned. This flag prevents any type of name resolution service (for example, NIS+ from being invoked.

If the AI_V4MAPPED flag is specified along with the *ai_family* with the value of AF_INET6, or a value of AF_UNSPEC when IPv6 is supported on the system, then the caller will accept IPv4-mapped IPv6 addresses. When the AI_ALL flag is not also specified and no IPv6 addresses are found, then a query is made for IPv4 addresses. If any IPv4 addresses are found, they are returned as IPv4-mapped IPv6 addresses.

If the *ai_family* does not have a value of AF_INET6 or contains AF_UNSPEC, and IPv6 is not supported on the system, this flag is ignored.

When the *ai_family* has a value of AF_INET6 and AI_ALL is set, the AI_V4MAPPED flag must also be set to indicate that the caller will accept all addresses (IPv6 and IPv4-mapped IPv6 addresses). When the *ai_family* has a value of AF_UNSPEC when the system supports IPv6 and AI_ALL is set, the caller accepts IPv6 addresses and either IPv4 (if AI_V4MAPPED is not set) or IPv4-mapped IPv6 (if AI_V4MAPPED is set) addresses. A query is first made for IPv6 addresses and if successful, the IPv6 addresses are returned. Another query is then made for IPv4 addresses and any found are returned as IPv4 addresses (if AI_V4MAPPED was not set) or as IPv4-mapped IPv6 addresses (if AI_V4MAPPED was set). If the *ai_family* does not have the value of AF_INET6, or the value of AF_UNSPEC when the system supports IPv6, the flag is ignored.

If the AI_ADDRCONFIG flag is specified, then a query for IPv6 address records should occur only if the node has at least one IPv6 source address configured. A query for IPv4 address records will always occur, whether or not any IPv4 addresses are configured. The loopback address is not considered a valid configured source address in this case.

All of the information returned by getaddrinfo() is dynamically allocated: the addrinfo structures, and the socket address structures and canonical node name strings pointed to by the addrinfo structures. To return this storage to the system, the function freeaddrinfo() is called.

Application Usage Notes

1. If the caller handles only TCP and not UDP, then the *ai_protocol* member of the *hints* structure should be set to IPPROTO_TCP when getaddrinfo() is called.
2. If the caller handles only IPV4 and not IPv6, then the *ai_family* member of the *hints* structure should be set to AF_INET when getadddrinfo() is called.

Returned Value

When successful, getaddrinfo() returns 0 and a pointer to a linked list of one or more addrinfo structures through the *res* argument. The caller can process each addrinfo structure in this list by following the *ai_next* pointer until a null pointer is encountered. In each returned addrinfo structure, the three members *ai_family*, *ai_socktype*, and *ai_protocol* are the corresponding arguments for a call to the socket() function. In each addrinfo structure the *ai_addr* member points to a filled-in socket address structure whose length is specified by the *ai_addrlen* member. Upon failure, getaddrinfo() returns a non-zero error code. The error codes are as follows:

Error Code
Description

EAI_AGAIN
The name specified by the *nodename* or *servname* parameter could not be resolved within the configured time interval, or the resolver address space has not been started. The request can be retried later.

EAI_BADFLAGS
The flags parameter had an incorrect setting.

EAI_FAIL
An unrecoverable error occurred.

EAI_FAMILY
The family parameter had an incorrect setting.

EAI_MEMORY
A memory allocation failure occurred during an attempt to acquire an *addrinfo* structure.

EAI_NONAME
One of the following conditions occurred:

1. The name does not resolve for the specified parameters. At least one of the *nodename* or *servname* operands must be specified.
2. The request name parameter is valid, but it does not have a record at the name server.

EAI_SERVICE
The service that was passed was not recognized for the specified socket type.

EAI_SOCKTYPE
The intended socket type was not recognized.

EAI_SYSTEM
A system error occurred.

Related Information

- “netdb.h” on page 34
- “sys/socket.h” on page 46
- “freeaddrinfo() — Free Addrinfo Storage” on page 279
- “gai_strerror() — Address and Name Information Error Description” on page 309
- “getnameinfo() — Get Name Information” on page 346

getclientid() — Get Identifier for Calling Application

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	

Format

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

int getclientid(int domain, struct clientid *clientid);
```

General Description

Returns the identifier by which the calling program is known to the TCP/IP virtual machine. The *clientid* is used in the givesocket() and takesocket() calls.

The getclientid(), givesocket(), and takesocket() calls can be used by cooperating applications to pass ownership of an open AF_INET or AF_INET6 stream socket between them.

Note: You can give an AF_INET socket only to an AF_INET socket, and an AF_INET6 socket only to an AF_INET6 socket; that is you can NOT give an AF_INET6 socket to an AF_INET socket or vise versa.

Assume that application A wishes to give a socket (typically, a socket obtained by means of accept(), but it can be any connected AF_INET stream socket) to application B. Application B passes its client ID to application A through an agreed- upon mechanism, and A uses B's client ID in the givesocket() call to identify to whom the socket is given. After A has successfully issued givesocket(), it passes its client ID and the proper "socket number" to B, and B uses them in the takesocket() call to identify from whom the socket is being taken. For more information about how to pass ownership of sockets from one application to another, see “givesocket() — Make Specified Socket Available” on page 387.

domain
The address domain requested. It must be AF_INET or AF_INET6.

clientid
The pointer to a `clientid` structure to be filled.

The `clientid` structure is filled in by the call and returned as follows:

The `clientid` structure:

```
struct clientid {
   int domain;
       name[8];
   char subtaskname[8];
       reserved[20];
};
```

domain
Specifies AF_INET or AF_INET6.

name
Specifies the VM user ID of the caller's virtual machine, left-justified and padded with blanks.

subtaskname
Specifies the "subtaskname" used by the caller when it established its IUCV connection with the TCP/IP server virtual machine.

reserved
Contains blanks.

Returned Value

If successful, getclientid() returns a value of `0`.

If unsuccessful, getclientid() returns a value of `-1` and sets errno to the following value under the conditions described:

EAFNOSUPPORT
The address family is not supported.

EFAULT
Using the *clientid* parameter, as specified, would result in an attempt to access storage outside the caller's virtual machine or storage not modifiable by the caller.

EPFNOSUPPORT
The domain is not AF_INET or AF_INET6.

Related Information

- "sys/socket.h" on page 46
- "sys/types.h" on page 46
- "givesocket() — Make Specified Socket Available" on page 387
- "takesocket() — Acquire a Socket from Another Program" on page 860

getcontext() — Get User Context

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

Initializes the `ucontext_t` type structure pointed to by *ucp* to the current user context of the calling process. The `ucontext_t` type includes the contents of the calling process' machine registers, the signal mask, and the current execution stack. Calling 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() had just returned.

The return value from getcontext() is the same regardless of whether it is from the initial invocation or from calling setcontext().

The makecontext() function may modify the context created by getcontext(). The getcontext() function is similar in some respects to sigsetjmp(), setjmp(), and _setjmp(). You cannot intermix the getcontext()–setcontext() pair, the sigsetjmp()–siglongjmp() pair, the setjmp()–longjmp() pair, and the _setjmp()–_longjmp() pair. You should only use setcontext() to restore a context saved by getcontext().

Recommendation: You should not use siglongjmp() from a signal handler to restore a context created with getcontext(), even though some compatibility exists with siglongjmp().

Portable applications should not modify or access the `uc_mcontext` member of `ucontext_t` because the context may not include process-wide static data.

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         A pointer to the context that will
                                 resume 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 z/VM C++ program, the behavior in terms of the destruction of automatic objects is undefined. This applies to both z/VM C++ and C/VM 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.

Special Behavior for XPLINK-compiled C++

Restrictions concerning setjmp.h and ucontext.h:

Note:

1. All XPLINK programs compiled with the V2.10 or later C compilers that are to run with Language Environment V2.10 or later libraries and use the **jmp_buf**, **sigjmp_buf** or **ucontext_t** types must not be compiled with C headers from Language Environment V2.9 or earlier.
2. Non-XPLINK functions compiled with any level of Language Environment headers must not define **jmp_buf**, **sigjmp_buf** or **ucontext_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), _longjmp(), setjmp(), _setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.
3. When __XPLINK__ is defined, the Language Environment V2.10 and later headers define a larger **jmp_buf**, **sigjmp_buf** or **ucontext_t**area that is required by setjmp(), getcontext(), and related functions when they are called from an XPLINK routine. If __XPLINK__ is not defined, the Language Environment V2.10 and later headers define a shorter **jmp_buf**, **sigjmp_buf** or **ucontext_t** area. The Language Environment headers prior to V2.10 also define the shorter version of these data areas. If an XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp_buf**, **sigjmp_buf** or **ucontext_t** area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.

Returned Value

If successful, getcontext() returns a value of `0`.

If unsuccessful, getcontext() returns a value of -1.

No errnos are defined for getcontext().

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 a value of `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 51
- "makecontext() — Modify User Context" on page 475
- "setcontext() — Restore User Context" on page 704
- "_setjmp() — Set Jump Point for a Nonlocal Goto" on page 717
- "setjmp() — Preserve Stack Environment" on page 718
- "sigaction() — Examine or Change a Signal Action" on page 749
- "sigsetjmp() — Save Stack Environment and Signal Mask" on page 778
- "swapcontext() — Save and Restore User Context" on page 849

getcwd() — Get Path Name of the Working Directory

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

Format

```
#define _OPEN_SYS
#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, as well as a terminating null to mark the end of the name.

Note: The path name of the working directory will contain the fully qualified root if it resides in a different file system from the mounted root.

Returned Value

If successful, getcwd() returns a pointer to the buffer.

If unsuccessful, getcwd() returns a NULL pointer and sets errno to one of the following values under the conditions described:

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.

ENAMETOOLONG
The resulting CWD length is greater than PATH_MAX.

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 `0` 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 _OPEN_SYS
#include <unistd.h>
#undef _OPEN_SYS
#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 51
- “chdir() — Change the Working Directory” on page 116

getdate() — Convert User Format Date and Time

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

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <time.h>
struct tm *getdate(const char *string);

extern int getdate_err;
```

General Description

Converts definable date and/or time specifications, pointed to by *string*, into a *tm* structure that is declared in the <time.h> header file.

The getdate() function uses templates to parse and interpret the input string. They are contained in text files created by the process and identified by the environment variable DATEMASK, which indicates the path name of the file containing the templates. The first line in the template that matches the input string 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, that is, `tm_sec` in the `tm` structure returned by getdate() is set to 59 if the input string specifies 59, 60 or 61 for %S.

%t
same as \t (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 %m/%d/%y.

%X
locale's time representation. In the POSIX locale this is equivalent to %H:%M:%S.

%y
year within century (69-99)

%Y
year as ccyy (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 getdate() expects, an invalid input specification error will result. The getdate() function calculates an expected time zone based on time and date information supplied to it.

Notes:

- The match between the template and the input string is case insensitive.
- The month and weekday names can consist of any combination of upper or lower case letters.
- The input date and time can be set to 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, a maximum of 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.

Rules: The following rules apply for converting the input string into a *tm* structure:

- If only the 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.This structure is not shared with any other functions and is unique to the thread on which it runs.

If unsuccessful, getdate() returns a null pointer and sets the external variable *getdate_err* to a value indicating the error.

As is true for all external variables, IBM XL C/C++ for z/VM 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."

IBM XL C/C++ for z/VM allocates storage on a per thread basis for an analog of `getdate_err`. The __gderr() function returns a pointer to this storage.

Recommendation: If getdate() returns a null pointer, you should use the __gderr() function for multithread applications and applications running from a DLL to determine in a thread-safe manner why getdate() was unsuccessful.

The __gderr() function 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 descriptions:

1
The DATEMSK environment variable is null or undefined.

2
The template file cannot be opened for reading.

3
An attempt to get file status information failed.

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
The input string is invalid, for example, February 31; or a time that cannot be represented in a time_t (representing the time is seconds since Epoch - midnight, January 1, 1970).

9
The current time cannot be determined.

Note: This value is unique for OpenExtensions services.

Related Information

- "time.h" on page 49

getdtablesize() — Get File Descriptor Table Size

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

Format

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

int getdtablesize(void);
```

General Description

Returns 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 (for example, the first file descriptor assigned is 0).

Returned Value

Returns the file descriptor limit.

No errno values are defined for getdtablesize().

Related Information

- "unistd.h" on page 51
- "close() — Close a File" on page 127
- "open() — Open a File" on page 527
- "select() — Monitor Activity on Files/Sockets and Message Queues" on page 680

getegid() — Get the Effective Group ID

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

Format

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

gid_t getegid(void);
```

General Description

Finds the effective group ID (GID) of the calling process.

The getegid() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

Returns the effective group ID (GID). The getegid() function is always successful.

No errnos are defined for getegid().

Example

CBC3BG04

```
/* CBC3BG04
   This example finds the group ID.
 */
#define _OPEN_SYS
#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 46
- "getgid() — Get the Real Group ID" on page 329
- "setegid() — Set the Effective Group ID" on page 706
- "setgid() — Set the Group ID" on page 711

getenv() — Get Value of Environment Variables

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 the table entry.

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. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Special Behavior for C

Under CMS, the behavior of getenv() is affected by the _EDC_COMPAT environment variable. When the getenv bit of _EDC_COMPAT is set and the environment variable is not found in the environment table, and the variable is not an IBM XL C/C++ for z/VM specific environment variable, the LASTING GLOBALV file is checked.

You can set global variables for use in your C environment by issuing the following CMS command:

```
    GLOBALV SELECT CENV SETLP var_name var_value group_id
```

The following getenv() calls show where the environment variable will be retrieved under CMS when the getenv bit of the _EDC_COMPAT environment variable is set:

- `getenv("var1")` gets the value of *var1* from the environment variable table. If *var1* is not in the environment variable table, the value of *var1* is obtained from under group ID CENV (the default group ID) using the GLOBALV facility.
- `getenv(".var1")` gets the value of `.var1` from the environment variable table. If `.var1` is not in the environment variable table, the value of *var1* is obtained from under the group ID UNNAMED using the GLOBALV facility. Thus, `getenv("mygroup.var1")` gets the value of `mygroup.var1` from the environment variable table. If `mygroup.var1` is not in the environment variable table, the value of *var1* is obtained from under the group ID mygroup using the GLOBALV facility.

Returned Value

Returns a pointer to the environment table entry containing the current string value of *varname*. You should copy the returned string because a subsequent call to getenv() could overwrite it.

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

- For information about using environment variables, see *z/OS: XL C/C++ Programming Guide*.
- “stdlib.h” on page 42
- “clearenv() — Clear Environment Variables” on page 122
- “setenv() — Add, Delete, and Change Environment Variables” on page 708

geteuid() — Get the Effective User ID

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

Format

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

```
uid_t geteuid(void);
```

General Description

Finds the effective user ID (UID) of the calling process.

The geteuid() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

Returns the effective user ID of the calling process. The geteuid() function is always successful.

No errnos are defined for geteuid().

Example

CBC3BG06

```
/* CBC3BG06
   This example returns information for your user ID.
 */
#define _OPEN_SYS
#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 user ID:

```
pw_name  : vmusr1
pw_uid   : 25
pw_gid   : 500
pw_dir   : /u/vmusr1
pw_shell : /bin/sh
```

Related Information

- “sys/types.h” on page 46
- “getuid() — Get the Real User ID” on page 378
- “seteuid() — Set the Effective User ID” on page 710
- “setuid() — Set the Effective User ID” on page 739

getgid() — Get the Real Group ID

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

Format

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

gid_t getgid(void);
```

General Description

Finds the real group ID (GID) of the calling process.

The getgid() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

Returns the real group ID of the calling process. The getgid() function is always successful.

No errnos are defined for getgid().

Example

CBC3BG07

```
/* CBC3BG07
   This example gets the real group ID.
 */
#define _OPEN_SYS
#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 46
- "getegid() — Get the Effective Group ID" on page 325
- "geteuid() — Get the Effective User ID" on page 327
- "getuid() — Get the Real User ID" on page 378
- "setgid() — Set the Group ID" on page 711

getgrgid() — Access the Group Database by ID

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

Format

```
#define _OPEN_SYS
#include <grp.h>

struct group *getgrgid(gid_t gid);
```

General Description

Provides information about the group, *gid* and its members.

The getgrgid() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

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, getgrgid() returns a NULL pointer and sets errno to one of the following values under the conditions described:

ECMSERR
CMS could not retrieve the information because it ran out of storage.

ECPERR
CP suffered an internal error while trying to retrieve the information.

Example

CBC3BG08

```
/* CBC3BG08
   This example provides the root GID and group name.
 */
#define _OPEN_SYS
#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 26
- "sys/types.h" on page 46
- "getgrnam() — Access the Group Database by Name" on page 331

getgrnam() — Access the Group Database by Name

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

Format

```
#define _OPEN_SYS
#include <grp.h>

struct group *getgrnam(const char *name);
```

General Description

Accesses the `group` structure containing an entry from the group database with the specified *name*.

The getgrnam() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

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 and sets errno to one of the following values under the conditions described:

ECMSERR
CMS could not retrieve the information, because it ran out of storage.

ECPERR
CP suffered an internal error while trying to retrieve the information.

Example

CBC3BG09

```
/* CBC3BG09
   This example provides the members of a group.
 */
#define _OPEN_SYS
#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:
  vmusr1
  vmusr2
  vmusr3
  vmusr4
  vmusr5
  vmusr6
  vmusr7
  vmusr8
  vmusr9
```

Related Information

- “grp.h” on page 26
- “sys/types.h” on page 46
- “getgrgid() — Access the Group Database by ID” on page 330

getgroups() — Get a List of Supplementary Group IDs

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

Format

```
#define _OPEN_SYS
#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.

The getgroups() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, getgroups() returns the number of 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.

If *size* is `0`, getgroups() returns the total number of supplementary group IDs for the process. The getgroups() function does not try to store group IDs in *list*.

If unsuccessful, getgroups() returns a value of -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.

In addition, getgroups() can return ECMSERR or ECPERR to indicate that the system detected an internal error or had insufficient storage.

Example

CBC3BG10

```
/* CBC3BG10
   This example provides a list of the supplementary group IDs.
 */
#define _OPEN_SYS
#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 46
- "getegid() — Get the Effective Group ID" on page 325
- "getgid() — Get the Real Group ID" on page 329
- "getgrnam() — Access the Group Database by Name" on page 331
- "setgid() — Set the Group ID" on page 711

getgroupsbyname() — Get Supplementary Group IDs by User Name

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	

Format

```
#define _OPEN_SYS
#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*. The *size* argument gives the number of `gid_t` elements that can be stored in the array, *list*.

The getgroupsbyname() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, getgroupsbyname() returns the number of the 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.

If *size* is `0`, getgroupsbyname() returns the total number of supplementary group IDs for the user. The getgroupsbyname() function does not try to store group IDs in *list*.

If unsuccessful, getgroupsbyname() returns a value of -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 user; *list* may or may not contain a subset of the supplementary group IDs for the user.

Example

CBC3BG11

```
/* CBC3BG11
   This example provides a list of the supplementary group IDs for vmusr1.
 */
#define _OPEN_SYS
#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[ ]="vmusr1";

  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");
      else
        printf("  %8s (%d)\n", grp->gr_name, (int) gids[curr]);
    }
  }
}
```

Output

```
The following is the list of vmusr1's supplementary groups:
      SYS1 (500)
     USERS (523)
```

Related Information

- “unistd.h” on page 51
- “getegid() — Get the Effective Group ID” on page 325
- “getgid() — Get the Real Group ID” on page 329
- “getgroups() — Get a List of Supplementary Group IDs” on page 332
- “setgid() — Set the Group ID” on page 711

gethostbyaddr() — Get a Host Entry by Address

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

Tries to resolve the host address through a name server if one is present. The first call to either gethostbyaddr() or gethostbyname() will set up a cache to store host names and IP addresses. You may set the size of the cache with the _EDC_IP_CACHE_ENTRIES environmental variable. You may set the size of the cache only once. If you do not specify _EDC_IP_CACHE_ENTRIES, the function will use a default value of 20.

This function is not supported for AF_INET6 sockets.

If a name server is not present, gethostbyaddr() searches the HOSTS ADDRINFO file until it finds a matching host address or reaches an EOF marker.

address
The pointer to a structure containing the address of the host (an unsigned long for AF_INET.

address_len
The size of *address* in bytes.

domain
The address domain supported (AF_INET.

If you want gethostbyaddr() to bypass the name server and instead resolve the host address using the HOSTS ADDRINFO file, you must define the RESOLVE_VIA_LOOKUP symbol before including any sockets-related header files in your source program.

You can use the X_ADDR environment variable to specify a file other than HOSTS ADDRINFO.

Note: HOSTS LOCAL, HOSTS ADDRINFO, and HOSTS SITEINFO are described in *z/VM: TCP/IP Planning and Customization*.

The gethostbyaddr() call returns a pointer to a `hostent` structure for the host address specified on the call.

The gethostent(), gethostbyaddr(), and gethostbyname() functions 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 `hostent` structure is defined in the netdb.h header file and contains the following elements:

h_addr_list
A pointer to a zero-terminated list of host network addresses.

h_addrtype
The type of address returned; it is always set to AF_INET.

h_aliases
A zero-terminated array of alternative names for the host.

h_length
The length of the address in bytes.

h_name
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.

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.

If unsuccessful in X/Open, gethostbyaddr() sets *h_errno* to indicate the error as follows:

HOST_NOT_FOUND
No such host is known.

TRY_AGAIN
A temporary error, such as no response from a server, indicating the information is not available now but may be at a later time.

NO_RECOVERY
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 183
- "gethostbyname() — Get a Host Entry by Name" on page 337
- "gethostent() — Get Next Host Entry" on page 339
- "sethostent() — Open Host Information Data Set" on page 712

gethostbyname() — Get a Host Entry by Name

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

Tries to resolve the host address through a name server, if one is present. The first call to either gethostbyaddr() or gethostbyname() will set up a cache to store host names and IP addresses. You may set the size of the cache with the _EDC_IP_CACHE_ENTRIES environmental variable. You may set the size of the cache only once. If you do not specify _EDC_IP_CACHE_ENTRIES, the function will use a default value of 20.

This function is not supported for AF_INET6 sockets.

The gethostbyname() function will search for the host name in the cache first. If it does not exist, it will construct the long name from the name and domain and send it to the Name Server(s). If a name server is not present, gethostbyname() searches the HOSTS SITEINFO file until it finds a matching host name or reaches an EOF marker.

name
The name of the host.

If you want gethostbyname() to bypass the name server and instead resolve the host name using the HOSTS SITEINFO file, you must define the RESOLVE_VIA_LOOKUP symbol before including any sockets-related header 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 file other than HOSTS SITEINFO.

Note: HOSTS LOCAL, HOSTS ADDRINFO, and HOSTS SITEINFO are described in *z/VM: TCP/IP Planning and Customization*.

The gethostbyname() call returns a pointer to a `hostent` structure for the host name specified on the call.

The gethostent(), gethostbyaddr(), and gethostbyname() functions 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 `hostent` structure is defined in the netdb.h header file and contains the following elements:

h_addr_list
A pointer to a zero-terminated list of host network addresses.

h_addrtype
The type of address returned; it is always set to AF_INET.

h_aliases
A zero-terminated array of alternative names for the host.

h_length
The length of the address in bytes.

h_name
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.

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.

If unsuccessful in X/Open, gethostbyname() sets *h_errno* to indicate the error as follows:

HOST_NOT_FOUND
No such host is known.

TRY_AGAIN
A temporary error, such as no response from a server, indicating the information is not available now but may be at a later time.

NO_RECOVERY
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 183
- “gethostbyaddr() — Get a Host Entry by Address” on page 335
- “gethostent() — Get Next Host Entry” on page 339
- “gethostname() — Get Name of Host Processor” on page 340

• "sethostent() — Open Host Information Data Set" on page 712

gethostent() — Get Next Host Entry

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

Reads the next line of the HOSTS SITEINFO file and returns a pointer to the next entry in the HOSTS SITEINFO file. The gethostent() function uses HOSTS ADDRINFO to get aliases.

You can use the X_SITE environment variable to specify a file other than HOSTS SITEINFO.

Note: HOSTS LOCAL, HOSTS ADDRINFO, and HOSTS SITEINFO are described in *z/VM: TCP/IP Planning and Customization*.

The gethostent(), gethostbyaddr(), and gethostbyname() functions 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 `hostent` structure is defined in the netdb.h header file and contains the following elements:

h_addr
A pointer to the network address of the host.

h_addrtype
The type of address returned; it is always set to AF_INET or AF_INET6.

h_aliases
A zero-terminated array of alternative names for the host.

h_length
The length of the address in bytes.

h_name
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 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.

Related Information

- "gethostbyaddr() — Get a Host Entry by Address" on page 335
- "gethostbyname() — Get a Host Entry by Name" on page 337
- "sethostent() — Open Host Information Data Set" on page 712

gethostid() — Get Unique Identifier of Current Host

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

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

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, gethostname() returns a value of -1 and sets errno to one of the following values under the conditions described:

EAFNOSUPPORT
The address family is not supported.

Related Information

- "gethostname() — Get Name of Host Processor" on page 340

gethostname() — Get Name of Host Processor

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

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

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 insufficient space is available in the name array.

name
The character array to be filled with the host name.

namelen
The length of *name*.

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 gethostname() returns a value of `0`.

If unsuccessful, gethostname() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EAFNOSUPPORT
The address family is not supported.

EFAULT
Using *name* and *namelen* would result in an attempt to copy the address into a portion of the caller's virtual machine to which data cannot be written.

ECMSPARM
Incorrect parameters were passed to the service.

Related Information

- “gethostbyname() — Get a Host Entry by Name” on page 337
- “gethostid() — Get Unique Identifier of Current Host” on page 340

getibmsockopt() — Get Options Associated with a Bulk Mode Socket

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	

Format

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

```
int getibmsockopt(int s,int level,int optname,
      char *optval,int *optlen);
```

General Description

Like getsockopt(), the getibmsockopt() function gets the options associated with a socket in the AF_INET or AF_INET6 domains.

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 getibmsockopt() 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 virtual machine 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 and AF_INET6 domains.

socket
The socket descriptor.

level
The level for which the option is set.

optname
The name of a specified socket option.

optval
The pointer to option data.

optlen
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:

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. If either address or buffer cannot be addressed, errno is set to EFAULT. 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. The getibmsockopt() function returns a value of `0` in *optval* if the socket is in blocking mode, and returns a value of 1 in *optval* if the socket is in nonblocking mode.

For SO_IGNOREINCOMINGPUSH, *optval* should point to an integer. The getibmsockopt() function returns a value of `0` in *optval* if the option is not set, and returns a value of 1 in *optval* if the option is set.

Returned Value

If successful, getibmsockopt() returns a value of `0`.

If unsuccessful, getibmsockopt() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EBADF
The *socket* 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 virtual machine.

ENOPROTOOPT
The *optname* parameter is unrecognized, or the *level* parameter is not SOL_SOCKET.

Example

The following is an example of the getibmsockopt() 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 = getibmsockopt(s, SOL_SOCKET, SO_BULKMODE, (char *) &bulkstr, &optlen);
   if (rc < 0)
     { tcperror("on getibmsockopt()");
       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 46
- “ibmsflush() — Flush Application-side Datagram Queue” on page 398
- “setibmsockopt() — Set Options Associated with a Bulk Mode Socket” on page 713

getlogin() — Get the User Login Name

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

Format

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

char *getlogin(void);
```

General Description

Finds the user name associated with the current terminal. If called from a virtual machine with an alternate user ID in effect, getlogin() finds the name associated with the alternate user ID. This string is stored in a static data area and, therefore, may be overwritten with every call to getlogin().

The getlogin() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

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.

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.

Returned Value

No errnos are defined for getlogin().

- If successful, getlogin() returns a pointer to a string that has the login name for the current terminal.
- If unsuccessful, getlogin() returns a NULL pointer.

  Special Behavior for XPG4.2

  If unsuccessful, getlogin() returns a NULL pointer and sets errno to one of the following values:

  Error Code
  Description

  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 _OPEN_SYS
#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 51
- “getpwuid() — Access the User Database by User ID” on page 362
- “getuid() — Get the Real User ID” on page 378

getmccoll() — Get Next Collating Element from String

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <collate.h>

collel_t getmccoll(char **src)
```

General Description

If the object pointed to by *src* is not a null pointer, getmccoll() determines the longest sequence of bytes in the array pointed to by *src* that constitute a valid multicharacter collating element. It then produces the value of type `collel_t` corresponding to that collating element. The object pointed to by *src* is assigned the address just past the last byte of the multicharacter collating element processed.

Returned Value

If the object pointed to by *src* is a null pointer or if it points to a null character, getmccoll() returns a value of `0`. Otherwise, it returns the value of type `collel_t` that represents the collating element found.

Related Information

- “collate.h” on page 20
- “cclass() — Return Characters in a Character Class” on page 104
- “collequiv() — Return List of Equivalent Collating Elements” on page 132
- “collorder() — Return List of Collating Elements” on page 133
- “collrange() — Calculate the Range List of Collating Elements” on page 134
- “colltostr() — Return a String for a Collating Element” on page 136
- “getwmccoll() — Get Next Collating Element from Wide String” on page 386
- “ismccollel() — Identify a Multicharacter Collating Element” on page 429
- “maxcoll() — Return Maximum Collating Element” on page 479

- “strtocoll() — Return Collating Element for String” on page 839

getnameinfo() — Get Name Information

Standards

Standards / Extensions	C or C++	Dependencies
RFC2553	both	

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(cons struct sockaddr *sa, socklen_t salen,
                char *host, socklen_t hostlen,
                char *serv, socklen_t servlen,
                int flags);
```

General Description

The getnameinfo() function translates a socket address to a node name and service location. The getnameinfo() function looks up an IP address and port number provided by the caller in the DNS and system-specific database, and returns text strings for both in buffers provided by the caller.

The *sa* argument points to either a sockaddr_in structure (for IPv4) or a sockaddr_in6 structure (for IPv6) that holds the IP address and port number. The *salen* argument gives the length of the sockaddr_in or sockaddr_in6 structure.

If the socket address structure contains an IPv4-mapped IPv6 address or an IPv4-compatible IPv6 address, the embedded IPv4 address is extracted and the lookup is performed on the IPv4 address.

Note: The IPv6 unspecified address (“::”) and the IPv6 loopback address (“::1”) are not IPv4-compatible addresses. If the address is the IPv6 unspecified address, a lookup is not performed, and the EAI_NONAME error code is returned.

The node name associated with the IP address is returned in the buffer pointed to by the *host* argument. The caller provides the size of this buffer in the *hostlen* argument. Up to *hostlen* characters will be copied into the buffer as a null-terminated string. The caller elects not to return the node name by specifying a zero value for *hostlen* or a null *host* argument. If the node's name cannot be located, the numeric form of the node's address is returned instead of its name.

The service name associated with the port number is returned in the buffer pointed to by the *serv* argument, and the *servlen* argument gives the length of this buffer. Up to *servlen* characters will be copied into the buffer as a null-terminated string. The caller elects not to return the service name by specifying a zero value for *servlen* or a null *serv* argument. If the service's name cannot be located, the numeric form of the service address (for example, its port number) will be returned instead of its name.

The final argument, *flags*, is a flag that changes the default actions of this function. By default the fully-qualified domain name (FQDN) for the host is returned.

If the flag bit NI_NOFQDN is set, only the node name portion of the FQDN is returned for local hosts.

If the flag bit NI_NUMERICHOST is set, the numeric form of the host's address is returned instead of its name.

If the flag bit NI_NAMEREQD is set, an error is returned if the host's name cannot be located.

If the flag bit NI_NUMERICSERV is set, the numeric form of the service address is returned (for example, its port number) instead of its name.

If the flag bit NI_DGRAM is set, this specifies that the service is a datagram (UDP) service. This flag is required for the few ports (for example, [512,514]) that have different services for UDP and TCP.

Note: The two NI_NUMERICxxx flags are required to support the "-n" flag that many commands provide.

Returned Value

Error Code
Description

EAI_AGAIN
The specified host address could not be resolved within the configured time interval, or the resolver address space has not been started. The request can be retried later.

EAI_BADFLAGS
The flags parameter had an incorrect value.

EAI_FAIL
An unrecoverable error occurred.

EAI_FAMILY
The address family was not recognized, or the address length was not valid for the specified family.

EAI_MEMORY
A memory allocation failure occurred.

EAI_NONAME
The name does not resolve for the supplied parameters. One of the following occurred:

1. NI_NAMEREQD is set, and the host name cannot be located.
2. Both host name and service name were null.
3. The requested address is valid, but it does not have a record at the name server.

EAI_SYSTEM
An unrecoverable error occurred.

Related Information

- “gai_strerror() — Address and Name Information Error Description” on page 309
- “getaddrinfo() — Get Address Information” on page 313
- “getservbyname() — Get a Server Entry by Name” on page 365
- “getservbyport() — Get a Service Entry by Port” on page 366
- “inet_ntop() — Convert Internet Address Format from Binary to Text ” on page 413
- “socket() — Create a Socket” on page 791
- “netdb.h” on page 34
- “sys/socket.h” on page 46

getnetbyaddr() — Get a Network Entry by Address

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

Searches the HOSTS ADDRINFO file for the specified network address.

net
The network address.

type
The address domain supported (AF_INET or AF_INET6).

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 file other than HOSTS ADDRINFO.

Note: HOSTS LOCAL, HOSTS ADDRINFO, and HOSTS SITEINFO are described in *z/VM: TCP/IP Planning and Customization*.

The getnetbyaddr(), getnetbyname(), and getnetent() functions 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 header file and contains the following elements:

n_addrtype
The type of network address returned; it is always set to AF_INET or AF_INET6.

n_aliases
A zero-terminated array of alternative names for the network.

n_name
The official name of the network.

n_net
The network number, returned in host byte order.

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 183
- “endnetent() — Close Network Information Data Set” on page 184
- “getnetbyname() — Get a Network Entry by Name” on page 348
- “getnetent() — Get Next Network Entry” on page 350
- “setnetent() — Open Network Information Data Set” on page 728

getnetbyname() — Get a Network Entry by Name

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

Searches the HOSTS SITEINFO file for the specified network name and returns a pointer to a `netent` structure for the network name.

name
The pointer to a network name.

You can use the X_SITE environment variable to specify a file other than HOSTS SITEINFO.

Note: HOSTS LOCAL, HOSTS ADDRINFO, and HOSTS SITEINFO are described in *z/VM: TCP/IP Planning and Customization*.

The getnetbyaddr(), getnetbyname(), and getnetent() functions 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` header file and contains the following elements:

n_addrtype
The type of network address returned; it is always set to AF_INET or AF_INET6.

n_aliases
A zero-terminated array of alternative names for the network.

n_name
The official name of the network.

n_net
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 183
- “endnetent() — Close Network Information Data Set” on page 184
- “getnetbyaddr() — Get a Network Entry by Address” on page 347
- “getnetent() — Get Next Network Entry” on page 350
- “setnetent() — Open Network Information Data Set” on page 728

getnetent() — Get Next Network Entry

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

Reads the next entry of the HOSTS ADDRINFO file and returns a pointer to the next entry in the HOSTS SITEINFO file.

You can use the X_ADDR environment variable to specify a file other than HOSTS ADDRINFO.

Note: HOSTS LOCAL, HOSTS ADDRINFO, and HOSTS SITEINFO are described in *z/VM: TCP/IP Planning and Customization*.

The getnetbyaddr(), getnetbyname(), and getnetent() functions 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 header file and contains the following elements:

n_addrtype
The type of network address returned; it is always set to AF_INET.

n_aliases
A zero-terminated array of alternative names for the network.

n_name
The official name of the network.

n_net
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 183
- “endnetent() — Close Network Information Data Set” on page 184
- “gethostbyaddr() — Get a Host Entry by Address” on page 335

- “gethostbyname() — Get a Host Entry by Name” on page 337
- “setnetent() — Open Network Information Data Set” on page 728

getopt() — Parse Command Option

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 *optsting);

extern char *optarg;
extern int optind, opterr, optopt;
```

General Description

A command-line parser that can be used by applications that follow Utility Syntax Guidelines 3, 4, 5, 6, 7, 9 and 10 in *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 extension: If the external variable *optind* is set to h`.`, getopt() restarts 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 getopt() returns thread-specific data, you can use it safely from a multithreaded application.

Returned Value

Returns one of the following values:

- If successful, getopt() returns the value of the next option character from *argv* that matches a character in *optstring*.
- If getopt() detects a missing option-argument and the first character of *optstring* was a colon (:), it returns a colon (:) and sets the variable *optopt* to the option character that caused the error.
- If getopt() encounters an option character that is not contained in *optstring*, it returns a question mark (?).
- If getopt() detects a missing option-argument and the first character of `optstring` was not a colon (:), it returns a question mark (?) and 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 getopt() utility.
- When all command line errors have been parsed or an unexpected error is encountered in the command line, getopt() returns a value of -1.
- The getopt() function does not return any errno values.
- If getopt() detects a missing argument or an option character not contained in *optstring*, it will write an error message to `stderr` describing the option character in error and the invoking program.

The getopt() function sets the external variables *optind*, *optarg* and *optopt* as described in *X/Open CAE Specification System Interfaces and Headers, Issue 4, Version 2* for the getopt() function.

For multithreaded applications you should use the following functions, defined in <stdio.h>, when referencing or changing the *optind*, *optopt*, *optarg*, and *opterr* external variables:

```
int *__opindf(void);

int *__opoptf(void);

char **__opargf(void);

int *__operrf(void);
```

You should also use these functions for invoking getopt() in a DLL. The functions return a pointer to a thread-specific value for each variable.

Related Information

- "getsubopt() — Parse Suboption Arguments" on page 375

getpagesize() — Get the Current Page Size

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

Format

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

int getpagesize(void);
```

General Description

Returns the current page size. The getpagesize() function is equivalent to sysconf() (_SC_PAGE_SIZE) and sysconf() (_SC_PAGESIZE).

Note: getpagesize() is supported for legacy usage. Because getpagesize() returns a size result which is limited to an integer, it is recommended that applications use the sysconf() function instead of this function.

Returned Value

Returns the current page size.

Related Information

- "unistd.h" on page 51
- "sysconf() — Determine System Configuration Options" on page 856

getpass() — Read a Character String Without Echo

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

Format

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

char *getpass(const char *prompt);
```

General Description

Opens the process' controlling terminal, writes to that device the null-terminated string *prompt*, reads a character string up to the next new-line character or EOF, restores the terminal state, and closes the terminal.

Note: The password will be displayed on the controlling terminal.

The getpass() function only works in an environment where either a controlling terminal exists, or `stdin` and `stderr` refer to *tty* devices.

Returned Value

If successful, getpass() returns a pointer to a null-terminated string of at most {PASS_MAX} bytes that were read from the terminal device.

If getpass() encounters an error, it restores the terminal state and returns a null pointer.

Related Information

- "unistd.h" on page 51

getpeername() — Get Name of Peer Connected to a Socket

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

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 *address, int address_len);
```

General Description

Returns the name of the peer connected to socket descriptor *socket*. The *address_len* parameter must be initialized to indicate the size of the space pointed to by *address* 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.

socket
The socket descriptor.

address
The internet address of the connected socket that is filled by getpeername() before it returns. The exact format of *address* is determined by the domain in which communication occurs.

address_len
The size of the address structure pointed to by *address* in bytes.

Returned Value

If successful, getpeername() returns a value of `0`.

If unsuccessful, getpeername() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EBADF
The *socket* parameter is not a valid socket descriptor.

EFAULT
Using the *address* and *address_len* parameters, as specified, would result in an attempt to access storage outside of the caller's virtual machine.

EINVAL
The *address_len* parameter is not a valid length. The socket has been shut down.

ENOBUFS
The getpeername() function 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 69
- “bind() — Bind a Name to a Socket” on page 92
- “connect() — Connect a Socket” on page 138
- “getsockname() — Get Name of a Socket” on page 368
- “socket() — Create a Socket” on page 791

getpgrp() — Get the Process Group ID

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

Format

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

pid_t getpgrp(void);
```

General Description

Finds the process group ID of the calling process.

The getpgrp() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

Returns the found value. The getpgrp() function is always successful.

No errnos are defined for getpgrp().

Example

CBC3BG13

```
/* CBC3BG13
   This example gets the process group ID.
 */
#define _OPEN_SYS
#include <unistd.h>
#include <wait.h>

main() {
  int status;

   printf("pid is %d, process group id is %d\n",
             (int) getpid(), (int) getpgrp());
    printf("the parent's pid is %d\n", (int) getppid());
}
```

Output

```
pid is 5373959, process group id is 5111816
the parent's parent's pid is 5111816
```

Related Information

- “sys/types.h” on page 46
- “setpgid() — Set Process Group ID for Job Control” on page 729
- “setsid() — Create Session, Set Process Group ID” on page 733

getpid() — Get the Process ID

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

Format

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

pid_t getpid(void);
```

General Description

Finds the process ID (PID) of the calling process.

The getpid() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

Returns the found value. The getpid() function is always successful.

No errnos are defined for getpid().

Example

CBC3BG14

```
/* CBC3BG14 */
#define _OPEN_SYS
#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 46
- “exec Functions” on page 192
- “getppid() — Get the Parent Process ID” on page 356
- “kill() — Send a Signal to a Process” on page 439

getppid() — Get the Parent Process ID

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

Format

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

pid_t getppid(void);
```

General Description

Gets the parent process ID (PPID).

The getppid() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

Returns the parent process ID. The getppid() function is always successful.

No errnos are defined for getppid().

Example

CBC3BG15

```
/* CBC3BG15 */
#define _OPEN_SYS
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <spawn.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 spawn child\n", (int) getpid());

  if (spawn("CBC3BG16", 0,0,0,0,0) == -1)
    perror("spawn() failed");
  else {
    alarm(30);
    while (footprint == 0);
  sleep(1);
  wait(&status);
  puts("parent is exiting");
  }
}

main() {


printf("child is sending SIGUSR2 to pid %d\n", (int) getppid());
kill(getppid(), SIGUSER2);
exit(0);

}
```

```
/* CBC3BG16 */
#define _OPEN_SYS
#include <sys/types.h>
#include <unistd.h>
```

```
#include <signal.h>
#include <stdio.h>

main() {


  printf("child is sending SIGUSR2 to pid %d\n", (int) getppid());
  kill(getppid(), SIGUSR2);
  exit(0);

}
```

Output

```
parent (pid 6094854) is about to spawn child
child is sending SIGUSR2 to pid 6094854
caught SIGUSR2
parent is exiting
```

Related Information

- "sys/types.h" on page 46
- "exec Functions" on page 192
- "getpid() — Get the Process ID" on page 355
- "kill() — Send a Signal to a Process" on page 439

getprotobyname() — Get a Protocol Entry by Name

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

Searches the ETC PROTO file for the specified protocol name and returns a pointer to a `protoent` structure for the network protocol.

name
The name of the protocol.

The getprotobyname(), getprotobynumber(), and getprotoent() functions 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 header file and contains the following elements:

p_aliases
A zero-terminated array of alternative names for the protocol.

p_name
The official name of the protocol.

p_proto
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 static 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 185
- "getprotobynumber() — Get a Protocol Entry by Number" on page 359
- "getprotoent() — Get Next Protocol Entry" on page 360
- "setprotoent() — Open Protocol Information File" on page 731

getprotobynumber() — Get a Protocol Entry by Number

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

Searches the ETC PROTO file for the specified protocol number and returns a pointer to a `protoent` structure for the network protocol.

proto
The protocol number.

The getprotobyname(), getprotobynumber(), and getprotoent() functions 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 header file and contains the following elements:

p_aliases
A zero-terminated array of alternative names for the protocol.

p_name
The official name of the protocol.

p_proto
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 static data that is overwritten by subsequent calls. 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 185
- "getprotobyname() — Get a Protocol Entry by Name" on page 358
- "getprotoent() — Get Next Protocol Entry" on page 360
- "setprotoent() — Open Protocol Information File" on page 731

getprotoent() — Get Next Protocol Entry

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

Reads the ETC PROTO file and returns a pointer to the next entry in the ETC PROTO file.

The getprotobyname(), getprotobynumber(), and getprotoent() functions 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 header file and contains the following elements:

p_aliases
A zero-terminated array of alternative names for the protocol.

p_name
The official name of the protocol.

p_proto
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 static data that is overwritten by subsequent calls. 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 185
- "getprotobyname() — Get a Protocol Entry by Name" on page 358
- "getprotobynumber() — Get a Protocol Entry by Number" on page 359
- "setprotoent() — Open Protocol Information File" on page 731

getpwnam() — Access the User Database by User Name

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

Format

```
#define _OPEN_SYS
#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

The getpwnam() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

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.

The getpwnam() function can also return ECMSERR or ECPERR if a system error occurs while retrieving the information.

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 _OPEN_SYS
#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 37
- “sys/types.h” on page 46
- “getlogin() — Get the User Login Name” on page 344
- “getpwnam() — Access the User Database by User Name” on page 361
- “getpwuid() — Access the User Database by User ID” on page 362

getpwuid() — Access the User Database by User ID

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

Format

```
#define _OPEN_SYS
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);
```

General Description

Gets information about a user with the specified *uid*. The getpwuid() function 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

The getpwuid() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, getpwuid() returns a pointer.

If unsuccessful, getpwuid() returns a NULL pointer.

The getpwuid() function can also return ECMSERR or ECPERR if a system error was detected while retrieving the information.

Return values may point to the static data that is overwritten on each call.

Example

CBC3BG17

```
/* CBC3BG17
   This example provides information for user ID 0.
 */
#define _OPEN_SYS
#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 37
- "sys/types.h" on page 46
- "getlogin() — Get the User Login Name" on page 344

gets() — Read a String

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 places the null character (`\0`) immediately after the last byte read. If an error occurs, the value stored in *buffer* is undefined.

The gets() function is not supported for files opened with `type=record`.

The gets() function 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, an intervening flush or reposition must occur; Between a read and a subsequent write, an intervening flush or reposition must also occur, unless an EOF has been reached.

Returned Value

If successful, gets() returns its argument.

If unsuccessful, gets() returns a NULL pointer, indicating an error or EOF condition with no characters read.

Note: You should use ferror() or feof() to determine whether an error or an EOF condition occurred. An 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 40
- "feof() — Test End-of-File Indicator" on page 219
- "ferror() — Test for Read/Write Errors" on page 220
- "fgets() — Read a String from a Stream" on page 238
- "fputs() — Write a String" on page 273
- "puts() — Write a String" on page 615

getservbyname() — Get a Server Entry by Name

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

Searches the ETC SERVICES file for the first entry that matches the specified service name and protocol name. If proto is NULL, only the service name must match.

name
The service name.

proto
The protocol name.

The getservbyname() call returns a pointer to a `servent` structure for the network service specified on the call.

The getservbyname(), getservbyport(), and getservent() functions 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 header file and contains the following elements:

s_aliases
A zero-terminated array of alternative names for the service.

s_name
The official name of the service.

s_port
The port number of the service.

s_proto
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 static data that is overwritten by subsequent calls. 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 185
- "getservbyport() — Get a Service Entry by Port" on page 366
- "getservent() — Get Next Service Entry" on page 367
- "setservent() — Open Network Services Information File" on page 732

getservbyport() — Get a Service Entry by Port

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

Searches the ETC SERVICES file for the first entry that matches the specified port number and protocol name. If proto is NULL, only the port number must match.

port
The port number.

proto
The protocol name.

The getservbyport() call returns a pointer to a `servent` structure for the port number specified on the call.

The getservbyname(), getservbyport(), and getservent() functions 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 header file and contains the following elements:

s_aliases
A zero-terminated array of alternative names for the service.

s_name
The official name of the service.

s_port
The port number of the service.

s_proto
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 static data that is overwritten by subsequent calls. 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 185
- "getservbyname() — Get a Server Entry by Name" on page 365
- "getservent() — Get Next Service Entry" on page 367
- "setservent() — Open Network Services Information File" on page 732

getservent() — Get Next Service Entry

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

Reads the next line of the ETC SERVICES file and returns a pointer to the next entry in the ETC SERVICES file.

The getservbyname(), getservbyport(), and getservent() functions 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 header file and contains the following elements:

s_aliases
A zero-terminated array of alternative names for the service.

s_name
The official name of the service.

s_port
The port number of the service.

s_proto
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 static data that is overwritten by subsequent calls. 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 185
- "getservbyname() — Get a Server Entry by Name" on page 365
- "getservbyport() — Get a Service Entry by Port" on page 366
- "setservent() — Open Network Services Information File" on page 732

getsockname() — Get Name of a Socket

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

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 *address, int address_len);
```

General Description

Stores the current name for the socket specified by the *socket* parameter into the structure pointed to by the *address* 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 AF_INET 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; an unbound socket in the AF_INET6 domain would cause the name to point to a `sockaddr_in6` structure with the *sin_family* field set to AF_INET6 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.

socket
The socket descriptor.

address
The address of the buffer into which getsockname() copies the name of *socket*.

address_len
Must initially point to an integer that contains the size in bytes of the storage pointed to by *address*. Upon return, that integer contains the size of the data returned in the storage pointed to by *address*.

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

If successful, getsockname() returns a value of `0`.

If unsuccessful, getsockname() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EBADF
The *socket* parameter is not a valid socket descriptor.

EFAULT
Using the *address* and *address_len* parameters, as specified, would result in an attempt to access storage outside of the caller's virtual machine.

ENOBUFS
The getsockname() function 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 69
- “bind() — Bind a Name to a Socket” on page 92
- “connect() — Connect a Socket” on page 138
- “getpeername() — Get Name of Peer Connected to a Socket” on page 353
- “socket() — Create a Socket” on page 791

getsockopt() — Get Options Associated with a Socket

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

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

Gets options associated with an AF_IUCV, AF_INET, or AF_INET6 socket. Not all options are supported by all address families. See each option for details. Options can exist at multiple protocol levels.

socket
The socket descriptor.

level
The level for which the option is 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_len
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, you must set the *level* parameter 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. You can use the getprotobyname() call to return the protocol number for a named protocol.

Use the *option_value* and *option_len* parameters to return data used by the particular get command, specified in *option_name*. 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. You must initially set it 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:

IP_MULTICAST_TTL
Gets the IP time-to-live of outgoing multicast datagrams. The TTL value is passed back as `u_char`.

IP_MULTICAST_LOOP
Determines whether loopback is enabled or disabled. The loopback indicator is passed back as `u_char`. A value of `0` means loopback is disabled; a value of 1 means it is enabled.

IP_MULTICAST_IF
Gets the interface IP address used for sending outbound multicast datagrams. The IP address is passed back using struct `in_addr`.

The following options are recognized at the IPv6 level:

Option
Description

IPV6_CHECKSUM
Used to determine if checksum processing is enabled for a RAW (non-ICMPv6) socket. The option value returned is the offset into the user data where the checksum is located. It is passed back as int. A value of -1 means the function is disabled.

IPV6_MULTICAST_HOPS
Returns the hop limit value for outbound multicast datagrams. The hop limit value is passed back as int.

IPV6_MULTICAST_IF
Returns the interface index for the interface used for sending outbound multicast datagrams. The interface index is passed back using struct u_int.

IPV6_MULTICAST_LOOP
Used to determine whether loopback of outgoing multicast packets is enabled or disabled. The loopback indicator is passed back as u_int. 0 means the function is disabled and 1 means it is enabled.

IPV6_PKTINFO
Returns the source IP address for an outgoing packet and the outgoing interface. If a setsockopt() has been done for this option, the value from the setsockopt() will be returned. It is passed back in an ipv6_pktinfo structure as defined in **netinet/in.h**.

IPV6_RECVHOPLIMIT
Indicates whether the function to return the received hop limit as ancillary data is enabled or disabled. The option value is passed back as int. 0 means the function is disabled and 1 means it is enabled.

IPV6_RECVPKTINFO
Indicates whether the function is to return the destination IP address and incoming interface is enabled or disabled. The option value is passed back as int. 0 means the function is disabled and 1 means it is enabled.

IPV6_UNICAST_HOPS
Returns the hop limit value for outbound unicast datagrams. The hop limit value is passed back as int.

IPV6_USE_MIN_MTU
Indicates whether the IP layer will use the minimum MTU size (1280) for sending packets, bypassing path MTU discovery. The option value is passed back as int. 1 means the function is enabled and the minimum MTU size will be used. 0 means that the function is disabled.

IPV6_V6ONLY
Used to determine whether a socket is restricted to IPv6 communications only. The option value is passed back as int. A non-zero value means the option is enabled (socket can only be used for IPv6 communications). 0 means the option is disabled.

The following option is recognized at ICMPv6 level:

Option
Description

ICMP6_FILTER
Used to filter ICMPv6 messages. It returns the filter value being used for this socket. It is passed back in an icmp6_filter structure as defined in **netinet/icmp6.h**.

The following options are recognized at the socket level:

SO_BROADCAST
Interrogates the ability to broadcast messages. If this option is enabled, it allows the application to send broadcast messages over *socket* if the interface specified in the destination supports the broadcasting of packets. This option has no meaning for stream sockets.

SO_DEBUG
Reports whether debugging information is being recorded. This option stores an `int` value.

SO_ERROR
Returns any pending error on the socket and clears the error status. You can use SO_ERROR 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_KEEPALIVE
Interrogates 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 and AF_INET6 domains.

SO_LINGER
Interrogates whether close() will linger 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 virtual machine waits to try to send the data. Although the data transfer is usually successful, it cannot be guaranteed because the TCP/IP virtual machine 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.

For AF_IUCV sockets, the default behavior of close() is to block the closing application until all pending data is written. When this option is enabled and l_linger is zero, a close() call will immediately close the connection and unsent data is discarded. When this option is enabled and l_linger is non-zero, a close() call will cause the calling application to block until all unsent data is transmitted or the connection has timed out.

SO_OOBINLINE
Describes the handling 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 having 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 by specifying the MSG_OOB flag in those calls. This option has meaning only for stream sockets.

SO_RCVBUF
Reports receive buffer size information. This option stores an `int` value.

SO_REUSEADDR
Interrogates 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 and AF_INET6 domains.

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 the AF_UNIX, AF_INET, and AF_INET6 domains.

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, getsockopt() returns a value of `0`.

If unsuccessful, getsockopt() returns a value of `-1` and sets errno to one of the folowing values under the conditions described:

EAFNOSUPPORT
The address family is not supported.

EBADF
The *socket* parameter is not a valid socket descriptor.

EFAULT
Using the *option_value* and *option_len* parameters would result in an attempt to access storage outside the caller's virtual machine.

EINVAL
The specified option is invalid at the specified socket level.

ENOBUFS
Buffer space is not available to send the message.

ENOPROTOOPT
The *option_name* parameter is unrecognized. This error is also returned if getsockopt() is attempted on an AF_UNIX 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.

EOPNOTSUPP
The following options are not supported:

- IPV6_JOIN_GROUP
- IPV6_LEAVE_GROUP

Example

The following are examples of the getsockopt() call. For examples of how the setsockopt() call options are set, see “setsockopt() — Set Options Associated with a Socket” on page 734.

```
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 92
- “close() — Close a File” on page 127
- “getprotobyname() — Get a Protocol Entry by Name” on page 358
- “setsockopt() — Set Options Associated with a Socket” on page 734
- “socket() — Create a Socket” on page 791

getstablesize() — Get Socket Table Size

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	

Format

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

int getstablesize(void);
```

General Description

Obtains the number of sockets that are allowed for use in bulk mode operations for a process.

Returned Value

Returns the current limit for this process. If it has not been changed by the maxdesc() function, the default is returned. The maximum allowed value is 65535. 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*.

No errno values are defined for getstablesize().

Related Information

- "sys/socket.h" on page 46
- "maxdesc() — Get Socket Numbers to Extend Beyond Default Range" on page 479

getsubopt() — Parse Suboption Arguments

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

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 cannot 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. The token vector is organized as a series of pointers to strings.

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 end of the token vector is identified by a null pointer.

When getsubopt() returns, if *valuep* is not a null pointer, the processed suboption argument 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 getsubopt() returns thread-specific data, you can use it safely from a multithreaded application.

Returned Value

If successful, getsubopt() returns the index of the matched token string,

If no token strings were matched, getsubopt() returns a value of -1.

The getsubopt() function does not set any errno values.

Related Information

- "stdlib.h" on page 42
- "getopt() — Parse Command Option" on page 351

getsyntx() — Return LC_SYNTAX Characters

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <variant.h>

struct variant *getsyntx(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 52.

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, getsyntx() returns a null pointer.

The structure returned is not modified by the program that getsyntx() 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 = getsyntx();
   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

- For information about locales and character sets, see *z/OS: XL C/C++ Programming Guide*.
- "locale.h" on page 30
- "variant.h" on page 52
- "setlocale() — Set Locale" on page 721

gettimeofday() — Get Date and Time

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

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 IBM XL C/C++ for z/VM 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 an IBM XL C/C++ for z/VM pragma map statement for an IBM XL C/C++ for z/VM version of gettimeofday() which ignores *tzp*.

If _ALL_SOURCE is defined, the IBM XL C/C++ for z/VM preprocessor includes a prototype for gettimeofday() which defines *tzp* as a pointer to a timezone structure and includes a pragma map statement for an IBM XL C/C++ for z/VM 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, gettimeofday():

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

Returns a value of `0` unless overflow occurs. If overflow occurs, gettimeofday() returns a nonzero value. 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 28
- “sys/time.h” on page 46
- “ctime() — Convert Time to a Character String” on page 153
- “ftime() — Get the Date and Time” on page 302

getuid() — Get the Real User ID

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

Format

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

uid_t getuid(void);
```

General Description

Finds the real user ID (UID) of the calling process.

The getuid() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

Returns the found value. The getuid() function is always successful.

No errnos are defined for this function.

Example

CBC3BG20

```
/* CBC3BG20
   This example provides information for your user ID.
 */
#define _OPEN_SYS
#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  : vmusr1
pw_uid   : 25
pw_gid   : 500
pw_dir   : /u/vmusr1
pw_shell : /bin/sh
```

Related Information

"sys/types.h" on page 46
"geteuid() — Get the Effective User ID" on page 327
"seteuid() — Set the Effective User ID" on page 710
"setuid() — Set the Effective User ID" on page 739

getutxent() — Read Next Entry in utmpx Database

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

Format

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

struct utmpx *getutxent(void);
```

General Description

Reads in the next entry from the `utmpx database`. If the database is not already open, getutxent() opens it. If it reaches the end of the database, getutxent() fails.

The getutxent(), getutxid(), and getutxline() functions obtain a shared lock on the database. The pututxline() function 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, using the fcntl() command F_SETLKW on the first record in the database.

Because getutxent() returns thread-specific data, you can use it safely from a multithreaded 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 calls endutxent().

The name of the database file defaults to /etc/utmpx. To process a different database file name, you must use the __utmpxname() function.

For all entries that match a request, the `ut_type` member indicates the type of entry. Other members of the entry will contain meaningful data based on the value of the `ut_type` member as follows:

EMPTY
No other members have meaningful data.

BOOT_TIME
`ut_tv` is meaningful.

__RUN_LVL
`ut_tv` and `ut_line` are meaningful

OLD_TIME
`ut_tv` is meaningful.

NEW_TIME
`ut_tv` is meaningful.

USER_PROCESS
`ut_id`, `ut_user` (login name of the user), `ut_line`, `ut_pid`, and `ut_tv` are meaningful.

INIT_PROCESS
`ut_id`, `ut_pid`, and `ut_tv` are meaningful.

LOGIN_PROCESS
`ut_id`, `ut_user` (implementation-specific name of the login process), *ut_pid*, and *ut_tv* are meaningful.

DEAD_PROCESS
`ut_id`, `ut_pid`, and `ut_tv` 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.

If unsuccessful, getutxent() returns a null pointer.

No errnos are defined for getutxent().

Related Information

- "endutxent() — Close the utmpx Database" on page 186
- "getutxid() — Search by ID utmpx Database" on page 380
- "getutxline() — Search by Line utmpx Database" on page 381
- "pututxline() — Write Entry to utmpx Database" on page 616
- "setutxent() — Reset to Start of utmpx Database" on page 741
- "__utmpxname() — Change the utmpx Database Name" on page 915

getutxid() — Search by ID utmpx Database

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

Format

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

struct utmpx *getutxid(const struct utmpx *id);
```

General Description

Searches forward from the current point in the `utmpx` database.

- If the database is not already open, getutxid() 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 getutxid() 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 getutxid() 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 getutxid() reaches the end of the database without a match, getutxid() fails.

The getutxent(), getutxid(), and getutxline() functions obtain a shared lock on the database. The pututxline() function 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, using the fcntl() command F_SETLKW on the first record in the database.

Because getutxid() returns thread-specific data, you can use it safely from a multithreaded 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 calls endutxent().

The name of the database file defaults to /etc/utmpx. To process a different database file name, you must use the __utmpxname() function.

For all entries that match a request, the `ut_type` member indicates the type of entry. Other members of the entry will contain meaningful data based on the value of the `ut_type` member as follows:

EMPTY
No other members have meaningful data.

BOOT_TIME
`ut_tv` is meaningful.

__RUN_LVL
`ut_tv` and `ut_line` are meaningful

OLD_TIME
`ut_tv` is meaningful.

NEW_TIME
`ut_tv` is meaningful.

USER_PROCESS
`ut_id`, `ut_user` (login name of the user), `ut_line`, `ut_pid`, and `ut_tv` are meaningful.

INIT_PROCESS
`ut_id`, `ut_pid`, and `ut_tv` are meaningful.

LOGIN_PROCESS
`ut_id`, `ut_user` (implementation-specific name of the login process), *ut_pid*, and *ut_tv* are meaningful.

DEAD_PROCESS
`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.

If unsuccessful, getutxid() returns a null pointer.

No errnos are defined for getutxid().

Related Information

- "endutxent() — Close the utmpx Database" on page 186
- "getutxent() — Read Next Entry in utmpx Database" on page 379
- "getutxline() — Search by Line utmpx Database" on page 381
- "pututxline() — Write Entry to utmpx Database" on page 616
- "setutxent() — Reset to Start of utmpx Database" on page 741
- "__utmpxname() — Change the utmpx Database Name" on page 915

getutxline() — Search by Line utmpx Database

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

Format

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

struct utmpx *getutxline(const struct utmpx *line);
```

General Description

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 *line*. If the database is not already open, `getutxline` opens it. If `getutxline` reaches the end of the database, `getutxline` fails.

The getutxent(), getutxid(), and getutxline() functions obtain a shared lock on the database. The pututxline() function 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, using the fcntl() command F_SETLKW on the first record in the database.

Because getutxline() returns thread-specific data, you can use it safely from a multithreaded 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 calls endutxent().

The name of the database file defaults to /etc/utmpx. To process a different database file name, you must use the __utmpxname() function.

The getutxent(), getutxid(), and getutxline() functions cache the last entry read from the database. For this reason, in order to use `getutxline` to search for multiple occurrences, you must 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 entry. Other members of the entry will contain meaningful data based on the value of the `ut_type` member as follows:

EMPTY
No other members have meaningful data.

BOOT_TIME
`ut_tv` is meaningful.

__RUN_LVL
`ut_tv` and `ut_line` are meaningful

OLD_TIME
`ut_tv` is meaningful.

NEW_TIME
`ut_tv` is meaningful.

USER_PROCESS
`ut_id`, `ut_user` (login name of the user), `ut_line`, `ut_pid`, and `ut_tv` are meaningful.

INIT_PROCESS
`ut_id`, `ut_pid`, and `ut_tv` are meaningful.

LOGIN_PROCESS
`ut_id`, `ut_user` (implementation-specific name of the login process), *ut_pid*, and *ut_tv* are meaningful.

DEAD_PROCESS
`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.

If unsuccessful, getutxline() returns a null pointer.

No errnos are defined for getutxline().

Related Information

- “endutxent() — Close the utmpx Database” on page 186
- “getutxent() — Read Next Entry in utmpx Database” on page 379
- “getutxid() — Search by ID utmpx Database” on page 380
- “pututxline() — Write Entry to utmpx Database” on page 616
- “setutxent() — Reset to Start of utmpx Database” on page 741
- “__utmpxname() — Change the utmpx Database Name” on page 915

getw() — Get a Machine Word from a Stream

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

Format

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

int getw(FILE *stream);
```

General Description

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 will mark the *st_atime* field of the file associated with *stream* for update. This will occur with 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.

For errno values, see “fgetc() — Read a Character” on page 235.

Because the representation of EOF is a valid integer, applications should use ferror() and feof() for error-checking.

Related Information

- “stdio.h” on page 40
- “fopen() — Open a File” on page 254
- “fwrite() — Write Items” on page 308
- “putw() — Put a Machine Word on a Stream” on page 618

getwc() — Get a Wide Character

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 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().

The getwc() function 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition must also occur, unless an EOF has been reached.

Returned Value

Returns the next wide character from the input stream pointed to by *stream*; otherwise, the function returns WEOF. If an error occurs, getwc() sets the error indicator. If an EOF is encountered, it sets the EOF indicator. If an encoding error is encountered, it sets EILSEQ in errno.

Note: You should use ferror() or feof() to determine whether an error or an EOF condition occurred. An 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 40
- "wchar.h" on page 53
- "fgetwc() — Get Next Wide Character" on page 239

getwchar() — Get a Wide Character

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 `stdin` argument.

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`; otherwise, getwchar() returns WEOF. If the stream is at EOF, or a read error occurs, getwchar() returns WEOF and sets the EOF indicator for the stream. If an encoding error occurs, getwchar() returns WEOF and stores the value of the macro EILSEQ in errno.

Note: You should use ferror() or feof() to determine whether an error or an EOF condition occurred. An 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 53
- "fgetwc() — Get Next Wide Character" on page 239
- "getwc() — Get a Wide Character" on page 384

getwmccoll() — Get Next Collating Element from Wide String

Standards / Extensions	C or C++	Dependencies
C Library	both	

Format

```
#include <collate.h>

collel_t getwmccoll(wchar_t **src)
```

General Description

If the object pointed to by *src* is not a null pointer, getwmccoll() determines the longest sequence of wide characters in the array pointed to by *str* that constitute a valid multiwide-character collating element. It then produces the value of type `collel_t` corresponding to that collating element. The object pointed to by *src* is assigned the address just past the last wide character of the multiwide-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, getwmccoll() returns a value of `0`. If the object pointed to by *src* points to an invalid wide character, getwmccoll() returns a value of -1 and sets errno to EILSEQ. Otherwise, it returns the value of type `collel_t` that represents the collating element found.

Related Information

- "collate.h" on page 20
- "wchar.h" on page 53
- "cclass() — Return Characters in a Character Class" on page 104
- "collequiv() — Return List of Equivalent Collating Elements" on page 132
- "collorder() — Return List of Collating Elements" on page 133
- "collrange() — Calculate the Range List of Collating Elements" on page 134
- "colltostr() — Return a String for a Collating Element" on page 136

- “getmccoll() — Get Next Collating Element from String” on page 345
- “ismccollel() — Identify a Multicharacter Collating Element” on page 429
- “maxcoll() — Return Maximum Collating Element” on page 479
- “strtocoll() — Return Collating Element for String” on page 839

givesocket() — Make Specified Socket Available

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	

Format

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

int givesocket(int d,struct clientid *clientid);
```

General Description

Makes the specified socket available to a takesocket() call issued by another program. Any AF_INET or AF_INET6 stream socket can be given. Typically, givesocket() is used by a main program that obtains sockets by means of accept() and gives them to application programs that handle one socket at a time.

Note: You can give an AF_INET socket only to an AF_INET socket, and an AF_INET6 socket only to an AF_INET6 socket; that is, you cannot give an AF_INET6 socket to an AF_INET socket or vice versa.

d
The descriptor of a socket to be given to another application.

clientid
A pointer to a `clientid` structure specifying the program to which the socket is to be given.

To pass a socket, the giving program first calls givesocket() with the `clientid` structure filled in as follows:

The `clientid` structure:

```
      struct clientid {
         int domain;
             name[8];
         char subtaskname[8];
             reserved[20];
      };
```

domain
The domain of the input socket descriptor. It must be AF_INET or AF_INET6.

name
Specifies the VM user ID of the virtual machine that is running the target program, left-justified and padded with blanks. The target program can run in the same virtual machine as your program, in which case your program sets this field to its own VM user ID.

subtaskname
Specifies the "subtaskname" used by the target program when it established its IUCV connection with the TCP/IP server virtual machine. Alternatively, 8 blank characters may be specified to indicate that any application running in the target virtual machine can do the takesocket() call. If blanks are specified, the first application in the target virtual machine that issues the takesocket() call with the proper client ID of the giving program and proper socket number will become the new owner of the socket. Otherwise, only the application with the specified subtaskname will be authorized to take the socket.

reserved
Should contain blanks.

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 a value of -1, with errno set to EBADF.

Note: For backward compatibility, a client ID can point to the `clientid` structure obtained when the target program calls getclientid(). In this case, only the target program, and no other programs in the target program's virtaul machine, can take the socket.

Returned Value

If successful, givesocket() returns a value of `0`.

If unsuccessful, givesocket() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EAFNOSUPPORT
The address family is not supported.

EBADF
The *d* parameter is not a valid socket descriptor. The socket has already been given.

EBUSY
Listen() has been called for the socket (that is, it is not an *active* socket).

EFAULT
Using the *clientid* parameter, as specified, would result in an attempt to access storage outside the caller's virtual machine.

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.

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

- “sys/socket.h” on page 46
- “accept() — Accept a New Connection on a Socket” on page 69
- “close() — Close a File” on page 127
- “getclientid() — Get Identifier for Calling Application” on page 316
- “listen() — Prepare Server for Incoming Client Requests” on page 452
- “select() — Monitor Activity on Files/Sockets and Message Queues” on page 680
- “takesocket() — Acquire a Socket from Another Program” on page 860

glob() — Generate Path names Matching a Pattern

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

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

A path name 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 File name Expansion".

The `glob_t` structure 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 file names.

gl_offs
: Slots to reserve at the beginning of *gl_pathv*.

The *pattern* argument is a pointer to a path name pattern that will be expanded. The glob() function matches all accessible path names against this pattern and develops a list of all path names that match.

In order to have access to a path name, glob() requires search permission on every component of a path except the last, and read permission on each directory of any file name component of *pattern* that contains any of the following special characters:

```
        *        ?        [
```

The glob() function stores the number of matched path names into *pglob->gl_pathc* and a pointer to a list of pointers to path names into *pglob->gl_pathv*. The sort order of the path names is 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 path name is a null pointer. If the pattern does not match any path names, the returned number of matched paths is set to `0` and the contents of *pglob->gl_pathv* are implementation-dependent.

The caller is responsibile for creating 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 controls 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 file <glob.h>:

GLOB_APPEND
: Appends a new set of path names to the ones from a previous call to glob().

GLOB_DOOFFS
: Uses *pglob->gl_offs* 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* path name 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
Appends a slash to each path name that is a directory that matches *pattern*.

GLOB_NOCHECK
Supports rule 3 in Section 2.13.3, "Patterns Used for File name Expansion", in *X/Open CAE Specification, Commands and Utilities, Issue 4, Version 2*. If *pattern* does not match any path name, then glob() returns a list consisting of only *pattern*; and the number of matched path names is 1.

GLOB_NOESCAPE
Disables backslash escaping.

GLOB_NOSORT
Returns path names in no specific order. Ordinarily, glob() sorts the matching path names according to 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.)

Rules: The following rules apply when making two or more calls to glob() 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 path names that were in the *pglob->gl_pathv* list before the call, in the same order as before.
   c. Pointers to the new path names generated by the second call, in the specified order.
4. The count returned in *pglob->gl_pathc* will be the total number of path names from the two calls.
5. The application can change any of the fields after a call to glob(), but it must reset them to the original value before a subsequent call 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 a value of `0`. The *pglob->gl_pathc* argument returns the number of matched path names, and the *pglob->gl_pathv* argument contains a pointer to a null-terminated list of matched and sorted path names. 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 path name, 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 a value of 0, the error is ignored.

Related Information

- "glob.h" on page 26
- "exec Functions" on page 192
- "fnmatch() — Match File Name or Path Name" on page 253
- "opendir() — Open a Directory" on page 531
- "readdir() — Read an Entry from a Directory" on page 632
- "stat() — Get File Information" on page 803
- "wordexp() — Perform Shell Word Expansions" on page 972

globfree() — Free Storage Allocated by glob()

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

Format

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

void globfree(glob_t *pglob);
```

General Description

Frees storage associated with *pglob* by a previous call to glob().

Returned Value

The globfree() function has no returned value.

Related Information

- "glob.h" on page 26
- "glob() — Generate Path names Matching a Pattern" on page 389

gmtime() — Convert Time to Broken-Down UTC Time

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. Coordinated Universal Time was formerly known as Greenwich mean time (GMT). The value of *timer* is usually obtained by calling time().

Table 18. Fields of the tm structure

Field	Description
`tm_sec`	Seconds (0-61)
`tm_min`	Minutes (0-59)
`tm_hour`	Hours (0-23)
`tm_mday`	Day of month (1-31)
`tm_mon`	Month (0-11; January = 0)
`tm_year`	Year (current year minus 1900)
`tm_wday`	Day of week (0-6; Sunday = 0)
`tm_yday`	Day of year (0-365; January 1 = 0)
`tm_isdst`	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, corresponding to calendar time pointed to by *timer*. The fields in `tm` are shown in Table 18 on page 392. 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.
- 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 timef() 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 49
- "asctime() — Convert Time to Character String" on page 78
- "ctime() — Convert Time to a Character String" on page 153
- "localtime() — Convert Time and Correct for Local Time" on page 456
- "mktime() — Convert Local Time" on page 505
- "time() — Determine Current Time" on page 885

hcreate() — Create Hash Search Tables

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

Format

```
#define _XOPEN_SOURCE
#include <search>

int hcreate(size_t nel);
```

General Description

Allocates sufficient space for a hash table containing *nel* elements, which is an estimate of the maximum number of entries that the table will contain. The function may increase the number for the actual table allocation in order to obtain certain mathematically favorable circumstances.

Note: You must call hcreate() before using hsearch().

For information on threading behavior, see "hsearch() — Search Hash Tables" on page 394.

Returned Value

If successful, hcreate() returns a nonzero value.

If hcreate() cannot allocate sufficient space for the table, it returns a value of `0` and sets errno to the following value:

ENOMEM
Insufficient storage space is available.

Related Information

- "search.h" on page 38
- "bsearch() — Search Arrays" on page 100
- "hdestroy() — Destroy Hash Search Tables" on page 394
- "hsearch() — Search Hash Tables" on page 394
- "lsearch() — Linear Search and Update" on page 468
- "malloc() — Reserve Storage Block" on page 477
- "strcmp() — Compare Strings" on page 812

- “tsearch() — Search Binary Tree” on page 896

hdestroy() — Destroy Hash Search Tables

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

Format

```
#define _XOPEN_SOURCE
#include <search>

void hdestroy(void);
```

General Description

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.

For information on threading behavior, see “hsearch() — Search Hash Tables” on page 394.

Returned Value

The hdestroy() function has no returned value.

Related Information

- “search.h” on page 38
- “bsearch() — Search Arrays” on page 100
- “hcreate() — Create Hash Search Tables” on page 393
- “hsearch() — Search Hash Tables” on page 394
- “lsearch() — Linear Search and Update” on page 468
- “malloc() — Reserve Storage Block” on page 477
- “strcmp() — Compare Strings” on page 812
- “tsearch() — Search Binary Tree” on page 896

hsearch() — Search Hash Tables

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

Format

```
#define _XOPEN_SOURCE
#include <search>

ENTRY *hsearch(ENTRY item, ACTION action);
```

General Description

A hash-table search routine that 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>` 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

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 is the possible value of errno:

ENOMEM
Insufficient storage space is available.

Related Information

- "search.h" on page 38
- "bsearch() — Search Arrays" on page 100
- "hcreate() — Create Hash Search Tables" on page 393
- "hdestroy() — Destroy Hash Search Tables" on page 394
- "lsearch() — Linear Search and Update" on page 468
- "malloc() — Reserve Storage Block" on page 477
- "strcmp() — Compare Strings" on page 812
- "tsearch() — Search Binary Tree" on page 896

htonl() — Translate Address Host to Network Long

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

Translates a long integer from host byte order to network byte order.

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:

- Host byte order and network byte order are the same.
- Since this function is implemented as a macro, you need one of the feature test macros and the inet header file.

Returned Value

Returns the translated long integer.

Related Information

- "htons() — Translate an Unsigned Short Integer into Network Byte Order" on page 396
- "ntohl() — Translate a Long Integer into Host Byte Order" on page 525
- "ntohs() — Translate an Unsigned Short Integer into Host Byte Order" on page 526

htons() — Translate an Unsigned Short Integer into Network Byte Order

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

Translates a short integer from host byte order to network byte order.

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:

- Host byte order and network byte order are the same.
- Since this function is implemented as a macro, you need one of the feature test macros and the inet header file.

Returned Value

Returns the translated short integer.

Related Information

- "htonl() — Translate Address Host to Network Long" on page 395
- "ntohl() — Translate a Long Integer into Host Byte Order" on page 525
- "ntohs() — Translate an Unsigned Short Integer into Host Byte Order" on page 526

hypot() — Calculate Hypotenuse

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

Format

```
#include <math.h>

double hypot(double side1, double side2);
```

Compiler Option

LANGLVL(SAA), LANGLVL(SAAL2), or LANGLVL(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 63 for more information about IEEE floating-point.

Returned Value

Returns the calculated length of the hypotenuse. If the correct value is outside the range of representable values, hypot() returns plus or minus HUGE_VAL, according to the sign of the value and stores the value of the macro ERANGE in errno to show the calculated value is out of range. If the correct value would cause an underflow, hypot() returns a value of `0` and stores the value of the macro ERANGE 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 32
- "sqrt() — Calculate Square Root" on page 799

ibmsflush() — Flush Application-side Datagram Queue

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	

Format

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

int ibmsflush(int socket);
```

General Description

Bulk mode is supported only for AF_INET and AF_INET6 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.

socket
The socket descriptor.

For outbound sockets, the application-side datagram queue is flushed (transferred to the TCP/IP virtual machine) 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 setibmsockopt() is issued.

Returned Value

If successful, ibmsflush() returns a value of `0`.

If unsuccessful, ibmsflush() returns a value of `-1` and sets errno to the following value under the condition described:

EBADF
The *socket* 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 = setibmsockopt(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

- "getibmsockopt() — Get Options Associated with a Bulk Mode Socket" on page 341
- "setibmsockopt() — Set Options Associated with a Bulk Mode Socket" on page 713

iconv() — Code Conversion

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; *outbuf* 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 iconv() sets errno to EILSEQ. If the input buffer ends with an incomplete character or shift sequence, conversion stops after the previous successfully converted bytes and iconv() sets errno 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 iconv() function updates the variable pointed to by *inbuf* to point to the byte following the last byte of a successfully converted character, decrements the value pointed to by *inbytesleft* to reflect the number of bytes still not converted in the input buffer, updates the variable pointed to by *outbuf* to point to the byte following the last byte of converted output data, and decrements the value pointed to by *outbytesleft* to reflect the number of bytes still available in the output buffer. For state-dependent encoding, iconv() updates the conversion descriptor 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.

Special Considerations for Bidirectional Language Support

If the _BIDION environment variable is set to TRUE, iconv() performs bidirectional layout transformation to the converted characters. The required attributes for bidirectional layout transformation can be specified using the environment variable _BIDIATTR (eg. export _BIDIATTR="@ls typeoftext=visual:implicit, orientation=ltr:ltr,numerals=nominal:national"). For a detailed description of the bidirectional layout transformation, see “Bidirectional Language Support” in . If the environment variable _BIDIATTR is not set, the default values will be used.

iconv() can perform bidirectional layout transformation while converting the data from the fromCodePage to the toCodePage. Bidirectional layout transformation will take place only if bidirectional language support is activated, see “iconv_open() — Allocate Code Conversion Descriptor” on page 402 for more information about activating bidirectional layout transformation. In case iconv encounters any error in input or output buffers in the bidirectional part it will bypass the bidirectional layout transformation and continue its normal function as usual.

Special Behavior for POSIX C

In the POSIX environment, you may safely use within a single thread a conversion descriptor returned from a successful call to iconv_open(). In addition, it may be opened on one thread, used on a second thread (iconv()), and closed (iconv_open()) 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()). Undefined behavior may result if iconv() uses a shared conversion descriptor across multiple threads. For more information about using POSIX support, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

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 iconv() converts the entire string in the input buffer, 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 with errno 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 ()
{
   unsigned char   *inptr;  /* Pointer used for input buffer  */
   unsigned char   *outptr; /* Pointer used for output buffer */
   unsigned 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 = 0;
   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

- For information about locales and character sets, see *z/OS: XL C/C++ Programming Guide*.
- “locale.h” on page 30
- “iconv_close() — Deallocate Code Conversion Descriptor” on page 401
- “iconv_open() — Allocate Code Conversion Descriptor” on page 402
- “setlocale() — Set Locale” on page 721

iconv_close() — Deallocate Code Conversion Descriptor

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 iconv_open(). For an illustration of using iconv_open(), see "Example" on page 401.

Returned Value

If successful, iconv_close() returns a value of `0`.

If unsuccessful, iconv_close() returns a value of -1 and sets errno to indicate the error. If *cd* is not a valid descriptor, an error occurs and iconv_close() sets errno to EBADF.

Related Information

- For information about locales and characters, see *z/OS: XL C/C++ Programming Guide*.
- "locale.h" on page 30
- "iconv() — Code Conversion" on page 399
- "iconv_open() — Allocate Code Conversion Descriptor" on page 402
- "setlocale() — Set Locale" on page 721

iconv_open() — Allocate Code Conversion Descriptor

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 *fromcode* to the encoded character set specified in the array pointed to by *tocode*.

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 z/VM dataset name prefix to use to find UCS-2 tables if they cannot be found in the HFS.

For an illustration of using iconv_close(), see "Example" on page 401.

Special Considerations for Bidirectional Language Support

Performs all the initialization needed to activate the bidirectional layout transformation to be used by iconv. The following three conditions must be satisfied to enable the bidirectional layout transformation:

1. The _BIDION environment variable must be set to TRUE.
2. The current locale environment at iconv_open() time must be an Arabic or Hebrew locale (eg. Ar_AA or Iw_IL).
3. The conversion code set must be an Arabic or Hebrew code set.

Conversion code sets differ in the following three cases:

1. Case fromCodeSet is UCS-2 and toCodeSet is single byte code set. In this case toCodeSet must be an Arabic or Hebrew code set.
2. Case fromCodeSet is single byte code set and toCodeSet is UCS-2. In this case fromCodeSet must be an Arabic or Hebrew code set.
3. Case both fromCodeSet and toCodeSet are single byte code sets. In this case toCodeSet must be an Arabic or Hebrew code set.

iconv_open( ) checks for the existence of the environment variable _BIDIATTR to get the bidirectional layout transformation attributes. It will use default values in case _BIDIATTR is not defined, is unset, or in case of the existence of some erroneous values in the _BIDIATTR environment variable. The default values are code set dependent according to the Arabic or Hebrew code set used. For the Arabic 420 code set the default values will be: orientation RTL, type of text visual, shaping shaped, numerals national and swapping on. For the Hebrew 424 code set the default values will be: orientation RTL, type of text visual and swapping on. For the rest of the Arabic code sets the default values will be: orientation RTL, type of text implicit, shaping nominal, numerals national and swapping on.

iconv_open() uses the following environment variables.

_BIDION
Tells iconv_open() whether to activate bidirectional handling of the converted data or not. _BIDION can be assigned either the value TRUE, if you want to turn on bidirectional layout transformation, or the value FALSE, if you want to turn off the BiDi layout transformation. Bidirectional layout transformation can also be turned off if the variable_BIDION is not defined in the environment.

_BIDIATTR
Holds the bidirectional layout transformation attributes which will be used later by iconv, _BIDIATTR will be read only in iconv_open() time. The _BIDIATTR environment variable is in the form of input/output pairs separated by colon, at the beginning of the string there is an @ that identifies the beginning of the attributes list, then followed by the attributes in the form of

```
<attribute_name1>=<input1>:<output1>,
<attribute_name2>=<input2>:<output2>
```

```
..... (eg. export _BIDIATTR="@ls
typeoftext=visual:implicit,orientation=ltr:ltr, numerals=nominal:national").
```

Returned Value

If successful, iconv_open() returns a conversion descriptor.

If unsuccessful, iconv_open() returns `(iconv_t)-1`, and sets errno to indicate the error. If the conversion between specified encoded character sets is not supported, an error occurs, and errno is set to EINVAL, to show that the result of the conversion was not valid.

Related Information

- For information about locales and character sets, see *z/OS: XL C/C++ Programming Guide*.
- “locale.h” on page 30
- “iconv() — Code Conversion” on page 399

- “iconv_close() — Deallocate Code Conversion Descriptor” on page 401
- “setlocale() — Set Locale” on page 721

if_freenameindex() — Free the Memory Allocated by if_nameindex()

Standards

Standards / Extensions	C or C++	Dependencies
RFC2553	both	

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <net/if.h>

void if_freenameindex(struct if_nameindex *ptr);
```

General Description

The if_freenamindex() function frees the memory allocated by if_nameindex(). The *ptr* argument must be a pointer that was returned by if_nameindex().

Returned Value

No return value is defined.

Related Information

- “if_indextoname() — Map a Network Interface Index to Its Corresponding Name” on page 404
- “if_nameindex() — Return All Network Interface Names and Indexes” on page 405
- “if_nametoindex() — Map a Network Interface Name to Its Corresponding Index” on page 406

if_indextoname() — Map a Network Interface Index to Its Corresponding Name

Standards

Standards / Extensions	C or C++	Dependencies
RFC2553	both	

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <net/if.h>

char *if_indextoname(unsigned int ifindex, char *ifname);
```

General Description

The if_indextoname() function maps an interface index to its corresponding interface name. When this function is called, *ifname* must point to a buffer of at least IF_NAMESIZE bytes into which the interface name corresponding to interface index *ifindex* is returned. Otherwise, the function returns a NULL pointer and sets errno to indicate the error.

Returned Value

Error Code
Description

EINVAL
The *ifindex* parameter was zero, or the *ifname* parameter was NULL, or both.

ENOMEM
Insufficient storage is available to obtain the information for the interface name.

ENXIO
The *ifindex* does not yield an interface name.

Related Information

- "if_freenameindex() — Free the Memory Allocated by if_nameindex()" on page 404
- "if_nameindex() — Return All Network Interface Names and Indexes" on page 405
- "if_nametoindex() — Map a Network Interface Name to Its Corresponding Index" on page 406

if_nameindex() — Return All Network Interface Names and Indexes

Standards

Standards / Extensions	C or C++	Dependencies
RFC2553	both	

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <net/if.h>

struct if_nameindex *if_nameindex(void);
```

General Description

The if_nameindex() function returns an array of if_nameindex structures, one structure per interface. The end of the array is indicated by a structure with an if_index of zero and an if_name of NULL.

The if_nameindex structure holds the information about a single interface and is defined as a result of including the <net/if.h> header.

```
struct if_nameindex {
   unsignd int if_index;  /* 1, 2, ... */
   char       *if_name;   /* null terminated name:  "le0", ... */
};
```

The memory used for this array of structures, along with the interface names pointed to by the if_name members, is obtained dynamically. This memory is freed by calling the if_freenameindex() function.

Return Value

When successful, if_nameindex() returns a pointer to an array of if_nameindex structures. Upon failure, if_nameindex() returns NULL and sets errno to the following:

Error Code
Description

ENOMEM
Insufficient storage is available to supply the array.

Related Information

- "if_freenameindex() — Free the Memory Allocated by if_nameindex()" on page 404
- "if_indextoname() — Map a Network Interface Index to Its Corresponding Name" on page 404
- "if_nametoindex() — Map a Network Interface Name to Its Corresponding Index" on page 406

if_nametoindex() — Map a Network Interface Name to Its Corresponding Index

Standards

Standards / Extensions	C or C++	Dependencies
RFC2553	both	

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <net/if.h>

unsigned int if_nametoindex(const char *ifname);
```

General Description

The if_nametoindex() function returns the interface index corresponding to the interface name *ifname*.

Return Value

When successful, if_nametoindex() returns the interface index corresponding to the interface name *ifname*. Upon failure, if_nametoindex() returns zero and sets errno to one of the following:

Error Code
Description

EINVAL
Invalid parameter was specified. The *ifname* parameter was NULL.

ENOMEM
Insufficient storage is available to obtain the information for the interface name.

ENXIO
The specified interface name provided in the *ifname* parameter does not exist.

Related Information

- "if_freenameindex() — Free the Memory Allocated by if_nameindex()" on page 404
- "if_indextoname() — Map a Network Interface Index to Its Corresponding Name" on page 404
- "if_nameindex() — Return All Network Interface Names and Indexes" on page 405

ilogb() — Integer Unbiased Exponent

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

Format

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

```
int ilogb(double x);
```

General Description

Returns the unbiased exponent of *x* as an integer.

Note: This function works in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

Returned Value

If successful, `ilogb()` returns the unbiased exponent of *x* as an integer. If *x* is `0.0`, ilogb() returns INT_MIN.

Related Information

- "math.h" on page 32
- "logb() — Calculate Unbiased Exponent" on page 461

index() — Search for Character

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

Finds the first occurrence of *c* (converted to an unsigned character) in *string*. The *c* character can be the null character (`\0`); the ending null is included in the search.

Rule: The *string* argument must contain a null character (`\0`), marking the end of the string.

The index() function is identical to "strchr() — Search for Character" on page 811.

Returned Value

Returns a pointer to the first occurrence of *c* (converted to an unsigned character) in *string*.

If *c* is not found, `index()` returns a null pointer.

No errnos are defined for `index()`.

Related Information

- "strings.h" on page 44
- "memchr() — Search Buffer" on page 492
- "rindex() — Search for a Character" on page 673
- "strchr() — Search for Character" on page 811
- "strrchr() — Find Last Occurrence of Character in String" on page 836
- "strspn() — Search String" on page 837
- "strstr() — Locate Substring" on page 838

inet_addr() — Translate an Internet Address into Network Order

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

Interprets character strings representing host addresses expressed in standard IPv4 dotted-decimal notation and returns host addresses suitable for use as an internet address.

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 four-part address is specified, each part is interpreted as a byte of data and assigned, from left to right, to one of the four 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 virtual machine 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 according to the C language syntax. A leading `0x` or `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

Returns the internet address in network byte order. If the internet address is returned in error—for example, not in the correct format—the function returns INADDR_NONE. INADDR_NONE is defined in the netinet/in.h header file.

Related Information

- "inet_makeaddr() — Create an Internet Host Address" on page 410
- "inet_netof() — Get Network Number from Internet Host Address" on page 410
- "inet_network() — Get Network Number from Decimal Host Address" on page 411
- "inet_ntoa() — Get Decimal Internet Host Address" on page 412

inet_lnaof() — Translate a Local Network Address into Host Byte Order

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

Breaks apart the internet host address and returns the local network address portion.

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

Returns the local network address in host byte order.

Related Information

- "inet_addr() — Translate an Internet Address into Network Order" on page 408
- "inet_makeaddr() — Create an Internet Host Address" on page 410
- "inet_netof() — Get Network Number from Internet Host Address" on page 410
- "inet_network() — Get Network Number from Decimal Host Address" on page 411

• “inet_ntoa() — Get Decimal Internet Host Address” on page 412

inet_makeaddr() — Create an Internet Host Address

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

Takes a network number and a local network address and constructs an internet address.

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

Returns the internet address in network byte order.

Related Information

- “inet_addr() — Translate an Internet Address into Network Order” on page 408
- “inet_lnaof() — Translate a Local Network Address into Host Byte Order” on page 409
- “inet_netof() — Get Network Number from Internet Host Address” on page 410
- “inet_network() — Get Network Number from Decimal Host Address” on page 411
- “inet_ntoa() — Get Decimal Internet Host Address” on page 412

inet_netof() — Get Network Number from Internet Host Address

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

Breaks apart the internet host address and returns the network number portion.

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

Returns the network number in host byte order.

Related Information

- "inet_addr() — Translate an Internet Address into Network Order" on page 408
- "inet_lnaof() — Translate a Local Network Address into Host Byte Order" on page 409
- "inet_makeaddr() — Create an Internet Host Address" on page 410
- "inet_ntoa() — Get Decimal Internet Host Address" on page 412

inet_network() — Get Network Number from Decimal Host Address

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

Interprets character strings representing numbers expressed in standard IPv4 dotted-decimal notation and returns numbers suitable for use as a network number. A whitespace character is considered to terminate the portion of the string to be converted. Any characters after the whitespace character(s) are ignored.

Numbers supplied as address parts in standard dotted-decimal notation can be decimal, hexadecimal, or octal. Numbers are interpreted according to the C language syntax. A leading `0x` or `0X` implies hexadecimal; a leading `0` implies octal. A number without a leading `0` implies decimal.

The difference between inet_network() and inet_addr() is that inet_network() assumes that every dot-delimited numeric string represents a single byte of the resulting address, whereas inet_addr() has a more elaborate algorithm for deciding that numbers should be used for what bytes.

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

Returns the network number in host byte order.

Related Information

- "inet_lnaof() — Translate a Local Network Address into Host Byte Order" on page 409
- "inet_makeaddr() — Create an Internet Host Address" on page 410
- "inet_ntoa() — Get Decimal Internet Host Address" on page 412

inet_ntoa() — Get Decimal Internet Host Address

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

Returns a pointer to a string expressed in the dotted-decimal notation. The inet_ntoa() call accepts an internet address expressed as a 32-bit quantity in network byte order and returns a string expressed in dotted-decimal notation.

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

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 Order" on page 408
- "inet_lnaof() — Translate a Local Network Address into Host Byte Order" on page 409
- "inet_makeaddr() — Create an Internet Host Address" on page 410
- "inet_netof() — Get Network Number from Internet Host Address" on page 410
- "inet_network() — Get Network Number from Decimal Host Address" on page 411

inet_ntop() — Convert Internet Address Format from Binary to Text

Standards

Standards / Extensions	C or C++	Dependencies
RFC2553	both	

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <arpa/inet.h>

const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
```

General Description

The inet_ntop() function converts from an Internet address in binary format (network form), specified by *src*, to standard text format (presentation form), and places the result in *dst*, when *size*, the space available in *dst*, is sufficient. The argument *af* specifies the family of the Internet address. This can be AF_INET or AF_INET6.

The argument *src* points to a buffer holding an IPv4 Internet address if the *af* argument is AF_INET, or an IPv6 Internet address if the *af* argument is AF_INET6.

The argument *dst* points to a buffer where the function will store the resulting text string. The *size* argument specifies the size of this buffer. The application must specify a non-NULL *dst* argument. For IPv6 addresses, the buffer must be at least 46 bytes. For IPv4 addresses, the buffer must be at least 16 bytes.

In order to allow applications to easily declare buffers of the proper size to store IPv4 and IPv6 addresses in string form, the following two constants are defined in <netinet/in.h>:

```
#define INET_ADDRSTRLEN 16
#define INET6_ADDRSTRLEN 46
```

Returned Value

If successful, inet_ntop() returns a pointer to the buffer containing the converted address.

If unsuccessful, inet_ntop() returns NULL and sets errno to one of the following values:

Error Code
Description

EAFNOSUPPORT
The address family specified in *af* is unsupported.

ENOSPC
The destination buffer *size* is too small.

Related Information

- "arpa/inet.h" on page 18
- "netinet/in.h" on page 34
- "sys/socket.h" on page 46
- "inet_addr() — Translate an Internet Address into Network Order" on page 408
- "inet_makeaddr() — Create an Internet Host Address" on page 410
- "inet_netof() — Get Network Number from Internet Host Address" on page 410
- "inet_network() — Get Network Number from Decimal Host Address" on page 411
- "inet_ntoa() — Get Decimal Internet Host Address" on page 412
- "inet_pton() — Convert Internet Address Format from Text to Binary" on page 414

inet_pton() — Convert Internet Address Format from Text to Binary

Standards

Standards / Extensions	C or C++	Dependencies
RFC2553	both	

Format

```
#define _OPEN_SYS_SOCK_IPV6
#include <arpa/inet.h>

int inet_pton(int af, const char *src, void *dst);
```

General Description

The inet_pton() function converts an Internet address from its standard text format (presentation form) into its numeric binary form (network form). The argument *af* specifies the family of the address.

Note: AF_INET and AF_INET6 address families are currently supported.

The argument *src* points to the string being passed in. The argument *dst* points to a buffer into which inet_pton() stores the numeric address. The address is returned in network byte order. The caller must ensure that the buffer pointed to by *dst* is large enough to hold the numeric address.

If the *af* argument is AF_INET, inet_pton() accepts a string in the standard IPv4 dotted-decimal form:

```
ddd.ddd.ddd.ddd
```

where *ddd* is a 1 to 3 digit decimal number between 0 and 255.

If the *af* argument is AF_INET6, the *src* string must be in one of the following standard IPv6 text forms:

1. The preferred form is *x:x:x:x:x:x:x:x*, where the *x*'s are the hexadecimal values of the eight 16-bit hextets of the address. Leading zeros in individual fields can be omitted, but there should be at least one numeral in every field.
2. A string of contiguous zero fields in the preferred form can be shown as *::* The *::* can only appear once in an address. Unspecified addresses *(0:0:0:0:0:0:0:0)* may be represented simply as *::*.
3. A third form that is sometimes more convenient when dealing with a mixed environment of IPv4 and IPv6 is IPv4-mapped or IPv4-compatible addresses in *x:x:x:x:x:x:d.d.d.d* format, where *x*'s are the hexadecimal values of the six high-order 16-bit hextets of the IPv6 part of the address, and the *d*'s are the decimal values of the four low-order 8-bit octets of the IPv4 part of the address (standard IPv4 representation).

Note: A more extensive description of the IPv6 standard representations can be found in RFC2373.

Returned Value

If successful, inet_pton() returns 1 and stores the binary form of the Internet address in the buffer pointed to by *dst*.

If unsuccessful because the input buffer pointed to by *src* is not a valid string, inet_pton() returns 0.

If unsuccessful because the *af* argument is unknown, inet_pton() returns -1 and sets errno to the following value:

Error Code
Description

EAFNOSUPPORT
The address family specified in *af* is unsupported.

Related Information

- “arpa/inet.h” on page 18
- “sys/socket.h” on page 46
- “inet_addr() — Translate an Internet Address into Network Order” on page 408
- “inet_makeaddr() — Create an Internet Host Address” on page 410
- “inet_netof() — Get Network Number from Internet Host Address” on page 410
- “inet_network() — Get Network Number from Decimal Host Address” on page 411
- “inet_ntoa() — Get Decimal Internet Host Address” on page 412
- “inet_ntop() — Convert Internet Address Format from Binary to Text ” on page 413

initstate() — Initialize Generator for random()

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

Initializes a state array, pointed to by *state*, for future use in calls to the random() functions by the calling thread. The initstate() function uses the *size* argument, which specifies the size in bytes of the state array, to determine 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.

Returned Value

If successful, initstate() returns a pointer to the previous state array; otherwise, it returns a null pointer.

If initstate() is called with a *size* less than 8, it fails and writes a message to standard error.

Related Information

- "stdlib.h" on page 42
- "random() — Generate a Better Random Number" on page 627
- "setstate() — Change Generator for random()" on page 739
- "srandom() — Use Seed to Initialize Generator for random()" on page 801
- "drand48() — Generate Pseudo-Random Number" on page 178

insque() — Insert an Element into a Doubly Linked List

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

Inserts the element, pointed to by *element*, into a doubly linked list immediately after the element pointed to by *pred*. The insque() function operates on pointers to structures that 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 backward pointers to point to the element.

Returned Value

The insque() function has no returned value.

Related Information

- "search.h" on page 38
- "remque() — Remove an Element from a Doubly Linked List" on page 660

ioctl() — Control Device

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

Format

Terminals

```
#include <sys/ioctl.h>
int ioctl(int fildes int cmd, ... /* arg */);
```

Sockets

```
#define _XOPEN_SOURCE_EXTENDED 1

/**     OR      **/

#define _OE_SOCKETS

#include <sys/ioctl.h>
#include <net/rtrouteh.h>
#include <net/if.h>
int ioctl(int fildes, int cmd, ... /* arg */);
```

General Description

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 *fildes*.

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.

Sockets

The following ioctl() commands are used with AF_INET or AF_INET6 sockets:

FIONBIO
Sets or clears nonblocking I/O for a socket. *arg* 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. *arg* is a pointer to an integer. Sets the value of the integer to the number of immediately readable characters for the socket.

SIOCATMARK
Queries whether the current location in the data input is pointing to out-of-band data. *arg* 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`. For more information on receiving out-of-band data, see “recv() — Receive Data on a Socket” on page 640, “recvfrom() — Receive Messages on a Socket” on page 642, and “recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers” on page 645.

SIOCDINTERFACE
Removes an interface. *arg* is pointer to an `ifreq` structure, as defined in net/if.h. If the interface is inactive, the interface configuration information is removed from the TCP/IP server.

SIOCGIFADDR
Gets the network interface address. *arg* is a pointer to an `ifreq` structure, as defined in net/if.h. The interface address is returned in the argument.

SIOCGIFBRDADDR
Gets the network interface broadcast address. *arg* is a pointer to an `ifreq` structure, as defined in net/if.h. The interface broadcast address is returned in the argument.

SIOCGIFCONF
Gets the network interface configuration. *arg* is a pointer to an `ifconf` structure, as defined in net/if.h. The interface configuration is returned in the buffer pointed to by the `ifconf` structure. The returned data's length is returned in the field that had originally contained the length of the buffer.

SIOCGIFDSTADDR
Gets the network interface destination address. *arg* is a pointer to an `ifreq` structure, as defined in net/if.h. The interface destination (point-to-point) address is returned in the argument.

SIOCGIFFLAGS
Gets the network interface flags. *arg* is a pointer to an `ifreq` structure, as defined in net/if.h. The interface flags are returned in the argument.

SIOCGIFMETRIC
Gets the network interface routing metric. *arg* is a pointer to an `ifreq` structure, as defined in net/if.h. The interface routing metric is returned in the argument.

SIOCGIFNETMASK
Gets the network interface network mask. *arg* is a pointer to an `ifreq` structure, as defined in net/if.h. The interface network mask is returned in the argument.

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.

Special Behavior for C++

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

Sockets Returned Value

If successful, ioctl() returns a value of `0`.

If unsuccessful, ioctl() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EAFNOSUPPORT
The address family is not supported.

EBADF
The *fildes* parameter is not a valid socket descriptor.

EBUSY
The specified interface is still active.

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.

ECMSPARM
Incorrect parameters were passed to the service.

ENOTSOCK
The descriptor is for a file, not for a socket.

ENXIO
The specified interface does not exist.

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);
⋮
```

Related Information

- "close() — Close a File" on page 127
- "fcntl() — Control Open File Descriptors" on page 207
- "open() — Open a File" on page 527
- "pipe() — Create an Unnamed Pipe" on page 539
- "read() — Read From a File " on page 628
- "sigaction() — Examine or Change a Signal Action" on page 749
- "write() — Write Data on a File or Socket" on page 975

isalnum() to isxdigit() — Test Integer Value

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.

Recommendation: The functions in ctype.h are also available as macros, which you should use instead of the functional forms for better performance,

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')`

A description of each function in this group follows.

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" topic in *z/OS: XL C/C++ Programming Guide*.

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 22
- “isblank() — Test for Blank Character Classification” on page 427
- “iswalnum() to iswxdigit() — Test Wide Integer Value” on page 432
- “setlocale() — Set Locale” on page 721
- “tolower(), toupper() — Convert Character Case” on page 895

isascii() — Test for 7-bit US-ASCII Character

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

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

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

Returns a nonzero value if *c* is a 7-bit US-ASCII character code between `0` and hexadecimal `007F` inclusive; otherwise, it returns a value of `0`.

Special Behavior for _ALL_SOURCE

Returns a nonzero value 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 a value of `0`.

Table 19 on page 422 lists 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 19. 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

Table 19. Characters for which isascii() returns nonzero (continued)

Character (Symbolic Name)	IBM-1047 Encoding (Hex)	ISO 8859-1 Encoding (Hex)
<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
<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

Table 19. Characters for which isascii() returns nonzero (continued)

Character (Symbolic Name)	IBM-1047 Encoding (Hex)	ISO 8859-1 Encoding (Hex)
<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
<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

Table 19. Characters for which isascii() returns nonzero (continued)

Character (Symbolic Name)	IBM-1047 Encoding (Hex)	ISO 8859-1 Encoding (Hex)
<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
<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

Table 19. Characters for which isascii() returns nonzero (continued)

Character (Symbolic Name)	IBM-1047 Encoding (Hex)	ISO 8859-1 Encoding (Hex)
<DEL>	07	7F

Related Information

- "toascii() — Translate Integer to a 7-Bit ASCII Character" on page 890

isatty() — Test if Descriptor Represents a Terminal

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

Format

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

int isatty(int fildes);
```

General Description

Determines if a file descriptor, *fildes*, is associated with a terminal.

The isatty() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

Returns a value of 1 if the given file descriptor is a terminal; otherwise, isatty() returns a value of `0`.

No errnos are defined for isatty().

Special Behavior for XPG4

isatty() returns 1 if the given file descriptor is a terminal, or 0 otherwise and sets errno to one of the following values:

Error Code
Description

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 _OPEN_SYS
#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 51
- “ttyname() — Get the Name of a Terminal” on page 897

isblank() — Test for Blank Character Classification

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 nonstandard 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 (for example, 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 a nonzero value if the current LC_CTYPE locale category assigns *c* the blank character attribute; otherwise, isblank() returns a value of 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 22
- “iswalnum() to iswxdigit() — Test Wide Integer Value” on page 432
- “iswblank() — Test for Blank Character Classification” on page 433
- “setlocale() — Set Locale” on page 721

iscics() — Verify Whether CICS is Running

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

Returns a nonzero integer if your program is currently running under CICS; otherwise, iscics() returns a value of 0.

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.
*/
#pragma linkage(ABCPGM, OS)
void ABCPGM(char *);

#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 19

iscntrl() — Test for Control Classification

The information for this function is included in "isalnum() to isxdigit() — Test Integer Value" on page 419.

isdigit() — Test for Hexadecimal-Digit Classification

The information for this function is included in "isalnum() to isxdigit() — Test Integer Value" on page 419.

isgraph() — Test for Graphic Classification

The information for this function is included in "isalnum() to isxdigit() — Test Integer Value" on page 419.

islower() — Test for Lowercase

The information for this function is included in "isalnum() to isxdigit() — Test Integer Value" on page 419.

ismccollel() — Identify a Multicharacter Collating Element

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <collate.h>

int ismccollel(collel_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 ismccollel() function returns:

1
if collel_t represents a multicharacter collating element

0
if collel_t represents a single-character collating element

-1
if collel_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 ismccollel() function to determine if the collating
   element is a multicharacter collating element.
 */
#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 20
- “cclass() — Return Characters in a Character Class” on page 104
- “collequiv() — Return List of Equivalent Collating Elements” on page 132
- “collorder() — Return List of Collating Elements” on page 133
- “collrange() — Calculate the Range List of Collating Elements” on page 134
- “colltostr() — Return a String for a Collating Element” on page 136
- “getmccoll() — Get Next Collating Element from String” on page 345
- “getwmccoll() — Get Next Collating Element from Wide String” on page 386
- “maxcoll() — Return Maximum Collating Element” on page 479
- “strtocoll() — Return Collating Element for String” on page 839

isnan() — Test for NaN

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

Format

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

int isnan( double x);
```

General Description

Tests whether *x* is NaN (not a number).

Recommendation: For better performance, you should use the macro form of isnan() instead of the functional form. To use the functional form, however, 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. For hexadecimal floating-point isnan() always returns 0, but for IEEE floating-point, this function returns nonzero if x is a NaN. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

Returned Value

Always returns a value of `0`.

No errnos are defined for isnan().

Related Information

- "math.h" on page 32

isprint() — Test for Printable Character Classification

The information for this function is included in "isalnum() to isxdigit() — Test Integer Value" on page 419.

ispunct() — Test for Punctuation Classification

The information for this function is included in "isalnum() to isxdigit() — Test Integer Value" on page 419.

isspace() — Test for Space Character Classification

The information for this function is included in "isalnum() to isxdigit() — Test Integer Value" on page 419.

isupper() — Test for Uppercase Letter Classification

The information for this function is included in "isalnum() to isxdigit() — Test Integer Value" on page 419.

iswalnum() to iswxdigit() — Test Wide Integer Value

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 *z/OS: XL C/C++ Programming Guide*.

A description of each function in this group follows.

iswalnum()
Test for a wide alphanumeric character, as defined in the `alnum` locale source file and in the `alnum` class of the LC_CTYPE category of the current locale.

iswalpha()
Test for a wide alphabetic character, as defined in the `alpha` locale source file and in the `alpha` class of the LC_CTYPE category of the current locale.

iswcntrl()
Test for a wide control character, as defined in the `cntrl` locale source file and in the `cntrl` class of the LC_CTYPE category of the current locale.

iswdigit()
Test for a wide decimal-digit character: 0 through 9, as defined in the `digit` locale source file and in the `digit` class of the LC_CTYPE category of the current locale.

iswgraph()
Test for a wide printing character, not a space. as defined in the `graph` locale source file and in the `graph` class of the LC_CTYPE category of the current locale.

iswlower()
Test for a wide lowercase letter, as defined in the `lower` locale source file and in the `lower` class of the LC_CTYPE category of the current locale.

iswprint()
Test for any wide printing character, as defined in the `print` locale source file and in the `print` class of the LC_CTYPE category of the current locale.

iswpunct()
Test for a wide nonalphanumeric, nonspace character, as defined in the `punct` locale source file and in the `punct` class of the LC_CTYPE category of the current locale.

iswspace()
Test for a wide 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.

iswupper()
Test for a wide uppercase letter, as defined in the `upper` locale source file and in the `upper` class of the LC_CTYPE 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 `xdigit` locale source file and in the `xdigit` class of the LC_CTYPE category of the current locale.

The behavior of these wide-character functions 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 54

iswblank() — Test for Blank Character Classification

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 nonstandard 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 (for example, 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 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 54
- "isblank() — Test for Blank Character Classification" on page 427
- "iswalnum() to iswxdigit() — Test Wide Integer Value" on page 432

iswcntrl() — Test for Control Classification

The information for this function is included in "iswalnum() to iswxdigit() — Test Wide Integer Value" on page 432.

iswctype() — Test for Character Property

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 not valid (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`, `egraph`, `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

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 54
- “wctype() — Obtain Handle for Character Property Classification” on page 968

iswdigit() — Test for Hexadecimal-Digit Classification

The information for this function is included in “iswalnum() to iswxdigit() — Test Wide Integer Value” on page 432.

iswgraph() — Test for Graphic Classification

The information for this function is included in “iswalnum() to iswxdigit() — Test Wide Integer Value” on page 432.

iswlower() — Test for Lowercase

The information for this function is included in "iswalnum() to iswxdigit() — Test Wide Integer Value" on page 432.

iswprint() — Test for Printable Character Classification

The information for this function is included in "iswalnum() to iswxdigit() — Test Wide Integer Value" on page 432.

iswpunct() — Test for Punctuation Classification

The information for this function is included in "iswalnum() to iswxdigit() — Test Wide Integer Value" on page 432.

iswspace() — Test for Space Character Classification

The information for this function is included in "iswalnum() to iswxdigit() — Test Wide Integer Value" on page 432.

iswupper() — Test for Uppercase Letter Classification

The information for this function is included in "iswalnum() to iswxdigit() — Test Wide Integer Value" on page 432.

iswxdigit() — Test for Hexadecimal-Digit Classification

The information for this function is included in "iswalnum() to iswxdigit() — Test Wide Integer Value" on page 432.

isxdigit() — Test for Hexadecimal-Digit Classification

The information for this function is included in "isalnum() to isxdigit() — Test Integer Value" on page 419.

jrand48() — Generate Pseudorandom Number

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 drand48() and erand48() functions return non-negative, double-precision, floating-point values, uniformly distributed over the interval `[0.0,1.0]`.
- The lrand48() and nrand48() functions return non-negative, long integers, uniformly distributed over the interval `[0,2**31]`.

- The mrand48() and jrand48() functions 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)mod(2**48)     n>=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)
```

You may change the values of a and c by calling the lcong48() function. Calling either seed48() or srand48() restores a and c.

Special Behavior for OpenExtensions Services

You can make jrand48() 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 VM/ESA serializes access to the storage for `X(n)`, a, and c by functions in the `drand48` family when a multithreaded application calls them.

If you request thread-specific behavior and call erand48() from thread `t`, erand48() generates the next 48-bit integer value in a sequence of 48-bit integer values, `X(t,i)`, for the thread `t`, according to the linear congruential formula:

```
X(t,n+1) = (a(t)X(t,n) + c(t))mod(2**48)     n>=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 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)
```

You may change the values of `a(t)` and `c(t)` by calling the lcong48() function from the thread `t`. Calling either seed48() or srand48() from the thread restores the initial values of `a(t)` and `c(t)`.

Returned Value

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 OpenExtensions Services

If you request thread-specific behavior for the `drand48` family and call jrand48() on thread `t`, jrand48() 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 42
- "drand48() — Generate Pseudo-Random Number" on page 178

- “erand48() — Generate Pseudo-Random Number” on page 187
- “lcong48() — Initialize Pseudorandom Number” on page 444
- “lrand48() — Generate Pseudorandom Number” on page 467
- “mrand48() — Generate Pseudorandom Number” on page 510
- “nrand48() — Pseudo-Random Number Generator” on page 523
- “seed48() — Initialize Pseudorandom Number” on page 679
- “srand48() — Initialize Pseudorandom Number” on page 801

j0(), j1(), jn() — Bessel Functions of the First Kind

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

LANGLVL(SAA), LANGLVL(SAAL2), or LANGLVL(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 *x* argument must be positive, and the *n* argument should be greater than or equal to 0. If *n* is less than 0, the result will contain a negative exponent.

Note: This function works in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE floating-point” on page 63 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. Otherwise, the function returns the calculated value.

Special Behavior for IEEE

If *x* is negative, y0(), y1(), and yn() return the value NaNQ. If *x* is 0, y0(), y1(), and yn() return the value -HUGE_VAL. In all cases, errno remains unchanged.

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 32
- "erf() - erfc() — Calculate Error and Complementary Error Functions" on page 189
- "gamma() — Calculate Gamma Function" on page 310
- "y0(), y1(), yn() — Bessel Functions of the Second Kind" on page 983

kill() — Send a Signal to a Process

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	POSIX(ON)

Format

```
#define _OPEN_SYS
#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 no other unblocked signals are pending, 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 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 24 on page 750.) 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 *pid* is valid.

The kill() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

Returns a value of 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 a value of -1 and sets errno to one of the following values under the conditions described.

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
No processes or process groups correspond to *pid*.

Example

CBC3BK01

```
/* CBC3BK01 */
#define _OPEN_SYS
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
#include <spawn.h>
#include <unistd.h>
#include <wait.h>

main() {
  sigset_t sigset;
  int    fd_count;
  int    fd_map[5];
  int    i;
  int    p[2], status;
  char   c='z';
  pid_t  pid;

  if (pipe(p) != 0)
    perror("pipe() error");
  else {
    for (i = 0; i < 3; i++)
      fd_map[i] = i;
    fd_map[3] = p[0];
    fd_map[4] = p[1];
    fd_count = 5;
    if ((pid = spawn("CBC3BK02", fd_count,fd_map, 0, 0, 0)) == -1)
      perror("spawn() error");
    else {
      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]);
}

/* CBC3BK02 */
#define _OPEN_SYS
#include <signal.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

main() {
  sigset_t sigset;
  char   c='z';

  sigemptyset(&sigset);
  puts("child is letting parent know he's ready for signal");
  write(4, &c, 1);
  puts("child is waiting for signal");
  sigsuspend(&sigset);
  exit(0);

}
```

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 38
- “unistd.h” on page 51
- “getpid() — Get the Process ID” on page 355
- “pthread_kill() — Send a Signal to a Thread” on page 586
- “raise() — Raise Signal” on page 623
- “setsid() — Create Session, Set Process Group ID” on page 733
- “sigaction() — Examine or Change a Signal Action” on page 749
- “signal() — Handle Interrupts” on page 767
- “sigprocmask() — Examine or Change a Thread” on page 773

killpg() — Send a Signal to a Process Group

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON)

Format

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

int killpg(pid_t pgrp, int sig);
```

General Description

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 the 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 1, 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 1, killpg() returns a value of -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 24 on page 750.) This must be 0 or one of the signals defined in the <signal.h> include file. If *sig* is 0, killpg() performs error-checking but does not send a signal. You can code *sig* as 0 to check whether the *pid* argument is valid.

Note: This function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, killpg() returns a value of 0 if it has permission to send *sig* to any of the processes in the process group ID specified by *pgrp*.

If unsuccessful, killpg() returns a value of -1 and sets errno to one of the following values under the conditions described:

EINVAL
The value of *sig* is incorrect or is not the number of a supported signal, or the value of *pgrp* is less than or equal to 1.

EPERM
The caller does not have permission to send the signal to any process in the process group ID specified by *pgrp*.

ESRCH
No process groups correspond to *pgrp*.

Related Information

- "signal.h" on page 38
- "bsd_signal() — Establish BSD Version of signal()" on page 99
- "getpid() — Get the Process ID" on page 355
- "kill() — Send a Signal to a Process" on page 439
- "raise() — Raise Signal" on page 623
- "setsid() — Create Session, Set Process Group ID" on page 733

- “sigaction() — Examine or Change a Signal Action” on page 749
- “sighold() — Add a Signal to a Thread” on page 761
- “sigignore() — Set Disposition to Ignore a Signal” on page 762
- “signal() — Handle Interrupts” on page 767
- “sigprocmask() — Examine or Change a Thread” on page 773
- “sigrelse() — Remove a Signal from a Thread” on page 776
- “sigset() — Change a Signal Action and/or a Thread” on page 776

labs() — Calculate Long Absolute Value

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 *n*. The result is undefined when *n* 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 *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 42
- "abs(), absf(), absl() — Calculate Integer Absolute Value" on page 68
- "fabs(), fabsf(), fabsl() — Calculate Floating-Point Absolute Value" on page 200

lcong48() — Initialize Pseudorandom Number

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 lcong48(), seed48(), and srand48() functions are initialization functions. You should invoke one of these before calling either the drand48(), lrand48, or mrand48() function.

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 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)mod(2**48)     n>=0
```

The initial values of X, a, and c are:

```
    X(0)= 1
    a   = 5deece66d (base 16)
    c   = b         (base 16)
```

IBM XL C/C++ for z/VM provides storage to save the most recent 48-bit integer value of the sequence, `X(i)`. The drand48(), lrand48(), and mrand48() functions share this storage. You may reinitialize the most recent 48-bit value in this storage by calling lcong48(). 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].

You may also change the values of a and c, by calling lcong48(). 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 OpenExtensions Services

You can make lcong48() 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, IBM XL C/C++ for z/VM serializes access to the storage for `X(n)`, a, and c by functions in the `drand48` family when a multithreaded application calls them.

If you request thread-specific behavior, making calls to drand48(), lrand48(), and mrand48() from thread `t` generates 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))mod(2**48)     n>=0
```

IBM XL C/C++ for z/VM provides thread-specific storage to save the most recent 48-bit integer value of the sequence, `X(t,i)`. When you call lcong48() from thread `t`, lcong48() 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].

You may also change the values of a`(t)` and c`(t)`, for the thread `t` by calling lcong48(). 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 lcong48() has used values from the argument array, *param*[7], to change the values of a and c and to reinitialize storage for the most recent 48-bit integer value in the sequence, `X(t,i)`, it returns.

Special Behavior for OpenExtensions Services

If you request thread-specific behavior for the `drand48` family and call `lcong48` on thread `t`, `lcong48` uses the argument array, *param*[7], to change the values of a`(t)` and c`(t)` and to reinitialize the 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 42
- “drand48() — Generate Pseudo-Random Number” on page 178
- “erand48() — Generate Pseudo-Random Number” on page 187
- “jrand48() — Generate Pseudorandom Number” on page 436
- “lrand48() — Generate Pseudorandom Number” on page 467
- “mrand48() — Generate Pseudorandom Number” on page 510
- “nrand48() — Pseudo-Random Number Generator” on page 523
- “seed48() — Initialize Pseudorandom Number” on page 679
- “srand48() — Initialize Pseudorandom Number” on page 801

ldexp(), ldexpf(), ldexpl() — Multiply by a Power of Two

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

Format

```
#include <math.h>

double ldexp(double x, int exp);
float ldexp(float x, int exp);                /* C++ only */
long double ldexp(long double x, int exp);    /* C++ only */
float ldexpf(float x, int exp);
long double ldexpl(long double x, int exp);
```

General Description

Calculates the value of $x*(2^{exp})$.

Returned Value

Returns the calculated value. Otherwise, if the correct calculated value is outside the range of representable values, ldexp() returns plus or minus HUGE_VAL, 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. The ldexp() function returns a value of `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 32
- “frexp() — Extract Mantissa and Exponent of the Floating-Point Value” on page 282
- “modf() — Extract Fractional and Integral Parts of a FLoating-Point Value” on page 507

ldiv() — Compute Quotient and Remainder of Integral Division

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 42
- “div() — Calculate Quotient and Remainder” on page 168

lfind() — Linear Search Routine

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

A linear search routine, which is the same as lsearch(), with one exception: if the entry is not found, it is not added to the table; instead, lfind() returns a null pointer.

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 lfind() finds the entry, lfind() returns a pointer to it; otherwise, lfind() returns a null pointer.

No errnos are defined for lfind().

Related Information

- “search.h” on page 38
- “bsearch() — Search Arrays” on page 100
- “hsearch() — Search Hash Tables” on page 394
- “lsearch() — Linear Search and Update” on page 468
- “tsearch() — Search Binary Tree” on page 896

lgamma() — Log Gamma Function

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

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 *x* argument 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. The math.h header file redefines the string "*signgam*" to an invocation of __signgam(). The actual *signgam* external variable is used to store the *signgam* value for the IPT.

See "__signgam() — Return __signgam Reference" on page 771 and "math.h" on page 32.

Returned Value

If successful, lgamma() returns the above function of its argument.

The lgamma() function fails under the following conditions:

- If the result overflows, lgamma() returns HUGE_VAL. and sets 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 32
- "exp(), expf(), expl()— Calculate Exponential Function" on page 199
- "isnan() — Test for NaN" on page 431
- "__signgam() — Return __signgam Reference" on page 771

__librel() — Query Release Level

Standards / Extensions	C or C++	Dependencies
Language Environment	both	

Format

```
#include <stdlib.h>

int __librel(void);
```

General Description

Provides the release level of the IBM XL C/C++ for z/VM library.

Returned Value

Returns the IBM XL C/C++ for z/VM specific library release level of your IBM XL C/C++ for z/VM program, printed in a hexadecimal format. The first byte of the returned value contains the product and version, second byte the release, and third and fourth bytes the modification level. For C programs running with the Language Environment, the product designation is 1. For C programs running under the C/370 Specific Library

The following diagram shows the format of the 32-bit int returned by the C/370 V2.2 version of __librel().

```
The C/370 V2.2 version of __librel() returns 0x02020000
                                               ||||||
                                               ||||--- Mod level
                                               ||----- Release
                                               ------- Version
```

Example

CBC3BL04

```
/* CBC3BL04
   This example demonstrates what is returned when the function __librel()
   is called when using the Version 1 Release 8 Modification Level 0, of
   Language Environment for VM.
 */
#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: 11080000
```

Related Information

- “stdlib.h” on page 42

link() — Create a Link to a File

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

Format

```
#define _OPEN_SYS
#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. The link() function 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 exist 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.

Returned Value

If successful, link() returns a value of `0`.

If unsuccessful, link() returns a value of `-1` and sets errno to one of the following values under the conditions described:

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 detected during resolution of *oldfile* or *newname* is greater than POSIX_SYMLOOP.

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 path name 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 _OPEN_SYS
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _OPEN_SYS
#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 51
- “rename() — Rename File” on page 661
- “symlink() — Create a Symbolic Link to a Path Name” on page 854
- “unlink() — Remove a Directory Entry” on page 910

listen() — Prepare Server for Incoming Client Requests

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

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

Transforms an active stream socket into a passive socket and applies only to stream sockets.

socket
The socket descriptor.

backlog
Defines the maximum length for the queue of pending connections for AF_INET or AF_INET6 sockets.

For AF_UNIX and AF_IUCV sockets, *backlog* is ignored.

The listen() call indicates a readiness to accept client connection requests and creates a connection request queue of length *backlog* to queue incoming connection requests. Once the queue is full, additional connection requests are rejected.

Once listen() is 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(). If bind() has not been called for socket, listen() completes the necessary binding. 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_INET and AF_INET6 sockets, the value cannot exceed the maximum number of connections allowed by the installed TCP/IP, which is set by the SOMAXCONN statement in the TCP/IP profile. (For more information on the SOMAXCONN statement, see *z/VM: TCP/IP Planning and Customization*.) For AF_UNIX and AF_IUCV sockets, this parameter is ignored.

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, listen() returns a value of `0`.

If unsuccessful, listen() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EBADF
The *socket* parameter is not a valid socket descriptor.

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 69
- “bind() — Bind a Name to a Socket” on page 92
- “connect() — Connect a Socket” on page 138
- “socket() — Create a Socket” on page 791

localdtconv() — Date/Time Formatting Convention Inquiry

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 nonstandard 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 (that is, 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

- For information about locales and character sets, see *z/OS: XL C/C++ Programming Guide*.
- “locale.h” on page 30
- “localeconv() — Query Numeric Conventions” on page 454
- “localtime() — Convert Time and Correct for Local Time” on page 456
- “setlocale() — Set Locale” on page 721

localeconv() — Query Numeric Conventions

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 5 on page 30.

The localdtconv() function 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             */
};
```

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.

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

- For information about locales and character sets, see *z/OS: XL C/C++ Programming Guide*.
- “locale.h” on page 30
- “localdtconv() — Date/Time Formatting Convention Inquiry” on page 453
- “localtime() — Convert Time and Correct for Local Time” on page 456
- “setlocale() — Set Locale” on page 721

localtime() — Convert Time and Correct for Local Time

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, *timeval*, to a broken-down time expressed in local time. Calendar time is usually obtained by calling time() For a description of the fields of the *tm* structure to which localtime() returns a pointer, see "gmtime() — Convert Time to Broken-Down UTC Time" on page 391.

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 localtime() cannot convert the calendar time, it returns a null pointer.

Notes:

- The ctime(), localtime(), and mktime() functions now return coordinated 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 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, 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. For more information about using POSIX support, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.
  – For all other C applications, time zone information is provided by the current LC_TOD locale category.

For a description of LC_TOD, which is a nonstandard, IBM XL C/C++ for z/VM proprietary locale category, see the "Internationalization: Locales and Character Sets" topic in *z/OS: XL C/C++ Programming Guide*.

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

- For information about locales and character sets, see *z/OS: XL C/C++ Programming Guide*.
- "locale.h" on page 30
- "time.h" on page 49
- "asctime() — Convert Time to Character String" on page 78
- "ctime() — Convert Time to a Character String" on page 153
- "gmtime() — Convert Time to Broken-Down UTC Time" on page 391
- "localdtconv() — Date/Time Formatting Convention Inquiry" on page 453
- "mktime() — Convert Local Time" on page 505
- "setlocale() — Set Locale" on page 721
- "time() — Determine Current Time" on page 885
- "tzset() — Set the Time Zone" on page 900

lockf() — Record Locking on Files

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

Format

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

int lockf(int filedes, int function, off_t size);
```

General Description

Allows sections of a file to be locked with advisory-mode locks. Calls to lockf() from other processes that 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. You must have opened the file descriptor 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
     -------------------------------------------------------
     F_ULOCK    unlock locked sections
     F_LOCK     lock a section for exclusive use
     F_TLOCK    test and lock a section for exclusive use
     F_TEST     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 (from 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 the 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.

Large Files for HFS

Applications that are compiled with the option LANGLVL(EXTENDED) and also define the Feature Test Macro (FTM) _LARGE_FILES before any headers are included will enable this function to operate on HFS files that are larger than 2 gig-1 in size. File size and offset fields will be enlarged to 63 bits in width so any other function operating on this file will have to be enabled with the same FTM.

Returned Value

If successful, lockf() returns a value of 0.

If unsuccessful, lockf() returns a value of -1 and sets errno to one of the following values under the conditions described. Existing locks are not changed.

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.

EOVERFLOW
The offset of the first, or if size is not 0 then the last, byte in the requested section cannot be represented correctly in an object of type off_t.

Related Information

- "unistd.h" on page 51

log(), logf(), logl() — Calculate Natural Logarithm

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++	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: These functions work in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

Returned Value

Returns the computed value. If *x* is negative, log() sets errno to EDOM and returns -HUGE_VAL. If *x* is 0, log() returns -HUGE_VAL and may set errno to ERANGE. If the correct value would cause an underflow, log() returns 0 and stores the value of the macro ERANGE in errno.

Special Behavior for IEEE

If *x* greater than 0, the function returns the natural logarithm (base e) of *x*.

If *x* is negative, the function sets errno to EDOM and returns NaNQ. If *x* is 0.0, the function returns -HUGE_VAL and errno remains unchanged.

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 32
- "exp(), expf(), expl()— Calculate Exponential Function" on page 199
- "log10(), log10f(), log10l() — Calculate Base 10 Logarithm" on page 462
- "pow() — Raise to Power" on page 543

logb() — Calculate Unbiased Exponent

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

Format

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

double logb(double x);
```

General Description

Returns the radix-16 exponent of *x* which is the integral part of

```
log base r (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 63 for more information about IEEE floating-point. For hexadecimal floating-point, the radix-16 exponent is returned. And for IEEE floating-point, the radix-2 exponent is returned.

Returned Value

If successful, logb() returns the radix-16 exponent of *x*.

If *x* is equal to `0.0`, logb() will return -HUGE_VAL and set errno to EDOM.

Related Information

- "math.h" on page 32
- "ilogb() — Integer Unbiased Exponent" on page 406

log1p() — Calculate Natural Log of x plus 1

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

Format

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

double log1p(double x);
```

General Description

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 63 for more information about IEEE floating-point.

Returned Value

If successful, log1p() returns the value of the above function of *x*.

The 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 32
- "log(), logf(), logl() — Calculate Natural Logarithm" on page 460

log10(), log10f(), log10l() — Calculate Base 10 Logarithm

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++	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 63 for more information about IEEE floating-point.

Returned Value

Returns the computed value. If *x* is negative, log10() returns -HUGE_VAL. and sets errno to EDOM. If *x* is `0`, log10() returns -HUGE_VAL, and sets errno to ERANGE. If the correct value would cause an underflow, log10() returns a value of `0` and stores the value of the macro ERANGE in errno.

Special Behavior for IEEE

If successful, the function returns the base 10 logarithm of the positive value of *x*.

If *x* is negative, the function sets errno to EDOM and returns NaNQ. If *x* is 0, the function returns -HUGE_VAL and errno remains unchanged.

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 32
- "exp(), expf(), expl()— Calculate Exponential Function" on page 199
- "log(), logf(), logl() — Calculate Natural Logarithm" on page 460
- "pow() — Raise to Power" on page 543

_longjmp() — Nonlocal Goto

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

Restores a stack environment previously saved in *env* by _setjmp(). The _setjmp() and _longjmp() functions provide a way to perform a nonlocal *goto* and are often used in signal handlers.

The *env* argument is an address for a jmp_buf structure. The *value* argument is the return value from _setjmp().

A call to _setjmp() saves the current stack environment 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, as are nonvolatile *auto* variables that change between calls to _setjmp() and _longjmp().

The X/Open standard states that _longjmp() and _setjmp() are functionally identical to longjmp() and setjmp(), respectively, with the additional restriction that _longjmp() and _setjmp() do not manipulate the signal mask. Since on this implementation longjmp() and setjmp() do not manipulate the signal mask, 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, you should 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().

Recommendation: Although on this implementation the _setjmp()—_longjmp() pair are functionally identical to the setjmp()—longjmp() pair and it is possible to intermix them, you should not do so.

Note: You should 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 make it `0` _longjmp() substitutes a value of 1 in its place.

Special Behavior for C++

If _setjmp() and _longjmp() are used to transfer control in a z/VM C++ program, the behavior in terms of the destruction of automatic objects is undefined. Additionally, if any automatic objects would be destroyed by a thrown exception transferring control to another (destination) point in the program, then a call to _longjmp() at the throw point that transfers control to the same (destination) point has undefined behavior. This applies both to z/VM C++ and C/VM ILC modules. The use of _setjmp() and _longjmp() in conjunction with try(), catch(), and throw() is also undefined.

Special Behavior for XPLINK-compiled C++

Restrictions concerning setjmp.h and ucontext.h:

Note:

1. All XPLINK programs compiled with the Release 10 or later C compilers that are to run with Language Environment Release 10 or later libraries and use the **jmp_buf**, **sigjmp_buf** or **ucontext_t** types must not be compiled with C headers from LE 2.9 or earlier.
2. Non-XPLINK functions compiled with any level of LE headers must not define **jmp_buf**, **sigjmp_buf** or **ucontext_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), _longjmp(), setjmp(), _setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.
3. When __XPLINK__ is defined, the Release 10 and later headers define a larger **jmp_buf**, **sigjmp_buf** or **ucontext_t** area that is required by setjmp(), getcontext(), and related functions when they are called from an XPLINK routine. If __XPLINK__ is not defined, the Release 10 and later headers define a shorter **jmp_buf**, **sigjmp_buf** or **ucontext_t** area. The LE headers prior to Release 10 also define the shorter version of these data areas. If an XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp_buf**, **sigjmp_buf** or **ucontext_t** area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.

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 38
- “getcontext() — Get User Context” on page 318
- “longjmp() — Restore Stack Environment” on page 465
- “setcontext() — Restore User Context” on page 704
- “_setjmp() — Set Jump Point for a Nonlocal Goto” on page 717
- “setjmp() — Preserve Stack Environment” on page 718
- “siglongjmp() — Restore the Stack Environment and Signal Mask” on page 764

- "sigsetjmp() — Save Stack Environment and Signal Mask" on page 778
- "swapcontext() — Save and Restore User Context" on page 849

longjmp() — Restore Stack Environment

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` and are often used in signal handlers.

A call to setjmp() saves the current stack environment 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 as are nonvolatile *auto* variables that change between calls to setjmp() and longjmp().

Note: You should 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.

Rule: The *value* argument passed to longjmp() must be nonzero; if you make it `0`, longjmp() substitutes a value of 1 in its place.

Special Behavior for C

In a POSIX program, the signal mask is *not* saved. To save and restore a stack environment that includes the current signal mask, you should use sigsetjmp() and siglongjmp() instead of setjmp() and longjmp(). You cannot intermix the sigsetjmp()—siglongjmp() pair and the setjmp()—longjmp() pair. A stack environment saved by setjmp() can be restored only by longjmp(). For more information about using POSIX support, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Special Behavior for C++

If setjmp() and longjmp() are used to transfer control in a z/VM C++ program, the behavior in terms of the destruction of automatic objects is undefined. Additionally, if any automatic objects would be destroyed by a thrown exception transferring control to another (destination) point in the program, then a call to longjmp() at the throw point that transfers control to the same (destination) point has undefined behavior. This applies to both z/VM C++ and C/VM 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. Although on this implementation, _setjmp()—_longjmp() are functionally identical to setjmp()—longjmp() and it is possible to intermix them, you should not do so.

Special Behavior for XPLINK-compiled C++

Restrictions concerning setjmp.h and ucontext.h:

Note:

1. All XPLINK programs compiled with the Release 10 or later C compilers that are to run with Language Environment Release 10 or later libraries and use the **jmp_buf**, **sigjmp_buf** or **ucontext_t** types must not be compiled with C headers from LE 2.9 or earlier.
2. Non-XPLINK functions compiled with any level of LE headers must not define**jmp_buf**, **sigjmp_buf** or **ucontext_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), _longjmp(), setjmp(), _setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.
3. When __XPLINK__ is defined, the Release 10 and later headers define a larger **jmp_buf**, **sigjmp_buf** or **ucontext_t**area that is required by setjmp(), getcontext(), and related functions when they are called from an XPLINK routine. If __XPLINK__ is not defined, the Release 10 and later headers define a shorter **jmp_buf**, **sigjmp_buf** or **ucontext_t** area. The LE headers prior to Release 10 also define the shorter version of these data areas. If an XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp_buf**, **sigjmp_buf** or **ucontext_t** area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.

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 value of `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 38
- "setjmp() — Preserve Stack Environment" on page 718
- "siglongjmp() — Restore the Stack Environment and Signal Mask" on page 764
- "sigsetjmp() — Save Stack Environment and Signal Mask" on page 778

lrand48() — Generate Pseudorandom Number

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 drand48() and erand48() functions return non-negative, double-precision, floating-point values, uniformly distributed over the interval `[0.0,1.0]`.
- The lrand48() and nrand48() functions return non-negative, long integers, uniformly distributed over the interval `[0,2**31]`.
- The mrand48() and jrand48() functions 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)mod(2**48)     n>=0
```

The initial values of X, a, and c are:

```
   X(0) = 1
   a   = 5deece66d (base 16)
   c   = b         (base 16)
```

IBM XL C/C++ for z/VM provides storage to save the most recent 48-bit integer value of the sequence, `X(i)`. The drand48(), lrand48(), and mrand48() functions share this storage. You may reinitialize the value of `X(n)` in this storage by calling the lcong48(), seed48(), or srand48() function. Likewise, you may change the values of a and c by calling lcong48(). Thereafter, whenever you call seed48() or srand48() to change `X(n)`, the initial values of a and c are also reestablished.

Special Behavior for OpenExtensions Services

You can make lrand48() 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, IBM XL C/C++ for z/VM serializes access to the storage for X(n), a, and c by functions in the drand48 family when a multithreaded application calls them.

If you request thread-specific behavior and call lrand48() from thread t, lrand48() generates the next 48-bit integer value in a sequence of 48-bit integer values, X(t,i), for the thread t, according to the linear congruential formula:

```
X(t,n+1) = (a(t)X(t,n) + c(t))mod(2**48)     n>=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)
```

IBM XL C/C++ for z/VM provides storage that is specific to the thread t to save the most recent 48-bit integer value of the sequence, X(t,i), generated by drand48(), lrand48(), or mrand48(). You may reinitialize the value X(t,n) in this storage by calling lcong48(), seed48(), or srand48() from the thread t. Likewise, you may change the values of a(t) and c(t) for thread t by calling lcong48() from the thread. Thereafter, whenever you call seed48() or srand48() from the thread t to change X(t,n), the initial values of a(t) and c(t) are also reestablished.

Returned Value

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 OpenExtensions Services

If you request thread-specific behavior for the drand48 family and call lrand48 on thread t, lrand48() 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 42
- “drand48() — Generate Pseudo-Random Number” on page 178
- “erand48() — Generate Pseudo-Random Number” on page 187
- “jrand48() — Generate Pseudorandom Number” on page 436
- “lcong48() — Initialize Pseudorandom Number” on page 444
- “mrand48() — Generate Pseudorandom Number” on page 510
- “nrand48() — Pseudo-Random Number Generator” on page 523
- “seed48() — Initialize Pseudorandom Number” on page 679
- “srand48() — Initialize Pseudorandom Number” on page 801

lsearch() — Linear Search and Update

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

A linear search routine that returns a pointer into a table, indicating the where an entry may be found. If the entry is not found, `lsearch` adds it at the end of the table.

The *key* argument points to the entry being 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 that the user must supply (strcmp(), for example) and is called with two arguments that point to the elements being compared. The lsearch() function must return a value of zero 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 entry is found, lsearch() 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 errnos are defined for lsearch().

Related Information

- “search.h” on page 38
- “bsearch() — Search Arrays” on page 100
- “hsearch() — Search Hash Tables” on page 394
- “lfind() — Linear Search Routine” on page 447
- “tsearch() — Search Binary Tree” on page 896

lseek() — Change the Offset of a File

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

Format

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

off_t lseek(int fildes,off_t offset,int pos);
```

General Description

Changes the current file offset to a new position in a BFS 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.

The lseek() function 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 binary zeros.)

Seeking past the end of a file, however, does not automatically extend the length of the file. A write operation must occur before the file is actually extended.

Special Behavior for POSIX C

For character special files, lseek() sets the file offset to the specified value. The file offset value is ignored during the read/write processing to pseudoterminal character special files.

`int` *fildes*;
The file whose current file offset you want to change.

`off_t` *offset*;
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).

`int` *pos*;
One of the following symbols (defined in the unistd.h 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

For more information about using POSIX support, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Large Files for HFS

Applications that are compiled with the option LANGLVL(EXTENDED) and also define the Feature Test Macro (FTM) _LARGE_FILES before any headers are included will enable this function to operate on HFS files that are larger than 2 gig-1 in size. File size and offset fields will be enlarged to 63 bits in width so any other function operating on this file will have to be enabled with the same FTM.

Returned Value

If successful, lseek() returns the new file offset, measured in bytes from the beginning of the file.

If unsuccessful, lseek() returns a value of -1 and sets errno to one of the following values under the conditions described:

EBADF
fildes is not a valid open file descriptor.

EEXTLINK
This is not supported for external links.

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
fildes 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(fildes,-10,SEEK_END);
```

Related Information

- "unistd.h" on page 51
- "creat() — Create a New File or Rewrite an Existing One" on page 146
- "dup() — Duplicate an Open File Descriptor" on page 180
- "fcntl() — Control Open File Descriptors" on page 207
- "fseek() — Change File Position" on page 292
- "fsetpos() — Set File Position" on page 294
- "open() — Open a File" on page 527
- "read() — Read From a File " on page 628
- "sigaction() — Examine or Change a Signal Action" on page 749
- "write() — Write Data on a File or Socket" on page 975

lstat() — Get Status of File, External Link, or Symbolic Link

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions XPG4.2	both	

Format

```
#define _OPEN_SYS
#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 *buf*. 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 or external link, lstat() returns information about the link itself.

The information is returned in the following `stat` structure, defined in the sys/stat.h header file.

Table 20. 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 117.
`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.

Table 20. Elements of the stat Structure (continued)

Structure	Description
`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 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 mode.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_ISVMEXTL(*mode*)
Is nonzero for external links.

S_ISVMEXTL_EXEC(*mode*)
Is nonzero for CMS EXEC external links.

S_ISVMEXTL_DATA(*mode*)
Is nonzero for CMS DATA external links.

S_ISVMEXTL_MEl(*mode*)
Is nonzero for MOUNT external links.

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.

If lstat() successfully determines all this information, it stores it in the area indicated by the *buf* argument.

Large Files for HFS

Applications that are compiled with the option LANGLVL(EXTENDED) and also define the Feature Test Macro (FTM) _LARGE_FILES before any headers are included will enable this function to operate on HFS files that are larger than 2 gig-1 in size. File size and offset fields will be enlarged to 63 bits in width so any other function operating on this file will have to be enabled with the same FTM.

Returned Value

If successful, lstat() returns a value of 0.

If unsuccessful, lstat() returns a value of -1 and sets errno to one of the following values under the conditions described.

EACCES
The process does not have search permission on some component of the *pathname* prefix.

EINVAL
buf 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 *pathname* argument is greater than POSIX_SYMLOOP.

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 through pathconf().

ENOENT
No file named *pathname* exists, or *pathname* is an empty string.

ENOTDIR
A component of the *pathname* prefix is not a directory.

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>
#undef _OPEN_SYS
#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 46
- “sys/types.h” on page 46
- “chmod() — Change the Mode of a File or Directory” on page 117
- “chown() — Change the Owner or Group of a File or Directory” on page 120
- “creat() — Create a New File or Rewrite an Existing One” on page 146
- “dup() — Duplicate an Open File Descriptor” on page 180
- “fcntl() — Control Open File Descriptors” on page 207
- “fstat() — Get Status Information about a File” on page 296
- “link() — Create a Link to a File” on page 450
- “mkdir() — Make a Directory” on page 498
- “mkfifo() — Make a FIFO Special File” on page 499
- “open() — Open a File” on page 527
- “pipe() — Create an Unnamed Pipe” on page 539
- “read() — Read From a File ” on page 628
- “readlink() — Read the Value of a Symbolic Link” on page 633
- “remove() — Delete File” on page 659
- “stat() — Get File Information” on page 803
- “symlink() — Create a Symbolic Link to a Path Name” on page 854
- “unlink() — Remove a Directory Entry” on page 910
- “utime() — Set File Access and Modification Times” on page 912
- “write() — Write Data on a File or Socket” on page 975

l64a() — Convert Long to Base-64 String Representation

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

Format

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

char *l64a (long value);
```

General Description

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 table provides the character-digit equivalents:

Character
Digit represented

.
0

/
1

0-9
2–11

A-Z
12–37

a-z
38–63

Returned Value

Returns a pointer to the base-64 representation of *value*. If *value* is `0`, 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.

No errnos are defined for l64a().

Related Information

- "stdlib.h" on page 42
- "a64l() — Convert Base-64 String Representation to Long Integer" on page 90
- "strtoul() — Convert String to Unsigned Integer" on page 845

makecontext() — Modify User Context

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

Modifies the context specified by *ucp*, which getcontext() initialized. When this context resumes using setcontext() or swapcontext(), program execution continues by calling *func()* and 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` determines the context that will resume when makecontext() returns. If the `uc_link` member is not `0`, the process continues like after calling setcontext(), with the

context pointed to by the `uc_link` member. If the `uc_link` member is `0`, the process exits as if exit() were called. You should initialize the `uc_link` member prior to calling makecontext().

The makecontext() 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         A pointer to the context that will
                                 resume 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++

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

The makecontext() function has no returned value.

The following errno values are defined for makecontext():

EINVAL
The modified context 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 with insufficient 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. 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* 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 51
- "getcontext() — Get User Context" on page 318
- "setcontext() — Restore User Context" on page 704
- "swapcontext() — Save and Restore User Context" on page 849

malloc() — Reserve Storage Block

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.

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`.

Special Behavior for C++

The C++ keywords new and delete are not interoperable with calloc(), free(), malloc(), or realloc().

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

- For information about the System Programming C Facilities, see *z/OS: XL C/C++ Programming Guide*.
- “stdlib.h” on page 42
- “calloc() — Reserve and Initialize Storage” on page 103
- “free() — Free a Block of Storage” on page 278
- “realloc() — Change Reserved Storage Block Size” on page 637

maxcoll() — Return Maximum Collating Element

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	

Format

```
#include <collate.h>

collel_t maxcoll(void)
```

General Description

Returns the largest possible value of a collating element in the current locale.

Related Information

- "collate.h" on page 20
- "cclass() — Return Characters in a Character Class" on page 104
- "collequiv() — Return List of Equivalent Collating Elements" on page 132
- "collorder() — Return List of Collating Elements" on page 133
- "collrange() — Calculate the Range List of Collating Elements" on page 134
- "colltostr() — Return a String for a Collating Element" on page 136
- "getmccoll() — Get Next Collating Element from String" on page 345
- "getwmccoll() — Get Next Collating Element from Wide String" on page 386
- "ismccollel() — Identify a Multicharacter Collating Element" on page 429
- "strtocoll() — Return Collating Element for String" on page 839

maxdesc() — Get Socket Numbers to Extend Beyond Default Range

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	

Format

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

int maxdesc(int *totdesc,int *inetdesc);
```

General Description

Reserves space in the virtual machine for socket descriptor numbers.

totdesc
: A pointer to an integer containing a value 1 greater than the largest desired socket number. The maximum allowed value is 65535.

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 uses sockets for bulk mode I/O, set the integer pointed to by *totdesc* to the range of socket descriptors your application may use and add 1.

You must call maxdesc() before your program creates its first socket. Your program should use getstablesize() 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

If successful, maxdesc() returns a value of `0`.

If unsuccessful, maxcesc() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EALREADY
Your program called maxdesc() after creating a socket, after a call to setibmsockopt(), or after a previous call to maxdesc().

EFAULT
Using the *totdesc* parameter, as specified, results in an attempt to access storage outside the caller's virtual machine or storage not modifiable by the caller.

ENOMEM
Your virtual machine 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 up to 100 AF_INET and/or AF_INET6 family sockets. The socket numbers (socket descriptors) run from 0 through 99.

Related Information

- “sys/socket.h” on page 46
- “sys/types.h” on page 46
- “close() — Close a File” on page 127
- “getdtablesize() — Get File Descriptor Table Size” on page 324
- “getstablesize() — Get Socket Table Size” on page 374
- “listen() — Prepare Server for Incoming Client Requests” on page 452
- “open() — Open a File” on page 527
- “select() — Monitor Activity on Files/Sockets and Message Queues” on page 680

mblen() — Calculate Length of Multibyte Character

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 and may cause undefined results.

If the current locale supports EBCDIC DBCS characters, then the shift state is updated where applicable. (See the information about conforming to ANSI standards in the .) 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 mblen() function maintains the internal shift state that is altered by subsequent calls.

Returned Value

If *string* is NULL, mblen() returns the following values:

- Nonzero when DBCS-host code (EBCDIC systems) is used
- Nonzero if multibyte encodings are state-dependent
- Zero otherwise

If *string* is not NULL, mblen() returns:

- Zero if *string* points to the null character
- The number of bytes comprising the multibyte character
- -1 if *string* does not point to a valid multibyte character

Example

```
/* In this example we attempt to continue processing a string after having
   processed a second string.
 */
string1 = "a\x0F\x4243\x4344\x4445\x0Eb";
string2 = "abcd";

/* SI and SO are the shift-in and shift-out characters, and .X, .Y,
   and .Z (\x4243\x4344\x4445) are 2-byte values
   representing multibyte characters.  Using mblen(), let's see what
   happens in the following sequence of calls.
 */

n = mblen(NULL, MB_CUR_MAX)     /* n is nonzero, indicating
                                   state-dependent encoding; mblen()
                                   has forced the internal
                                   shift state to the initial state */
n = mblen(string1, MB_CUR_MAX)  /* n is 1, a is a multibyte character
                                   of length 1, internal shift state
                                   is at initial */
s = string1 + n;
n = mblen(NULL, MB_CUR,MAX)     /* n is 3, shift in plus two bytes,
                                   internal shift state is "shift-in" */
s = s + 3;
n = mblen(NULL, MB_CUR_MAX)     /* n is nonzero; shift state is at
                                   the initial state */
n = mblen(string2, MB_CUR_MAX)  /* n is 1; shift state is at the
                                   initial state */
                                /* go back to string1 */
n = mblen(s, MB_CUR_MAX)
```

Here is another example.

```
/* This example uses mblen() and mbtowc() to convert a multibyte
   character into a single wide character.
 */
#include <stdio.h>
#include <stdlib.h>

int length, temp;
char string [6] = "w";
wchar_t arr[6];

int main(void)

{
   /* Set string to point to a multibyte character  */
⋮
   length = mblen(string, MB_CUR_MAX);
   temp = mbtowc(arr,string,length);
   printf("wide character string: %ls",arr);
}
```

Related Information

- For information about locales and character sets, see *z/OS: XL C/C++ Programming Guide*.
- "locale.h" on page 30
- "stdlib.h" on page 42
- "mbrlen() — Calculate Length of Multibyte Character" on page 482
- "mbrtowc() — Convert a Multibyte Character to a Wide Character" on page 484
- "mbsrtowcs() — Convert a Multibyte String to a Wide-Character String" on page 487
- "mbstowcs() — Convert Multibyte Characters to Wide Characters" on page 489
- "mbtowc() — Convert Multibyte Character to Wide Character" on page 490
- "setlocale() — Set Locale" on page 721
- "strlen() — Determine String Length" on page 825
- "wcrtomb() — Convert a Wide Character to a Multibyte Character" on page 933
- "wcslen() — Calculate Length of Wide-Character String" on page 944
- "wcsrtombs() — Convert Wide-Character String to Multibyte String" on page 951
- "wctomb() — Convert Wide Character to Multibyte Character" on page 967

mbrlen() — Calculate Length of Multibyte Character

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 mbrlen().

The mbrlen() function 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.h> header file, you must also define the _MSE_PROTOS feature test macro to make the declaration of the mbrlen() in the <wchar.h> header file available when you compile your program.

Returned Value

If *s* is a null pointer, mbrlen() resets the shift state to the initial shift state and returns a value of `0`.

If *s* is not a null pointer, mbrlen() returns the first of the following values 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 returned value 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 mbrlen() returns a value of a -2 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, you should increment the pointer by the value *n*, and call 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

- For information about locales and character sets, see *z/OS: XL C/C++ Programming Guide*.
- "locale.h" on page 30
- "wchar.h" on page 53
- "mblen() — Calculate Length of Multibyte Character" on page 480
- "mbrtowc() — Convert a Multibyte Character to a Wide Character" on page 484
- "mbsrtowcs() — Convert a Multibyte String to a Wide-Character String" on page 487
- "mbtowc() — Convert Multibyte Character to Wide Character" on page 490
- "setlocale() — Set Locale" on page 721
- "wcrtomb() — Convert a Wide Character to a Multibyte Character" on page 933
- "wcsrtombs() — Convert Wide-Character String to Multibyte String" on page 951

mbrtowc() — Convert a Multibyte Character to a Wide Character

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

Format

Non-XPG4

```
#include <wchar.h>

size_t mbrtowc (wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);
```

XPG4

```
#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.

The mbrtowc() function 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 XPG4

If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the <wchar.h> header file, you must also define the _MSE_PROTOS feature test macro to make the declaration of mbrtowc() in the <wchar.h> header file available when you compile your program.

Returned Value

If *s* is a null pointer, mbrtowc() resets the shift state to the initial shift state and returns a value or `0`.

If *s* is not a null pointer, mbrtowc() returns the first of the following values that applies:

0
: If the next *n* or fewer bytes complete the valid multibyte character that corresponds to the null wide character.

positive integer
: If the next *n* or fewer bytes complete the valid multibyte character; mbrtowc() returns 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 mbrtowc() returns a value of -2 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, you should 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 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

- For information about locales and character sets, see *z/OS: XL C/C++ Programming Guide*.
- "locale.h" on page 30
- "wchar.h" on page 53
- "mblen() — Calculate Length of Multibyte Character" on page 480
- "mbrlen() — Calculate Length of Multibyte Character" on page 482
- "mbsrtowcs() — Convert a Multibyte String to a Wide-Character String" on page 487
- "setlocale() — Set Locale" on page 721
- "wcrtomb() — Convert a Wide Character to a Multibyte Character" on page 933
- "wcsrtombs() — Convert Wide-Character String to Multibyte String" on page 951

mbsinit() — Test State Object for Initial State

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.h> header file, you must also define the _MSE_PROTOS feature test macro to make the declaration of mbsinit() in the <wchar.h> header file available when you compile your program.

Returned Value

Returns a nonzero value if *ps* is a null pointer or if the pointed-to object describes an initial conversion state; otherwise, mbsinit() returns a value of 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

- For information about locales and character sets, see *z/OS: XL C/C++ Programming Guide*.
- "locale.h" on page 30
- "wchar.h" on page 53
- "mbrlen() — Calculate Length of Multibyte Character" on page 482
- "mbrtowc() — Convert a Multibyte Character to a Wide Character" on page 484
- "mbsrtowcs() — Convert a Multibyte String to a Wide-Character String" on page 487
- "setlocale() — Set Locale" on page 721
- "wcrtomb() — Convert a Wide Character to a Multibyte Character" on page 933
- "wcsrtombs() — Convert Wide-Character String to Multibyte String" on page 951

mbsrtowcs() — Convert a Multibyte String to a Wide-Character String

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*. The mbsrtowcs() function converts this sequence into a sequence of corresponding wide characters, that, if *dst* is not a null pointer, it then stores into the array pointed to by *dst*. Conversion continues up to and including a terminating null character, which it also stores. Conversion stops earlier in two cases: (1) when it reaches a sequence of bytes 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 mbrtowc().

If *dst* is not a null pointer, the pointer object specified 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 converted multibyte character. If conversion stopped because mbsrtowcs() reached a terminating null character, the resulting state is the initial state.

The mbsrtowcs() function 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.h> header file, you must also define the _MSE_PROTOS feature test macro to make the declaration of mbsrtowcs() in the <wchar.h> header file available when you compile your program.

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

- For information about locales and character sets, see *z/OS: XL C/C++ Programming Guide*.
- "locale.h" on page 30
- "wchar.h" on page 53
- "mblen() — Calculate Length of Multibyte Character" on page 480
- "mbrlen() — Calculate Length of Multibyte Character" on page 482
- "mbrtowc() — Convert a Multibyte Character to a Wide Character" on page 484
- "mbstowcs() — Convert Multibyte Characters to Wide Characters" on page 489
- "setlocale() — Set Locale" on page 721
- "wcrtomb() — Convert a Wide Character to a Multibyte Character" on page 933
- "wcsrtombs() — Convert Wide-Character String to Multibyte String" on page 951

mbstowcs() — Convert Multibyte Characters to Wide Characters

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*. The mbstowcs() function 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 mbstowcs() encounters an invalid multibyte sequence or converts *n* codes.

Processing continues up to and including the terminating null character; characters that follow it are not processed. The mbstowcs() function converts the terminating null character into a code with a value of `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.

Returned Value

If mbstowcs() encounters an invalid multibyte character, the function returns (size_t)-1. Otherwise, mbstowcs() returns the number of modified *pwc* array elements, not counting the terminating `0` code (the `wchar_t 0` code), which is `0` if *pwc* is a null pointer.

Note: 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

- For information about locales and character sets, see *z/OS: XL C/C++ Programming Guide*.
- “locale.h” on page 30
- “stdlib.h” on page 42
- “mblen() — Calculate Length of Multibyte Character” on page 480
- “mbsrtowcs() — Convert a Multibyte String to a Wide-Character String” on page 487
- “mbtowc() — Convert Multibyte Character to Wide Character” on page 490
- “setlocale() — Set Locale” on page 721
- “wcslen() — Calculate Length of Wide-Character String” on page 944
- “wcstombs() — Convert a Wide-Character String to Multibyte Character” on page 960

mbtowc() — Convert Multibyte Character to Wide Character

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. The mbtowc() function 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.

Returned Value

If *string* is NULL, the mbtowc() function returns the following:

- 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.

If *string* is not NULL, the mbtowc() function returns one of the following:

- 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 mbtowc() 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

- For information about locales and character sets, see *z/OS: XL C/C++ Programming Guide*.
- “locale.h” on page 30
- “stdlib.h” on page 42
- “mblen() — Calculate Length of Multibyte Character” on page 480
- “mbrtowc() — Convert a Multibyte Character to a Wide Character” on page 484
- “mbstowcs() — Convert Multibyte Characters to Wide Characters” on page 489
- “setlocale() — Set Locale” on page 721
- “wcslen() — Calculate Length of Wide-Character String” on page 944
- “wctomb() — Convert Wide Character to Multibyte Character” on page 967

memccpy() — Copy Bytes in Memory

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

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

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 / 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

A built-in function that 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*. The memchr() function 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 skixing
 */
#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 43
- “memcmp() — Compare Bytes” on page 493
- “memcpy() — Copy Buffer” on page 495
- “memmove() — Move Buffer” on page 496
- “memset() — Set Buffer to Value” on page 497
- “strchr() — Search for Character” on page 811

memcmp() — Compare Bytes

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

A built-in function that 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. The memcmp() 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 *buf1* is less than the contents of the buffer pointed to by *buf2*

0
The contents of the buffer pointed to by *buf1* is identical to the contents of the buffer pointed to by *buf2*

Greater than 0
The contents of the buffer pointed to by *buf1* is greater than the contents of the buffer pointed to by *buf2*.

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 43
- “memchr() — Search Buffer” on page 492
- “memcpy() — Copy Buffer” on page 495

- “memmove() — Move Buffer” on page 496
- “memset() — Set Buffer to Value” on page 497
- “strcmp() — Compare Strings” on page 812

memcpy() — Copy Buffer

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

Copies *count* bytes from the object pointed to by *src* to the object pointed to by *dest*. For memcpy(), the behavior is undefined if copying takes place between objects that overlap. (The memmove() function allows 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 43
- “memchr() — Search Buffer” on page 492
- “memcmp() — Compare Bytes” on page 493
- “memmove() — Move Buffer” on page 496

- “memset() — Set Buffer to Value” on page 497
- “strcpy() — Copy String” on page 815

memmove() — Move Buffer

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* were 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 white sphere"
```

Related Information

- “string.h” on page 43
- “memchr() — Search Buffer” on page 492

- “memcmp() — Compare Bytes” on page 493
- “memcpy() — Copy Buffer” on page 495
- “memset() — Set Buffer to Value” on page 497
- “strcpy() — Copy String” on page 815

memset() — Set Buffer to Value

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

A built-in function that 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 43

- “memchr() — Search Buffer” on page 492
- “memcmp() — Compare Bytes” on page 493
- “memcpy() — Copy Buffer” on page 495
- “memmove() — Move Buffer” on page 496

mkdir() — Make a Directory

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

Format

```
#define _OPEN_SYS
#include <sys/stat.h>

int mkdir(char *pathname, mode_t mode);
```

General Description

Creates a new, empty directory, *pathname*. The file creation mask of the process modifies the file permission bits in *mode* and uses them to set the file permission bits of the directory being created. For more information on the file creation mask, see “umask() — Set or Display the File Mode Creation Mask” on page 902; for information about the file permission bits, see “chmod() — Change the Mode of a File or Directory” on page 117.

The mkdir() function sets the owner ID of the new directory to the effective user ID of the process and sets the group ID of the new directory to the group ID of the owning directory.

The mkdir() function 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 a value of `0`.

If unsuccessful, mkdir() does not create a directory; it returns a value of -1 and sets errno to one of the following values under the conditions described:

EACCES
The process did not have search permission on some component of *pathname*, 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 a file or directory with the given *pathname* already exists.

EINVAL
The byte file system (BFS) path name begins with "//", which is not allowed for BFS files.

ELOOP
A loop exists in symbolic links. The mkdir() function issues this error if more than POSIX_SYMLOOP (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 LINK_MAX (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 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 directory, or the parent directory cannot be extended.

ENOTDIR
A component of the *pathname* 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 _OPEN_SYS
#include <sys/stat.h>
#include <unistd.h>
#undef _OPEN_SYS
#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 28
- “sys/stat.h” on page 46
- “chdir() — Change the Working Directory” on page 116
- “chmod() — Change the Mode of a File or Directory” on page 117
- “stat() — Get File Information” on page 803
- “umask() — Set or Display the File Mode Creation Mask” on page 902

mkfifo() — Make a FIFO Special File

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

Format

```
#define _OPEN_SYS
#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.

The mkfifo() function creates a new FIFO special file, *pathname*. The file creation mask of the process changes the file permission bits in *mode* and uses them 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 or Display the File Mode Creation Mask" on page 902; for information about the file permission bits, see "chmod() — Change the Mode of a File or Directory" on page 117.

The mkfifo() function sets the owner ID of the FIFO file to the effective user ID of the process and the group ID of the FIFO file to the group ID of the owning directory; *pathname* cannot end in a symbolic link.

The mkfifo() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, mkfifo() returns a value or 0. If mkfifo() fails, no FIFO file is created, the function returns a value of -1, and sets errno to one of the following values under the conditions described:

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 a file or directory with the given *pathname* already exists.

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).

EINVAL
The BFS path name begins with "//", which is not allowed for BFS files.

ELOOP
A loop exists in symbolic links. The mkfifo() function issues this error if it detects more than POSIX_SYMLOOP (defined in the limits.h header file) symbolic links 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 _OPEN_SYS
#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 28
- “sys/stat.h” on page 46
- “chmod() — Change the Mode of a File or Directory” on page 117
- “open() — Open a File” on page 527
- “pipe() — Create an Unnamed Pipe” on page 539
- “stat() — Get File Information” on page 803
- “umask() — Set or Display the File Mode Creation Mask” on page 902

mknod() — Make a FIFO or Charcter Special File

Standards / Extensions	C or C++	Dependencies
XPG4.2 z/VM OpenExtensions	both	

Format

```
#define _OPEN_SYS
#include <sys/stat.h>

int mknod(const char *path, mode_t mode, rdev_t dev_identifier);
```

General Description

Creates a new character special file or FIFO special file (named pipe), with the path name specified in the *path* argument.

The first byte of the *mode* argument determines the file type of the special file:

S_IFCHR
Character special file

S_IFFIFO
FIFO special file

The file permission bits of the new special 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 117.

dev_identifier applies only to a character special file, not to a FIFO special file (named pipe); *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 OpenExtensions, the device major numbers are:

```
3    /dev/tty
4    /dev/null
5    /dev/fdn
```

Device major numbers 3 and 4: The device minor number is ignored.

Device major number 5: The device minor number value represents the file descriptor 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.

The mknod() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, mknod() returns a value of `0`.

If unsuccessful, mknod() returns a value of -1 and sets errno to one of the following values under the conditions described:

EACCES
Search permission is denied on a component of *path*, or write permission is denied on the parent directory of the file to be created.

EEXIST
A file by that name already exists.

EINVAL
The BFS path name starts with "//", which is not allowed for BFS files.

ELOOP
A loop exists in symbolic links. The mknod() function issues this error if it detects more than POSIX_SYMLOOP (defined in the limits.h header file) symbolic links in the resolution of *path*.

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 through pathconf().

ENOENT
A component of *path* was not found, or no *path* 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 *path* is not a directory

EROFS
The file named in *path* cannot be created because it would reside on a read-only file system.

Example

CBC3BM18

```
/* CBC3BM18 */
#define _OPEN_SYS
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>

#define manager 0x00010000

main() {
  char fn[]="char.spec";

  if (mknod(fn, S_IFCHR¦S_IRUSR¦S_IWUSR, manager¦0x0001) != 0)
    perror("mknod() error");
  else if (unlink(fn) != 0)
    perror("unlink() error");
}
```

Related Information

- “limits.h” on page 28
- “sys/stat.h” on page 46
- “mkfifo() — Make a FIFO Special File” on page 499

mkstemp() — Make a Unique File Name

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

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 mkstemp() 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, such that the resulting name does not duplicate the name of an existing file.

Returned Value

If successful, mkstemp() returns an open file descriptor.

If no suitable file could be created, mkstemp() returns a value of -1.

No errnos are defined for mkstemp().

Related Information

- "stdlib.h" on page 42
- "mktemp() — Make a Unique File Name" on page 504

mktemp() — Make a Unique File Name

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

Replaces the contents of the string pointed to by *template* with a unique file name. The application must initialize *template* to be a file name with six trailing X ' s; mktemp() replaces each X with a single byte character from the portable file name character set.

This function is supported only in a POSIX program.

Returned Value

If successful, mktemp() returns a pointer to *template*.

If a unique name cannot be created, *template* points to a null string.

No errors are defined for mktemp().

Related Information

- "stdlib.h" on page 42
- "mkstemp() — Make a Unique File Name" on page 503

mktime() — Convert Local Time

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 18 on page 392.

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" on page 391. The mktime() function ignores the values of `tm_wday` and `tm_yday` and assigns their correct values on return.

If successful, mktime() sets all the members of the structure pointed to by `time` to represent the specified time with their values forced into the ranges described in "gmtime() — Convert Time to Broken-Down UTC Time" on page 391. The function sets the values of `tm_wday` after it determines the values of `tm_mon` and `tm_year`.

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(), the function returns `(time-t)-1` to indicate an error.

Note: The mktime() 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.

For more information about using POSIX support, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

Returns the calendar time corresponding to broken-down time, expressed as local time, using the `tm` structure, which is pointed to by *tmptr*. 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.

- The ctime(), localtime(), and mktime() functions now return coordinated, 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 49
- “ctime() — Convert Time to a Character String” on page 153
- “gmtime() — Convert Time to Broken-Down UTC Time” on page 391
- “localtime() — Convert Time and Correct for Local Time” on page 456
- “time() — Determine Current Time” on page 885

modf() — Extract Fractional and Integral Parts of a FLoating-Point Value

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++	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 and stores these parts 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 32
- “frexp() — Extract Mantissa and Exponent of the Floating-Point Value” on page 282
- “ldexp(), ldexpf(), ldexpl() — Multiply by a Power of Two” on page 445

mount() — Make a File System Available

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	

Format

```
#define _OPEN_SYS
#include <sys/stat.h>
#include <bpxhmnt.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 byte file system directory tree for a user. The same file system cannot be mounted at more than one place in the byte file system (BFS).

Any user can mount a file system into the BFS hierarchy of the virtual machine.

path
The mount point directory to which the file system is to be mounted. An NFS file system cannot be mounted as the root directory ("/").

filesystem
The name of the file system to be mounted; it must be unique within the system. For a Byte File System (BFS) or a Network File System (NFS), this is the fully qualified file system name.

This name is terminated with null characters.

filesystype
The type of file system to mount. For BFS, this must be the string VMBFS. For NFS, this must be the string BPXFSNFS.

mtm
This is a flag field that specifies the mount mode:

MTM_RDONLY
Mount the file system as a read-only file system.

MTM_RDWR
Mount the file system as a read/write file system.

parmlen
The length of the *parm* argument. The maximum length is 1024 characters. For BFS, this parameter is not specified. For NFS, this parameter can be 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. BFS does *not* require a *parm*. For NFS, this parameter can be specified. See “bpxhmnt.h” on page 19 for the format and content of *parm*.

The mount() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

OpenExtensions supports up to ten concurrent mount points per user.

Returned Value

If successful, mount() returns a value or 0.

If unsuccessful, mount() returns a value of -1 and sets errno to one of the following values under the conditions described:

EACCESS
Permission denied.

EBUSY
The specified file system is unavailable.

EINVAL
A parameter was incorrectly specified. Verify *filesystype* and *mtm*. 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
Input/Output error.

ECMSSTORAGE
Only enough storage is available to support up to ten concurrent mount points.

ELOOP
A loop exists in symbolic links. The mount() function issues this error if it detects more than POSIX_SYMLOOP (defined in the limits.h header file) symbolic links in the resolution of *pathname*.

ENAMETOOLONG
Either the path or file system name is longer than PATH_MAX characters, or some component is longer than NAME_MAX characters.

ENOENT
The mount point does not exist.

ENOMEM
Not enough storage is available to save the information required for this file system.

ENOTDIR
The mount point is not a directory.

EPERM
Operation not permitted.

Example

CBC3BM21

```
/* CBC3BM21
   This example adds a file system to the user's logical file system
   hierarchy.
 */
#define _OPEN_SYS
#include <sys/stat.h>
#include <stdio.h>
#include <unistd.h>

main() {
  char mount_point--="/";
  char BFS--="/../VMBFS:VMSYS:ROOT/";
  char filesystype-9-="VMBFS   ";

  puts("before mount()");
  system("EXEC OPENVM QUERY MOUNT");
  if (mount(mount_point, BFS, filesystype, MTM_RDWR, 0, NULL) != 0)
    perror("mount() error");
  else {
    puts("After mount()");
    system("EXEC OPENVM QUERY MOUNT");
    if (umount(BFS, MTM_UMOUNT) != 0)
      perror("umount() error");
  }
}
```

Output

before mount()

```
Nothing is mounted
```

after mount()

```
Mount point = '/'
Type Stat Mounted
BFS  R/W '/../VMBFS:VMSYS:ROOT/'
```

Related Information

- "limits.h" on page 28
- "sys/stat.h" on page 46
- "umount() — Remove a Virtual File System" on page 903

mrand48() — Generate Pseudorandom Number

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

Format

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

long int mrand48(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 drand48() and erand48() functions return non-negative, double-precision, floating-point values, uniformly distributed over the interval `[0.0,1.0]`.
- The lrand48() and nrand48() functions return non-negative, long integers, uniformly distributed over the interval `[0,2**31]`.
- The mrand48() and jrand48() functions return signed long integers, uniformly distributed over the interval `[-2**31,2**31]`.

The mrand48() 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)mod(2**48)     n>=0
```

The initial values of X, a, and c are:

```
   X(0) = 1
   a    = 5deece66d (base 16)
   c    = b         (base 16)
```

C VM/ESA provides storage to save the most recent 48-bit integer value of the sequence, `X(i)`. The drand48(), lrand48(), and mrand48() functions share this storage. You may reinitialize the value of `X(n)` in this storage by calling the lcong48(), seed48(), or srand48() function. Likewise, you may change the

values of a and c by calling lcong48(). Thereafter, whenever you call seed48() or srand48() to change `X(n)`, the initial values of a and c are also reestablished.

Special Behavior for OpenExtensions Services

You can make mrand48() 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 VM/ESA serializes access to the storage for `X(n)`, a, and c by functions in the `drand48` family when a multithreaded application calls them.

If you request thread-specific behavior and call mrand48() from thread `t`, mrand48() generates the next 48-bit integer value in a sequence of 48-bit integer values, `X(t,i)`, for the thread `t`, according to the linear congruential formula:

```
X(t,n+1) = (a(t)X(t,n) + c(t))mod(2**48)     n>=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 VM/ESA provides storage that is specific to the thread `t` to save the most recent 48-bit integer value of the sequence, `X(t,i)`, generated by drand48(), lrand48(), or mrand48(). You may reinitialize the value `X(t,n)` in this storage by calling lcong48(), seed48(), or srand48() from the thread `t`. Likewise, you may change the values of `a(t)` and `c(t)` for thread `t` by calling lcong48() from the thread. Thereafter, whenever you call seed48() or srand48() from the thread `t` to change `X(t,n)`, the initial values of `a(t)` and `c(t)` are also reestablished.

Returned Value

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 OpenExtensions Services

If you request thread-specific behavior for the `drand48` family and call `mrand48` on thread `t`, mrand48() 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 42
- “drand48() — Generate Pseudo-Random Number” on page 178
- “erand48() — Generate Pseudo-Random Number” on page 187
- “jrand48() — Generate Pseudorandom Number” on page 436
- “lcong48() — Initialize Pseudorandom Number” on page 444
- “lrand48() — Generate Pseudorandom Number” on page 467
- “nrand48() — Pseudo-Random Number Generator” on page 523
- “seed48() — Initialize Pseudorandom Number” on page 679
- “srand48() — Initialize Pseudorandom Number” on page 801

msgctl() — Message Control Operations

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

Format

```
#define _XOPEN_SOURCE
#include <sys/msg.h>

int  msgctl(int msgid, int cmd, struct msqid_ds *buf);
```

General Description

Provides message control operations, as specified by *cmd*, which may have the following values. (The <sys/ipc.h> header defines these symbolic constants.)

IPC_STAT
Places 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
Sets 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_qbytes`

IPC_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
Removes the message queue identifier, *msgid* from the system and destroys the message queue and `msqid_ds` data structure associated with it.

IPC_RMID 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*.

Returned Value

If successful, msgctl() returns a value of `0`.

If unsuccessful, msgctl() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EACCES
The *cmd* argument 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 *cmd* argument is IPC_RMID or IPC_SET and the effective user ID of the calling process is not a process with appropriate privileges or the value of `msg_perm.cuid` or `msg_perm.uid` in the data structure associated with *msgid*.

If the *cmd* argument 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 45
- "sys/msg.h" on page 45
- "msgget() — Get Message Queue" on page 513
- "msgrcv() — Read and Receive a Message" on page 515
- "msgsnd() — Send a Message" on page 516
- "msgxrcv() — Read and Receive an Extended Message" on page 518

msgget() — Get Message Queue

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

Format

```
#define _XOPEN_SOURCE
#include <sys/msg.h>

int  msgget(key_t key, int msgflg);
```

General Description

Creates a message queue identifier, associated message queue, and data structure (see <sys/msg.h>) for the *key* argument if one of the following is true:

- The *key* argument is equal to IPC_PRIVATE.
- The *key* argument does not already have a message queue identifier associated with it, and the IPC_CREAT flag is on in *msgflg*.

Valid values for the *msgflg* argument include any combination of the following constants defined in <sys/ipc.h> and <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 msgget() 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.

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 `msg_perm.cgid` or `msg_perm.gid`.

S_IWGRP
Permits write access when the effective group ID of the caller matches either `msg_perm.cgid` or `msg_perm.gid`.

S_IROTH
Permits other read access.

S_IWOTH
Permits other write access.

When a message queue associated with *key* already exists, setting IPC_EXCL and IPC_CREAT in *msgflg* will force msgget() to fail.

The `msg_ds` data structure associated with the new message queue identifier is initialized as follows:

- The `msg_perm.cuid`, `msg_perm.uid`, `msg_perm.cgid`, and `msg_perm.gid` fields are set 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 to the low-order 9 bits of *msgflg*.
- The `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime`, and `msg_rtime` fields are set to zero.
- The `msg_ctime` field is set to the current time.
- The `msg_qbytes` field is set to the system limit.

Returned Value

If successful, msgget() returns a non-negative integer, namely a message queue identifier.

If unsuccessful, msgget() returns a value of -1 and sets errno to one of the following values under the conditions described:

EACCES
A message queue identifier exists for *key*, but access permission, as specified by the low order 9 bits of *msgflg*, could not be granted.

EEXIST
A message queue identifier exists for *key*, but both IPC_CREAT and IPC_EXCL are specified in *msgflg*.

EINVAL
The value of *msgflg* is not currently supported.

ENOENT
A message queue identifier does not exist for *key*, and IPC_CREAT 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 will be exceeded.

When *msgflg* is `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, 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, msgget() returns a value of -1 and sets errno to EACCES.
- If a message queue identifier has not been created with *key* earlier, msgget() returns a value of -1 and sets errno to ENOENT.

Related Information

- “sys/ipc.h” on page 45
- “sys/msg.h” on page 45
- “sys/types.h” on page 46

- “ftok() — Generate an Interprocess Communication (IPC) Key” on page 302
- “msgctl() — Message Control Operations” on page 512
- “msgrcv() — Read and Receive a Message” on page 515
- “msgsnd() — Send a Message” on page 516
- “msgxrcv() — Read and Receive an Extended Message” on page 518

msgrcv() — Read and Receive a Message

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 Single UNIX Specification, Version 2	both	

Format

```
#define _XOPEN_SOURCE
#include <sys/msg.h>

int  msgrcv(int msgid, void *msgp, size_t msgsz, long int msgtyp, int msgflg);
```

General Description

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 *msgp* argument 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 buffer should look like:

```
              struct message {
                 long int   mtype;    Message type
                 int        mtext[n]; Message text
           }
```

The `mtype` structure member is the received message's type as specified by the sending process. The `mtext` structure member is the text of the message.

The *msgsz* argument specifies the size in bytes of `mtext`. The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and you specified the the MSG_NOERROR flag in the *msgflg* argument. The truncated portion of the message is lost and no indication of the truncation is given to the calling process.

The *msgtyp* argument specifies the type of message requested, as follows:

- If *msgtyp* is 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 *msgflg* message specifies the action to be taken if a message of the desired type is not on the queue. These actions 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, a value of -1 is returned, and errno is set to EIDRM.
- The calling process receives a signal to be caught, no message is 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 `msgid_ds` data structure associated with *msgid*:

1. `msg_qnum` is decremented by 1.
2. `msg_lrpid` is set to the process ID of the calling process.
3. `msg_rtime` is set 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 *msgp*. A value of zero indicates that only the `mtype` field was received from the message queue.

If unsuccessful, msgrcv() returns a value of -1, and sets errno to one of the following values under the conditions described.

E2BIG
The value of *mtext* is greater than *msgsz*, and the MSG_NOERROR flag was not specified.

EACCES
The calling process does not have read permission to the message queue associated with the message queue identifier, *msgid*.

EIDRM
The message queue identifier, *msgid*, was removed from the system while the caller of msgrcv() was waiting.

EINTR
The msgrcv() function was interrupted by a signal before a message could be received.

EINVAL
The value of *msgid* is not a valid message queue identifier, or the value of *msgsz* is less than 0.

ENOMSG
The IPC_NOWAIT flag was specified, and the message queue does not contain a message of the desired type.

Related Information

- “sys/ipc.h” on page 45
- “sys/msg.h” on page 45
- “msgctl() — Message Control Operations” on page 512
- “msgget() — Get Message Queue” on page 513
- “msgsnd() — Send a Message” on page 516
- “msgxrcv() — Read and Receive an Extended Message” on page 518

msgsnd() — Send a Message

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

Format

```
#define _XOPEN_SOURCE
#include <sys/msg.h>

int  msgsnd(int msgid, const void  *msgp, size_t msgsz, int msgflg);
```

General Description

Sends a message to the queue associated with the message queue identifier, *msgid*.

The *msgp* argument 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 buffer should look like:

```
struct message {
      long int   mtype;    Message type
      int        mtext[n];  Message text
}
```

The `mtype` structure member must be a nonzero positive value that the receiving process can use for message selection. The `mtext` structure member is any text of length, *msgsz* bytes; *msgsz* is set to zero when no `mtext` accompanies `mtype`. It can range from zero to a system-imposed maximum or the maximum number of bytes allowed in the message queue.

The *msgflg* argument 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 will not be sent and the calling process will return immediately with a return value of -1 and errno set to EAGAIN.
- If the IPC_NOWAIT flag is off in *msgflg*, the calling process will suspend execution until one of the following occurs:
  - The condition responsible for the suspension no longer exists, in which case the message is sent.
  - The message queue identifier, *msgid*, is removed from the system; when this occurs, msgsnd() returns a value of -1 and sets errno to EIDRM.
  - The calling process receives a signal to be caught, no message is 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 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`.

If unsuccessful, no message is sent, msgsnd() returns a value of -1, and sets errno to one of the following values under the conditions described:

EACCES
The calling process does not have write permission to the message queue associated with *msgid*.

EAGAIN
The message cannot be sent for one of the reasons cited above, and IPC_NOWAIT was specified.

EIDRM
The message queue identifier, *msgid*, was removed from the system while the caller of msgsnd() was waiting.

EINTR
The msgsnd() function was interrupted by a signal before a message could be sent.

EINVAL
The value of *msgid* is not a valid message queue identifier, or the value of `mtype` is less than 1; or the value of *msgsz* is less than `0` or greater than the system-imposed limit.

ENOMEM
Insufficient system storage space prevents completion of msgsnd().

Related Information

- "sys/ipc.h" on page 45
- "sys/msg.h" on page 45
- "msgctl() — Message Control Operations" on page 512
- "msgget() — Get Message Queue" on page 513
- "msgrcv() — Read and Receive a Message" on page 515
- "msgxrcv() — Read and Receive an Extended Message" on page 518

msgxrcv() — Read and Receive an Extended Message

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	

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, you should define the feature test macro _OPEN_SYS_IPC_EXTENSIONS. Otherwise, the function's name is __msgxrcv().

General Description

Reads an extended message from the queue associated with the message queue identifier, *msgid*, and places it in the user-defined buffer, pointed to by *msgp*.

This buffer should have 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 `mtype` structure member is the received message's type as specified by the sending process. The `mtext` structure member is the text of the message.

The *msgsz* argument specifies the size in bytes of `mtext`, The received message is truncated to *msgsz* bytes if it is larger than *msgsz* and you specified the MSG_NOERROR flag in the *msgflg* argument. The truncated portion of the message is lost, and no indication of the truncation is given to the calling process.

The *msgtyp* argument specifies the type of message requested, as follows:

- If *msgtyp* is zero, the first message on the queue is received.
- If *msgtyp* is greater than zero, the first message of *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.

If a message of the desired type is not on the queue, the *msgflg* argument specifies that the following actions be taken under the conditions described:

- 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, a value of `-1` is returned and errno is set to EIDRM.
  - The calling process receives a signal to be caught, no message is 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 `msgid_ds` data structure associated with *msgid*:

1. `msg_qnum` is decremented by 1.
2. `msg_lrpid` is set to the process ID of the calling process.
3. `msg_rtime` is set to the current time.

Returned Value

If successful, msgxrcv() returns a value equal to the number of bytes actually placed into the `mtext` field *msgp*. A value of `0` indicates that only the `mtype` field was received from the message queue.

If unsuccessful, msgxrcv() returns a value of `-1` and sets errno to one of the following values under the conditions described:

E2BIG
The value of `mtext` is greater than *msgsz*, and the MSG_NOERROR flag was not specified.

EACCES
The calling process does not have read permission to the message queue associated with *msgid*.

EIDRM
The message queue identifier, *msgid*, was removed from the system while the caller of msgxrcv() was waiting.

EINTR
The msgxrcv() function was interrupted by a signal before a message could be received.

EINVAL
The value of *msgid* is not a valid message queue identifier, or the value of *msgsz* is less than `0`.

ENOMSG
The IPC_NOWAIT flag was specified, and the message queue does not contain a message of the desired type.

Related Information

- “sys/ipc.h” on page 45

- "sys/msg.h" on page 45
- "msgctl() — Message Control Operations" on page 512
- "msgget() — Get Message Queue" on page 513
- "msgrcv() — Read and Receive a Message" on page 515
- "msgsnd() — Send a Message" on page 516

nextafter() — Calculate Next Representable Double Float

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

Calculates 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*.

Note: This function works in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

Returned Value

The nextafter() function is always successful.

Related Information

- "math.h" on page 32

nftw() — Traverse a File Tree

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

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

Recursively descends the directory hierarchy rooted in *path*. It is similar to ftw() except that it takes an additional *flags* argument, 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 file system 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 it will not report the same file twice.

At each file it encounters, nftw() calls the function, *fn*, you have supplied with four arguments:

- The first argument is the path name 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 *flags*.)

  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 *fn*.
- The fourth argument is a pointer to an FTW structure. The value of *base* is the offset of the object's file name in the path name passed as the first argument to *fn()*. The value of *level* indicates the depth relative to the root of the walk, where the root level is `0`.

The *depth* argument 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 a value of `-1` and sets errno to indicate the error.
- The tree is exhausted, in which case nftw() returns a value of `0`.

The nftw() function will fail under the following conditions, setting errno to one of the following values under the conditions described. All other errnos returned by nftw() are unchanged.

EACCES
Search permission is denied for any component of *path*, or read permission is denied for *path*, or *fn()* returns a value of -1 and does not reset errno.

ELOOP
Too many symbolic links were detected.

EMFILE
{OPEN_MAX} file descriptors are currently open in the calling process.

ENAMETOOLONG
Path name resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

ENAMETOOLONG
The length of *path* exceeds {PATH_MAX}, or a path name component is longer than {PATH_MAX}.

ENFILE
Too many files are currently open in the system.

ENOENT
A component of *path* does not name an existing file. or *path* is an empty string.

ENOTDIR
A component of *path* is not a directory.

Errno may also be set if *fn* causes it to be set.

Related Information

- "ftw.h" on page 26
- "ftw() — Traverse a File Tree" on page 305
- "lstat() — Get Status of File, External Link, or Symbolic Link" on page 471
- "opendir() — Open a Directory" on page 531
- "readdir() — Read an Entry from a Directory" on page 632
- "stat() — Get File Information" on page 803

nl_langinfo() — Retrieve Locale Information

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*.

You cannot retrieve the following information from the current locale:

alt_digits
Defines alternative symbols for digits, corresponding to the %0 field descriptor modifier.

era
Defines how the years are counted and displayed for each era (or emperor's reign) in a locale.

era_d_fmt
Defines the format of the date in alternative `era` notation, corresponding to the %Ex field descriptor.

era_year
Defines the format of the year in alternative `era` format, corresponding to the %EY field descriptor.

t_fmt_ampm
Defines the appropriate time representation in the 12-hour clock format with am_pm, corresponding to the `%r` field descriptor.

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. Subsequent calls to the function modify the array pointed to by the returned value. The user's program will not modify the array.

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
   nll_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 28
- “nl_types.h” on page 36
- “localdtconv() — Date/Time Formatting Convention Inquiry” on page 453
- “localeconv() — Query Numeric Conventions” on page 454
- “setlocale() — Set Locale” on page 721

nrand48() — Pseudo-Random Number Generator

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

Format

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

```
long int nrand48(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 drand48() and erand48() functions return non-negative, double-precision, floating-point values, uniformly distributed over the interval `[0.0,1.0]`.
- The lrand48() and nrand48() functions return non-negative, long integers, uniformly distributed over the interval `[0,2**31]`.
- The mrand48() and jrand48() functions return signed long integers, uniformly distributed over the interval `[-2**31,2**31]`.

The nrand48() 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)mod(2**48)     n>=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(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)
```

You may change the values of a and c by calling the lcong48() function. Calling either seed48() or srand48() restores a and c.

Special Behavior for OpenExtensions Services

You can make nrand48() 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 VM/ESA serializes access to the storage for `X(n)`, a, and c by functions in the `drand48` family when a multithreaded application calls them.

If you request thread-specific behavior and call nrand48() from thread `t`, nrand48() generates the next 48-bit integer value in a sequence of 48-bit integer values, `X(t,i)`, for the thread `t`, according to the linear congruential formula:

```
X(t,n+1) = (a(t)X(t,n) + c(t))mod(2**48)     n>=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(i)`. The 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)
```

You may change the values of `a(t)` and `c(t)` by calling the lcong48() function from the thread `t`. Calling either seed48() or srand48() from the thread restores the initial values of `a(t)` and `c(t)`.

Returned Value

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 OpenExtensions Services

If you request thread-specific behavior for the `drand48` family and call nrand48() on thread `t`, nrand48() 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 42
- "drand48() — Generate Pseudo-Random Number" on page 178
- "erand48() — Generate Pseudo-Random Number" on page 187
- "jrand48() — Generate Pseudorandom Number" on page 436
- "lcong48() — Initialize Pseudorandom Number" on page 444
- "lrand48() — Generate Pseudorandom Number" on page 467
- "mrand48() — Generate Pseudorandom Number" on page 510
- "seed48() — Initialize Pseudorandom Number" on page 679
- "srand48() — Initialize Pseudorandom Number" on page 801

ntohl() — Translate a Long Integer into Host Byte Order

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

Translates a long integer from network byte order to host byte order.

a
: The unsigned long integer to be put into host byte order.

`in_addr_t` *netlong*
: Is typed to the unsigned long integer to be put into host byte order.

Notes:

- Host byte order and network byte order are the same.

- Since this function is implemented as a macro, you need one of the feature test macros and the inet header file.

Returned Value

Returns the translated long integer.

Related Information

- "htonl() — Translate Address Host to Network Long" on page 395
- "htons() — Translate an Unsigned Short Integer into Network Byte Order" on page 396
- "ntohs() — Translate an Unsigned Short Integer into Host Byte Order" on page 526

ntohs() — Translate an Unsigned Short Integer into Host Byte Order

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

Translates a short integer from network byte order to host byte order.

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:

- Host byte order and network byte order are the same.
- Since this function is implemented as a macro, you need one of the feature test macros and the inet header file.

Returned Value

Returns the translated short integer.

Related Information

- "htonl() — Translate Address Host to Network Long" on page 395
- "htons() — Translate an Unsigned Short Integer into Network Byte Order" on page 396
- "ntohl() — Translate a Long Integer into Host Byte Order" on page 525

open() — Open a File

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

Format

```
#define _OPEN_SYS
#include <fcntl.h>

int open(const char *pathname, int options, …);
```

General Description

Opens a file and returns a number called a *file descriptor*.

The *pathname* argument must be a byte file system 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: FIFOs, POSIX terminals, and character special files applies only to C VM/ESA programs running POSIX(ON). For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

The *pathname* argument is a string giving the name of the file you want to open. The *options* integer 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.

You need to include an additional argument (`...`) 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 use when creating a file. 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. You can use the following symbols 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 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.

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, some options can change this position. One of the following *must* be specified in the *options* argument of the openf() operation:

O_RDONLY
Open for reading only

O_WRONLY
Open for writing only

O_RDWR
Open for both reading and writing

One 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 open() has a *mode* 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 *mode*.

If O_CREAT is specified and the file did not previously exist, a successful open() 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.
- No controlling terminal exists 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.

Returned Value

If successful, open() returns a file descriptor.

If unsuccessful, open() returns a value of -1 and sets errno to one of the following values under the conditions described:

EACCES
Access is denied. Possible reasons include the following:

- The process does not have search permission on a component in *pathname*.
- The file exists, but the process does not have permission 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.

EBUSY
The process attempted to open a file that is in use.

EEXIST
O_CREAT and O_EXCL were specified, and either the named file refers to a symbolic link, or the named file already exists.

EEXTLINK
An attempt was made to open an external link. The attempt failed for one of the following reasons:

- O_RDONLY was specified, but the file pointed to by the external link does not exist.
- The external link points to a BFS file.

EINTR
The open() function was interrupted by a signal.

EIO
The file is currently open and has encountered an I/O error.

EISDIR
pathname is a directory, and *options* specifies write or read/write access.

ELOOP
A loop exists in symbolic links. The open() function issues this error if it detects the number of symbolic links in the resolution of *pathname* is greater than POSIX_SYMLOOP.

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, 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 system has reached the maximum number of file descriptors it can have open.

ENOENT
Typical causes are as follows:

- O_CREAT is not specified, and the named file does not exist.
- O_CREAT is specified, and either the prefix of *pathname* 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.

ENOSYS
For manager pseudoterminals, subsidiary initialization did not complete.

ENOTDIR
A component of *pathname* 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.

EROFS
pathname 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 25
- “close() — Close a File” on page 127
- “creat() — Create a New File or Rewrite an Existing One” on page 146
- “dup() — Duplicate an Open File Descriptor” on page 180
- “exec Functions” on page 192
- “fcntl() — Control Open File Descriptors” on page 207
- “fsync() — Write Changes to Direct-Access Storage” on page 299
- “lseek() — Change the Offset of a File” on page 469
- “read() — Read From a File ” on page 628
- “stat() — Get File Information” on page 803

- “umask() — Set or Display the File Mode Creation Mask” on page 902
- “write() — Write Data on a File or Socket” on page 975

opendir() — Open a Directory

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

Format

```
#define _OPEN_SYS
#include <dirent.h>

DIR *opendir(const char *dirname);
```

General Description

Opens a directory so that it can be read with readdir(). The *dirname* string gives the name of the directory you want to open. The first readdir() call reads the first entry in the directory.

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 values under the conditions described:

EACCES
The process does not have permission to search some component of *dirname*, or it does not have read permission on the directory itself.

EINVAL
The byte file system (BFS) pathname starts with "//", which is not allowed for BFS files.

ELOOP
A loop exists in the symbolic links. The opendir() function issues this error if it detects more than POSIX_SYMLOOP (defined in the limits.h header file) symbolic links during resolution of the *dirname* argument.

EMFILE
The process has too many other file descriptors already 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, 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.

ENOENT
The directory *dirname* does not exist.

ENOTDIR
Some component of the *dirname* path name is not a directory.

Example

CBC3BO01

```
/* CBC3BO01
   This example opens a directory.
 */
#define _OPEN_SYS
#include <dirent.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/types.h>
#undef _OPEN_SYS
#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 23
- “sys/types.h” on page 46
- “closedir() — Close a Directory” on page 129
- “readdir() — Read an Entry from a Directory” on page 632
- “rewinddir() — Reposition a Directory Stream to the Beginning” on page 672

pathconf() — Determine Configurable Path Name Variables

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

Format

```
#define _OPEN_SYS
#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). The pathconf() function 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, pathconf() returns EINVAL, indicating that 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. For details, see the callable service tcsetattr(). The *pathname* argument must refer to a character special file for a terminal.

_PC_ACL
Returns 1 if an access control mechanism is supported by the security product.

_PC_ACL_ENTRIES_MAX
Returns the maximum number of ACL entries in an ACL for a file or directory that supports ACLs.

Returned Value

If successful, pathconf() returns the value of the variable requested in *varcode*.

If unsuccessful, pathconf() returns a value of -1. If a particular variable has no limit, such as PATH_MAX, pathconf() returns a value of -1 but does not change errno.

If pathconf() cannot determine an appropriate value, it sets errno to one of the following values under the conditions described:

EACCES
The process does not have search permission on some component of the *pathname*.

EINVAL
varcode is not a valid variable code, or the given variable cannot be associated with the specified file.

- If *varcode* refers to MAX_CANON, MAX_INPUT, or _POSIX_VDISABLE, and *pathname* does not refer to a character special file, pathconf() returns a value of -1 and sets errno to EINVAL.
- If *varcode* refers to NAME_MAX, PATH_MAX, or POSIX_NO_TRUNC, and *pathname* does not refer to a directory, pathconf() returns the requested information.
- If *varcode* refers to PC_PIPE_BUF and *pathname* refers to a pipe or a FIFO, the value returned applies to the referenced object itself. If *pathname* refers to a directory, the value returned applies to any FIFOs that exist or can be created within the directory. If *pathname* refers to any other type of file, the function sets errno to EINVAL.
- The path name starts with "//", which is not allowed for BFS files.

ELOOP
A loop exists in symbolic links. The pathconf() function issues this error if it detects more than POSIX_SYMLOOPS symbolic links in the resolution of *pathname*.

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
No file named *pathname* exists, 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 _OPEN_SYS
#include <errno.h>
#include <unistd.h>
#undef _OPEN_SYS
#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 51
- “fpathconf() — Determine Configurable Path Name Variables” on page 261

pause() — Suspend a Thread Pending a Signal

Standards / Extensions	C or C++	Dependencies
POSIX.1 POSIX.4a XPG4 XPG4.2	both	POSIX(ON)

Format

```
#define _OPEN_SYS
#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*. For details, see “sigprocmask() — Examine or Change a Thread” on page 773.

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.

The pause() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

If successful, pause() always returns a value of -1 and sets errno to EINTR.

Example

CBC3BP02

```
/* CBC3BP02
   This example suspends execution and determines the current time.
 */
#define _OPEN_SYS
#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;
  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 51
- “alarm() — Set an Alarm” on page 76
- “kill() — Send a Signal to a Process” on page 439
- “raise() — Raise Signal” on page 623
- “sigaction() — Examine or Change a Signal Action” on page 749
- “signal() — Handle Interrupts” on page 767
- “sigprocmask() — Examine or Change a Thread” on page 773
- “sigsuspend() — Change Mask and Suspend the Thread” on page 781
- “wait() — Wait for a Child Process to End” on page 927

pclose() — Close a Pipe Stream to or from a Process

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	POSIX(ON)

Format

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

int pclose(FILE *stream);
```

General Description

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. If a call made the termination status unavailable to pclose(), then pclose() returns a value of -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, pclose() will return a termination status 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 *stream* argument to pclose() is not a pointer to a stream created by popen(), the returned termination status will be -1.

Note: You can execute the pclose() function from any thread within the parent process.

Returned Value

If successful, pclose() returns the termination status of the shell command.

If unsuccessful, pclose() returns a value of -1 and sets errno to the following value to indicate the error:

Error Code
Description

ECHILD
The status of the child process could not be obtained.

Related Information

- “stdio.h” on page 40
- “popen() — Initiate a Pipe Stream to or from a Process” on page 541
- “waitpid() — Wait for a Specific Child Process to End” on page 930

perror() — Print Error Message

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 does not point to a null character, perror() prints *string* to the standard error stream, followed by a colon and a space. It then prints the message associated with the value in errno, 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 errno argument.

To produce accurate results, you should call perror() 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.

When the _EDC_ADD_ERRNO2 environment variable is set to 1, it will append the current errno2 value to the end of the perror() string as shown.

```
EDC5121I Invalid argument. (errno2=0x0C0F8402)
```

Returned Value

The perror() function has no returned 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)
{
   ins s;

   if ((s = socket (AF_INET,SOCK_STREAM,0)) <0)
   {
      perror("Could not open socket");
      exit(-1);
   }
}
```

Related Information

- "stdio.h" on page 40
- "strerror() — Get Pointer to Runtime Error Message" on page 817

pipe() — Create an Unnamed Pipe

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

Format

```
#define _OPEN_SYS
#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. The *fdinfo*[2] argument points to a memory area where pipe() can store two file descriptors. The pipe() function 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(). For details, see "fcntl() — Control Open File Descriptors" on page 207.

If pipe() successfully creates a pipe, it updates the access, change, and modification times for the pipe.

The pipe() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, pipe() returns a value of 0.

If unsuccessful, pipe() returns a value of -1 and sets errno to one of the following values under the conditions described:

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 _OPEN_SYS
#include <unistd.h>
#include <stdio.h>
#include <spawn.h>

main() {
  char original[]="This is the original string";
  char buf[80];
  int  p1[2], p2[2];
  int  fd_count, i;
  int  fd_map[5];

  if (pipe(p1) != 0)
    perror("first pipe() failed");
  else if (pipe(p2) != 0)
    perror("second pipe() failed");
  else {
    for (i = 0; i < 3; i++)
      fd_map[i] = i;
    fd_map[3] = p1[0];
    fd_map[4] = p2[1];
    fd_count = 5;
    if (spawn("CBC3BP0A", fd_count,fd_map, 0, 0, 0) == -1)
      perror("spawn() error");
    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);
    }
  }
}


/* CBC3BP0A
   This is the child.
 */
#define _OPEN_SYS
#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 buf[80];

  if (read(3, buf, sizeof(buf)) == -1)
    perror("read() error in parent");
  else {
    reverse(buf);
    if (write(4, buf, strlen(buf)+1) == -1)
      perror("write() error in child");
  }
  exit(0);
}
```

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 51
- "close() — Close a File" on page 127
- "fcntl() — Control Open File Descriptors" on page 207
- "open() — Open a File" on page 527
- "read() — Read From a File " on page 628
- "write() — Write Data on a File or Socket" on page 975

popen() — Initiate a Pipe Stream to or from a Process

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

Executes the command specified by *command* and creates a pipe between the calling program and the executed command, returning a pointer to a stream for reading from or writing to the pipe.

The environment of the executed command is the same as a child process created within the popen() call using spawn().

```
    spawn("command", fd_cnt, fd_map, &inherit, *args, *env);
```

where the path argument to *spawn()* is the command specified in the *popen()* argument.

To allow the caller to control whether the spawned child process runs in a separate address space from the parent, the spawn service allows for the specification of the _BPX_SHAREAS environmental variable. The accepted values for _BPX_SHAREAS and the actions taken for each value are as follows:

- _BPX_SHAREAS = YES -- Indicates that the child process should 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 is a set-user-ID or set-group-ID program that will cause the effective user-ID or group-ID of the child process to be different than that of the parent process.
  - If the specified file name represents an external link or a sticky bit file. However, if the program is a shell script and _BPX_SPAWN_SHELL = YES is set, then the process will run in the same address space.
  - If the parent's address space lacks the necessary resources to create another process with the address space.
- _BPX_SHAREAS = NO -- Indicates that the child process should run in a separate space from the parent's address space. This is the default behavior for the spawn service if the _BPX_SHAREAS environmental variable is not specified or contains an unsupported value.

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:

- If *mode* is `r`, the file descriptor STDOUT_FILENO will be the writeable end of the pipe when the child process starts. The file descriptor *fileno(stream)* in the calling process, where *stream* is the pointer returned by popen(), will be the readable end of the pipe.
- If *mode* is `w`, the file descriptor STDIN_FILENO will be the readable end of the pipe when the child process starts. The file descriptor *fileno(stream)* in the calling process, where *stream* is the pointer returned by popen(), will be the writeable end of the pipe.
- If *mode* is any other value, popen() returns a null pointer and sets errno 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.
- Before popen(), buffered reading may leave the standard input of that filter mispositioned. You may prevent similar problems with an output filter by buffer flushing with fflush().
- If you open a stream with popen() you should close it with pclose().
- The behavior of popen() is specified for *mode* values of `r` and `w`; *mode* values of `rb` and `wb` are supported but are not portable.
- If pclose() cannot execute the shell command, it will return a child termination status 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 open that was created with popen(), it must ensure that *pid* does not refer to the process started by popen().

Rule: You cannot call the popen() function after you have called pthread_create().

Returned Value

If successful, popen() returns a pointer to an open stream for reading or writing to a pipe.

If unsuccessful, open() returns a null pointer and sets errno to the following value:

EINVAL
The *mode* argument is invalid.

The popen() function may also set errno as described by spawn() or pipe().

Related Information

- “stdio.h” on page 40
- “fflush() — Write Buffer to File” on page 233

- "fork() — Create a New Process" on page 259
- "pclose() — Close a Pipe Stream to or from a Process" on page 537
- "pipe() — Create an Unnamed Pipe" on page 539
- "system() — Execute a Command" on page 858

pow() — Raise to Power

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

Format

```
#include <math.h>

double pow(double x, double y);
```

General Description

Calculates the value of *x* to the power of *y*.

Note: These functions work in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

Returned Value

If *y* is 0, pow() returns a value of 1. If *x* is 0 and *y* is negative, pow() sets errno to EDOM and returns a value of 0. If both *x* and *y* are 0, or if *x* is negative and *y* is not an integer, pow() sets errno to EDOM, and returns a value of 0. If the correct value is outside the range of representable values, pow() returns plus or minus HUGE_VAL, according to the sign of the value, and stores the value of the macro ERANGE in errno.

Special Behavior for IEEE

If successful, the function returns the value of *x* to the power of *y*. If *x* is negative or 0, then the *y* parameter must be an integer. If *y* is 0, the function returns 1.0 for all *x* parameters.

If an overflow occurs, the function returns HUGE_VAL and sets errno to ERANGE. If *x* is negative and *y* is not an integer, the function returns NaNQ and sets errno to EDOM. If both *x* and *y* are negative, the function returns NaNQ and sets errno to EDOM. If *x* is 0 and *y* is negative, the function returns HUGE_VAL but does not modify 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 32
- “exp(), expf(), expl()— Calculate Exponential Function” on page 199
- “log(), logf(), logl() — Calculate Natural Logarithm” on page 460
- “log10(), log10f(), log10l() — Calculate Base 10 Logarithm” on page 462

printf() — Format and Write Data

The information for this function is included in “fprintf() - printf() - sprintf() — Format and Write Data” on page 264.

pthread_attr_destroy() — Destroy the Thread Attributes Object

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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().

A thread attribute object manages the characteristics of threads in an application by defining the set of values 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.

The pthread_attr_destroy() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

If successful, pthread_attr_destroy() returns a value of 0.

If unsuccessful, pthread_attr_destroy() returns a value of -1.

No errnos are defined for pthread_attr_destroy(). You should 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 36
- “pthread_attr_init() — Initialize a Thread Attribute Object” on page 549
- “pthread_create() — Create a Thread” on page 573

pthread_attr_getdetachstate() — Get the Detach State Attribute

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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().

A thread attribute object manages the characteristics of threads in an application by defining the set of values 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.

The pthread_attr_getdetachstate() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

If successful, pthread_attr_getdetachstate() returns the detachstate (0 or 1).

If unsuccessful, pthread_attr_getdetachstate() returns a value of -1.

No errnos are defined for pthread_attr_getdetachstate(). You should 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 36
- “pthread_attr_init() — Initialize a Thread Attribute Object” on page 549

- “pthread_attr_setdetachstate() — Set the Detach State Attribute Object” on page 551
- “pthread_create() — Create a Thread” on page 573

pthread_attr_getstacksize() — Get the Thread Attribute Stacksize

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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(). The pthread_attr_getstacksize() function returns the value in the variable pointed to by *stacksize*.

A thread attribute object manages the characteristics of threads in an application by defining 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.

The pthread_attr_getstacksize() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Note: An XPLINK application uses two stacks, an upward-growing stack and a downward-growing stack. `stacksize` always refers to the size of the upward-growing stack. The size of the downward-growing stack is inherited from the THREADSTACK runtime option.

Returned Value

If successful, pthread_attr_getstacksize() returns a value of `0` and stores the `stacksize` attribute value in *stacksize*.

If unsuccessful, pthread_attr_getstacksize() returns a value of -1.

No errnos are defined for pthread_attr_getstacksize(). You should 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_getstackstate()");
      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 36
- "pthread_attr_init() — Initialize a Thread Attribute Object" on page 549
- "pthread_attr_setstacksize() — Set the Stacksize Attribute Object" on page 552
- "pthread_create() — Create a Thread" on page 573

pthread_attr_getweight_np() — Get Weight of Thread Attribute Object

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	POSIX(ON)

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; but because POSIX threads are mapped directly to CMS threads, these attributes have no effect.

__MEDIUM_WEIGHT
Medium-weight thread attribute.

__HEAVY_WEIGHT
Heavy-weight thread attribute.

A thread attribute object manages the characteristics of threads in an application by defining 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.

The pthread_attr_getweight_np() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, pthread_attr_getweight_np() returns the value of the weight of the thread attribute.

If unsuccessful, pthread_attr_getweight_np() returns a value of -1.

No errnos are defined for pthread_attr_getweight_np(). You should 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 36
- “pthread_attr_init() — Initialize a Thread Attribute Object” on page 549
- “pthread_attr_setweight_np() — Set Weight of Thread Attribute Object” on page 554
- “pthread_create() — Create a Thread” on page 573

pthread_attr_init() — Initialize a Thread Attribute Object

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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 as follows:

stacksize
Inherited from the STACK runtime option

detachstate
Undetached

synch
Synchronous

weight
Heavy

A thread attribute object manages the characteristics of threads in an application by defining 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.

The pthread_attr_init() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Note: An XPLINK application uses two stacks, an upward-growing stack and a downward-growing stack. "stacksize" always refers to the size of the upward-growing stack. The size of the downward-growing stack is inherited from the THREADSTACK runtime option.

Returned Value

If successful, pthread_attr_init() returns a value of 0.

If unsuccessful, pthread_attr_init() returns a value of -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()
{
   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 36
- "pthread_attr_destroy() — Destroy the Thread Attributes Object" on page 544
- "pthread_create() — Create a Thread" on page 573

pthread_attr_setdetachstate() — Set the Detach State Attribute Object

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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().

A thread attribute object manages the characteristics of threads in an application by defining 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.

The pthread_attr_setdetachstate() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, pthread_attr_setdetachstate() returns a value of `0`.

If unsuccessful, pthread_attr_setdetachstate() returns a value of -1.

No errnos are defined for pthread_attr_setdetachstate(). You should 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         s1;
   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 36
- “pthread_attr_getdetachstate() — Get the Detach State Attribute” on page 545
- “pthread_attr_init() — Initialize a Thread Attribute Object” on page 549
- “pthread_create() — Create a Thread” on page 573

pthread_attr_setstacksize() — Set the Stacksize Attribute Object

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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.

A thread attribute object manages the characteristics of threads in an 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.

The pthread_attr_setstacksize() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

If successful, pthread_attr_setstacksize() returns a value of 0.

If unsuccessful, pthread_attr_setstacksize() returns a value of -1.

No errnos are defined for pthread_attr_setstacksize(). You should 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         s1;
   pthread_attr_t attr;
   pthread_t      thid;

   rc = pthread_attr_init(&attr);
   if (rc == -1) {
      perror("error in pthread_attr_init");
      exit(1);
   }

   s1 = 4096;
   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 36
- “pthread_attr_getstacksize() — Get the Thread Attribute Stacksize” on page 547
- “pthread_attr_init() — Initialize a Thread Attribute Object” on page 549
- “pthread_create() — Create a Thread” on page 573

pthread_attr_setweight_np() — Set Weight of Thread Attribute Object

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_attr_setweight_np(pthread_attr_t *attr, int threadweight);
```

General Description

Alters the current weight of the thread setting of the thread attribute object, *attr*.

You can set *threadweight* to one of the following two symbols for the weight of the thread, as defined in the pthread.h header file. Because POSIX threads are mapped directly to CMS threads, these attributes have no effect.

__MEDIUM_WEIGHT
Medium-weight thread attribute.

__HEAVY_WEIGHT
Heavy-weight thread attribute.

A thread attribute object manages the characteristics of threads in your application by defining 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.

The pthread_attr_setweight-np() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

If successful, pthread_attr_setweight-np() returns a value of `0`.

If unsuccessful, pthread_attr_setweight-np() returns a value of `-1`.

No errnos are defined for pthread_attr_setweight-np(). You should 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 36
- "pthread_attr_getweight_np() — Get Weight of Thread Attribute Object" on page 548
- "pthread_attr_init() — Initialize a Thread Attribute Object" on page 549
- "pthread_create() — Create a Thread" on page 573

pthread_cancel() — Cancel a Thread

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

General Description

Requests that a thread be canceled. The thread controls when the 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: closef(), 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 cancelled thread has a return value of -1. For more information, see "pthread_join() — Wait for a Thread to End" on page 581.

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 pthread_cancel() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Special Behavior for C++

Destructors for automatic objects on the stack will be run when a thread is canceled. The stack is unwound and the destructors are run in reverse order.

Returned Value

If successful, pthread_cancel() returns a value of 0, indicating 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 a value of -1 and sets errno to one of the following values under the conditions described:

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 36
- "pthread_cond_timedwait() — Wait on a Condition Variable" on page 564
- "pthread_cond_wait() — Wait on a Condition Variable" on page 566
- "pthread_exit() — Exit a Thread" on page 577
- "pthread_join() — Wait for a Thread to End" on page 581
- "pthread_setintr() — Set the Cancelability State of a Thread" on page 604
- "pthread_setintrtype() — Set the Cancelability Type of a Thread" on page 606
- "pthread_testintr() — Establish a Cancelability Point" on page 610

pthread_cleanup_pop() — Remove a Cleanup Handler

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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.

The pthread_cleanup_push() and pthread_cleanup_pop() functions 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().

This function is supported only in a POSIX program. See "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7 for more information.

Returned Value

The pthread_cleanup_pop() function has no returned value. The function is used as a statement. If an error occurs while a pthread_cleanup_pop() statement is being processed, a termination condition is raised.

No errnos are defined for pthread_cleanup_pop(). You should 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(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 36
- “pthread_cancel() — Cancel a Thread” on page 555
- “pthread_cleanup_push() — Establish a Cleanup Handler” on page 558
- “pthread_exit() — Exit a Thread” on page 577

pthread_cleanup_push() — Establish a Cleanup Handler

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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().

The pthread_cleanup_push() and pthread_cleanup_pop() functions must appear in pairs and within the same lexical scope, or undefined behavior will result.

The pthread_cleanup_push() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

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

The pthread_cleanup_push() function has no returned value. The function is used as a statement. If an error occurs while a pthread_cleanup_push() statement is being processed, a termination condition is raised.

No errnos are defined for pthread_cleanup_push(). You should 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 36
- “pthread_cancel() — Cancel a Thread” on page 555
- “pthread_cleanup_pop() — Remove a Cleanup Handler” on page 557
- “pthread_exit() — Exit a Thread” on page 577

pthread_cond_broadcast() — Broadcast a Condition

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
```

General Description

Unblocks 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.

The pthread_cond_broadcast() function has no effect if no threads are currently blocked on *cond*.

The pthread_cond_broadcast() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

If successful, pthread_cond_broadcast() returns a value of 0.

If unsuccessful, pthread_cond_broadcast() returns a value of -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 36
- “pthread_cond_init() — Initialize a Condition Variable” on page 562
- “pthread_cond_signal() — Signal a Condition” on page 563
- “pthread_cond_timedwait() — Wait on a Condition Variable” on page 564
- “pthread_cond_wait() — Wait on a Condition Variable” on page 566

pthread_cond_destroy() — Destroy the Condition Variable Object

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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, which is used in conjunction with mutexes to protect shared resources.

The pthread_cond_destroy() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, pthread_cond_destroy() returns a value of 0.

If unsuccessful, pthread_cond_destroy() returns a value of -1 and sets errno to one of the following values under the conditions described:

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 36
- "pthread_cond_broadcast() — Broadcast a Condition" on page 560
- "pthread_cond_init() — Initialize a Condition Variable" on page 562
- "pthread_cond_signal() — Signal a Condition" on page 563
- "pthread_cond_timedwait() — Wait on a Condition Variable" on page 564
- "pthread_cond_wait() — Wait on a Condition Variable" on page 566

pthread_cond_init() — Initialize a Condition Variable

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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.

The pthread_cond_init() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, pthread_cond_init() returns a value of 0.

If unsuccessful, pthread_cond_init() returns a value of -1 and sets errno to ENOMEM, which indicates that not enough memory is available 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 36
- "pthread_condattr_init() — Initialize a Condition Attribute Object" on page 571
- "pthread_cond_broadcast() — Broadcast a Condition" on page 560
- "pthread_cond_signal() — Signal a Condition" on page 563
- "pthread_cond_timedwait() — Wait on a Condition Variable" on page 564
- "pthread_cond_wait() — Wait on a Condition Variable" on page 566

pthread_cond_signal() — Signal a Condition

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t *cond);
```

General Description

Unblocks 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.

The pthread_cond_signal() function will have no effect if no threads are currently blocked on *cond*.

The pthread_cond_signal() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, pthread_cond_signal() returns a value of 0.

If unsuccessful, pthread_cond_signal() returns a value of -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 36
- "pthread_cond_broadcast() — Broadcast a Condition" on page 560
- "pthread_cond_init() — Initialize a Condition Variable" on page 562
- "pthread_cond_timedwait() — Wait on a Condition Variable" on page 564
- "pthread_cond_wait() — Wait on a Condition Variable" on page 566

pthread_cond_timedwait() — Wait on a Condition Variable

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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. The pthread_cond_timedwait() function 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 pthread_cond_timedwait() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

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 a value of `0`.

If unsuccessful, pthread_cond_timedwait() returns a value of -1 and sets errno to one of the following values under the conditions described:

EAGAIN
The time specified by *abstime* has passed.

EINVAL
Can be one of the following error conditions:

- The value specified by *cond* is not valid.
- The value specified by *mutex* is not valid.
- The value specified by *abstime* (`tv_sec`) is not valid.
- The value specified by *abstime* (`tv_nsec`) is not valid.
- Different mutexes were specified for concurrent operations on the same condition variable.
- The mutex is not owned by the current thread.

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;
  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 36
- “pthread_cond_broadcast() — Broadcast a Condition” on page 560
- “pthread_cond_signal() — Signal a Condition” on page 563
- “pthread_cond_wait() — Wait on a Condition Variable” on page 566
- “pthread_mutex_init() — Initialize a Mutex Object” on page 589

pthread_cond_wait() — Wait on a Condition Variable

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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. You must call pthread_cond_wait() with *mutex* locked by the calling thread, or undefined behavior will result. A mutex is locked using pthread_mutex_lock().

The condition variable, *cond*, 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 560 and to "pthread_cond_signal() — Signal a Condition" on page 563.

The pthread_cond_wait() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, pthread_cond_wait() returns a value of 0.

If unsuccessful, pthread_cond_wait() returns a value of -1 and sets errno to EINVAL, which can indicate the following:

- 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 36
- “pthread_cond_broadcast() — Broadcast a Condition” on page 560
- “pthread_cond_signal() — Signal a Condition” on page 563
- “pthread_cond_timedwait() — Wait on a Condition Variable” on page 564

pthread_condattr_destroy() — Destroy Condition Variable Attribute Object

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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.

You use the pthread_condattr_init function to define a condition variable attribute object. You use the pthread_condattr_destroy() function 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, see “pthread_cond_init() — Initialize a Condition Variable” on page 562.

The pthread_condattr_destroy() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7

Returned Value

If successful, pthread_condattr_destroy() returns a value of 0.

If unsuccessful, pthread_condattr_destroy() returns a value of -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 36
- "pthread_condattr_init() — Initialize a Condition Attribute Object" on page 571
- "pthread_cond_init() — Initialize a Condition Variable" on page 562
- "pthread_mutex_init() — Initialize a Mutex Object" on page 589

pthread_condattr_getkind_np() — Get Kind Attribute from a Condition Variable Attribute Object

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	POSIX(ON)

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.

The pthread_condattr_getkind_np() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

If successful, pthread_condattr_getkind_np() returns a value of `0`.

If unsuccessful, pthread_condattr_getkind_np() returns a value of `-1` and sets errno to EINVAL, which indicates that the value specified for *attr* is not valid.

Example

CBC3BP24

```
/* CBC3BP24 */
#pragma runopts(TEST(ALL))

#ifndef _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(&attr, __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_no() unrecognized");

    }

   exit(0);
}
```

Related Information

- “pthread.h” on page 36
- “pthread_condattr_init() — Initialize a Condition Attribute Object” on page 571

- "pthread_condattr_setkind_np() — Set Kind Attribute from a Condition Variable Attribute Object" on page 572

pthread_condattr_init() — Initialize a Condition Attribute Object

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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.

You use the pthread_condattr_init function to define a condition variable attribute object. You use the pthread_condattr_destroy() function 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, see "pthread_cond_init() — Initialize a Condition Variable" on page 562.

The pthread_condattr_init function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, pthread_condattr_init returns a value of 0.

If unsuccessful, pthread_condattr_init returns a value of -1 and sets errno to ENOMEM, which indicates that not enough memory is available 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) {
    perror("pthread_condattr_destroy() error");
```

```
    exit(2);
  }
}
```

Related Information

- "pthread.h" on page 36
- "pthread_cond_init() — Initialize a Condition Variable" on page 562
- "pthread_mutex_init() — Initialize a Mutex Object" on page 589

pthread_condattr_setkind_np() — Set Kind Attribute from a Condition Variable Attribute Object

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	POSIX(ON)

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.

The pthread_condattr_setkind_np() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, pthread_condattr_setkind_np() returns a value of 0.

If unsuccessful, pthread_condattr_setkind_np() returns a value of -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))

#ifndef _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(&attr, __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_no() unrecognized");
    }

   exit(0);
}
```

Related Information

- “pthread.h” on page 36
- “pthread_condattr_init() — Initialize a Condition Attribute Object” on page 571
- “pthread_condattr_getkind_np() — Get Kind Attribute from a Condition Variable Attribute Object” on page 569

pthread_create() — Create a Thread

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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. For a description of the thread attributes and their defaults, see “pthread_attr_init() — Initialize a Thread Attribute Object” on page 549. 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 specified by *thread* are undefined.

The pthread_create() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

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 a value of 0.

If unsuccessful, pthread_create() returns a value of -1 and sets errno to one of the following values under the conditions described:

EAGAIN
The system lacks the necessary resources to create another thread.

ENOMEM
Insufficient memory is available 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_create() 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 36
- "pthread_exit() — Exit a Thread" on page 577
- "pthread_join() — Wait for a Thread to End" on page 581

pthread_detach() — Detach a Thread

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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.

The pthread_detach() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, pthread_detach() returns a value of `0`.

If unsuccessful, pthread_detach() returns a value of -1 and sets errno to one of the following values under the conditions described:

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);
  }

  if (pthread_join(thid, &ret) != 0) {
    perror("pthread_create() 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 36
- “pthread_join() — Wait for a Thread to End” on page 581

pthread_equal() — Compare Thread IDs

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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.

The pthread_equal() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

If *t1* and *t2* are equal, pthread_equal() returns a positive value. Otherwise, pthread_equal() returns a value of 0. If *t1* or *t2* are not valid thread IDs, the behavior is undefined.

If unsuccessful, pthread_equal() returns a value of -1.

No errnos are defined for pthread_equal(). You should 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_create() error");
    exit(3);
  }
}
```

Output

```
the thread is not the IPT
```

Related Information

- “pthread.h” on page 36
- “pthread_create() — Create a Thread” on page 573
- “pthread_self() — Get the Caller” on page 603

pthread_exit() — Exit a Thread

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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, see “pthread_cleanup_pop() — Remove a Cleanup Handler” on page 557 and “pthread_cleanup_push() — Establish a Cleanup Handler” on page 558.
- For details on the destructor routine, see “pthread_key_create() — Create Thread-Specific Data Key” on page 584.

The pthread_exit() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Special Behavior for C++

Destructors for automatic objects on the stack will be run when a thread is canceled. The stack is unwound and the destructors are run in reverse order.

Returned Value

The pthread_exit() function has no returned value.

No errnos are defined for pthread_exit(). You should 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;

  if (pthread_create(&thid, NULL, thread, NULL) != 0) {
    perror("pthread_create() error");
    exit(1);
  }

  if (pthread_join(thid, &ret) != 0) {
    perror("pthread_create() error");
    exit(3);
  }

  printf("thread exited with '%s'\n", ret);
}
```

Output

```
thread exited with 'This is a test'
```

Related Information

- "pthread.h" on page 36
- "pthread_cleanup_pop() — Remove a Cleanup Handler" on page 557
- "pthread_cleanup_push() — Establish a Cleanup Handler" on page 558
- "pthread_create() — Create a Thread" on page 573
- "pthread_join() — Wait for a Thread to End" on page 581
- "pthread_key_create() — Create Thread-Specific Data Key" on page 584

pthread_getspecific() — Get the Thread-Specific Value for a Key

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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().

The pthread_getspecific() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If unsuccessful, pthread_getspecific() sets errno to EINVAL, which indicates that the value for *key* is not valid.

Example

CBC3BP31

```
/* CBC3BP31 */
#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 *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 36
- "pthread_key_create() — Create Thread-Specific Data Key" on page 584
- "pthread_setspecific() — Set the Thread-Specific Value for a Key" on page 608

pthread_join() — Wait for a Thread to End

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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 a value of -1.

The pthread_join() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, pthread_join() returns a value of `0`.

If unsuccessful, pthread_join() returns a value of -1 and sets errno to one of the following values under the conditions described:

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() returns successfully, the target thread has been detached.

- 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.
- 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_create() 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 36
- “pthread_create() — Create a Thread” on page 573
- “pthread_cond_wait() — Wait on a Condition Variable” on page 566
- “pthread_detach() — Detach a Thread” on page 575

pthread_join_d4_np() — Wait for a Thread to End (.4a)

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	POSIX(ON)

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 a value of -1.

The pthread_join_d4_np() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, pthread_join_d4_np() returns a value of 0.

If unsuccessful, pthread_join_d4_np() returns a value of -1, and sets errno to one of the following values under the conditions described:

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_create() 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 36
- "pthread_create() — Create a Thread" on page 573
- "pthread_cond_wait() — Wait on a Condition Variable" on page 566
- "pthread_detach() — Detach a Thread" on page 575

pthread_key_create() — Create Thread-Specific Data Key

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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 can use that key and can set its own unique value by using 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().

The pthread_key_create() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

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 a value of 0 and stores the newly created key identifier in *key*.

If unsuccessful, pthread_key_create() returns a value of -1 and sets errno to one of the following values under the conditions described:

EAGAIN
Insufficient system resources are available to create another thread-specific data key, or the limit is exceeded for the total number of keys per process.

ENOMEM
Insufficient memory is available to create *key*.

Example

CBC3BP34

```
/* CBC3BP34 */
#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 *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 36
- “pthread_getspecific() — Get the Thread-Specific Value for a Key” on page 579
- “pthread_setspecific() — Set the Thread-Specific Value for a Key” on page 608

pthread_kill() — Send a Signal to a Thread

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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. (For a list of signals, see Table 24 on page 750.) 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.

The pthread_kill() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

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.

Usage Note

The SIGTHSTOP and SIGTHCONT signals can be issued by this function. *pthread_kill()* is the only function that can issue SIGTHSTOP or SIGTHCONT.

Returned Value

If successful, pthread_kill() returns a value of 0.

If unsuccessful, pthread_kill() returns a value of -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

```
#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 36
- “signal.h” on page 38
- “kill() — Send a Signal to a Process” on page 439
- “pthread_self() — Get the Caller” on page 603
- “raise() — Raise Signal” on page 623
- “sigaction() — Examine or Change a Signal Action” on page 749
- “signal() — Handle Interrupts” on page 767
- “sigprocmask() — Examine or Change a Thread” on page 773

pthread_mutex_destroy() — Delete a Mutex Object

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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().

The pthread_mutex_destroy() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

If successful, pthread_mutex_destroy() returns a value of `0`.

If unsuccessful, pthread_mutex_destroy() returns a value of -1 and sets errno to one of the following values under the conditions described:

EBUSY
An attempt was made to destroy the object referenced by *mutex* while it was locked or referenced as part of a wait on a condition variable.

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 36
- “pthread_cond_timedwait() — Wait on a Condition Variable” on page 564
- “pthread_cond_wait() — Wait on a Condition Variable” on page 566
- “pthread_mutex_init() — Initialize a Mutex Object” on page 589
- “pthread_mutex_lock() — Wait for a Lock on a Mutex Object” on page 590
- “pthread_mutex_trylock() — Attempt to Lock a Mutex Object” on page 592
- “pthread_mutex_unlock() — Unlock a Mutex Object” on page 593

pthread_mutex_init() — Initialize a Mutex Object

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
```

General Description

Creates a *mutex* with attributes specified by *attr*. If *attr* is null, the default mutex attribute (NONRECURSIVE) is used.

The pthread_mutex_init() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, pthread_mutex_init() returns a value of 0, and the state of the mutex becomes initialized and unlocked.

If unsuccessful, pthread_mutex_init() returns a value of -1 and sets errno to ENOMEM, which indicates that not enough memory is available to initialize the mutex.

Example

CBC3BP37

```
/* CBC3BP37 */
#ifndef _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 36
- "pthread_cond_init() — Initialize a Condition Variable" on page 562
- "pthread_cond_timedwait() — Wait on a Condition Variable" on page 564
- "pthread_cond_wait() — Wait on a Condition Variable" on page 566
- "pthread_mutexattr_init() — Initialize a Mutex Attribute Object" on page 598
- "pthread_mutex_lock() — Wait for a Lock on a Mutex Object" on page 590
- "pthread_mutex_trylock() — Attempt to Lock a Mutex Object" on page 592
- "pthread_mutex_unlock() — Unlock a Mutex Object" on page 593

pthread_mutex_lock() — Wait for a Lock on a Mutex Object

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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.

The pthread_mutex_lock() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

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 relock a mutex that it has already locked, it will return with an error. If a thread attempts to unlock 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 to 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 a value of `0`.

If unsuccessful, pthread_mutex_lock() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EINVAL
The value specified by *mutex* is not valid.

EDEADLK
The current thread already owns the mutex, and the mutex has a *kind* attribute of __MUTEX_NONRECURSIVE.

Example

CBC3BP38

```
/* CBC3BP38 */
#ifndef _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 36
- "pthread_cond_init() — Initialize a Condition Variable" on page 562
- "pthread_cond_wait() — Wait on a Condition Variable" on page 566
- "pthread_mutex_destroy() — Delete a Mutex Object" on page 588
- "pthread_mutex_init() — Initialize a Mutex Object" on page 589

pthread_mutex_trylock() — Attempt to Lock a Mutex Object

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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 the thread must call pthread_mutex_unlock() to release the mutex (it has the same behavior as a pthread_mutex_lock()).

The pthread_mutex_trylock() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, pthread_mutex_trylock() returns a value of `0`.

If unsuccessful, pthread_mutex_trylock() returns a value of -1 and sets errno to one of the following values under the conditions described:

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_mutex_trylock() error");
    exit(5);
  }
}
```

Output

```
IPT was granted the mutex
thread was denied access to the mutex
```

Related Information

- “pthread.h” on page 36
- “pthread_mutex_destroy() — Delete a Mutex Object” on page 588
- “pthread_mutex_init() — Initialize a Mutex Object” on page 589

pthread_mutex_unlock() — Unlock a Mutex Object

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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.

The pthread_mutex_unlock() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

If successful, pthread_mutex_unlock() returns a value of 0.

If unsuccessful, pthread_mutex_unlock() returns a value of -1 and sets errno to EINVAL, which indicates that the value specified by *mutex* is not valid.

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_mutex_lock() error");
    exit(7);
  }
}
```

Output

```
IPT was granted the mutex
thread was granted the mutex
```

Related Information

- “pthread.h” on page 36
- “pthread_mutex_destroy() — Delete a Mutex Object” on page 588
- “pthread_mutex_init() — Initialize a Mutex Object” on page 589
- “pthread_mutex_lock() — Wait for a Lock on a Mutex Object” on page 590

pthread_mutexattr_destroy() — Destroy a Mutex Attribute Object

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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. A mutex attribute object 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. You use pthread_mutexattr_init() to define a mutex attribute object.

The pthread_mutexattr_destroy() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

If successful, pthread_mutexattr_destroy() returns a value of 0.

If unsuccessful, pthread_mutexattr_destroy() returns a value of -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(&attr, __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 36
- “pthread_cond_init() — Initialize a Condition Variable” on page 562
- “pthread_create() — Create a Thread” on page 573
- “pthread_mutexattr_init() — Initialize a Mutex Attribute Object” on page 598

pthread_mutexattr_getkind_np() — Get Kind from a Mutex Attribute Object

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	POSIX(ON)

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. A mutex attribute object 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.

The values for the attribute *kind* are as follows:

__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.

The pthread_mutexattr_getkind_np() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

If successful, pthread_mutexattr_getkind_np() returns a value of `0`.

If unsuccessful, pthread_mutexattr_getkind_np() returns a value of -1 and sets errno to EINVAL, which indicates that the value specified for *attr* is not valid.

Example

CBC3BP43

```
/* CBC3BP43 */
#pragma runopts(TEST(ALL))

#ifndef _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 36
- “pthread_mutexattr_init() — Initialize a Mutex Attribute Object” on page 598
- “pthread_mutexattr_setkind_np() — Set Kind for a Mutex Attribute Object” on page 599

pthread_mutexattr_init() — Initialize a Mutex Attribute Object

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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. A mutex attribute object 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.

For a valid mutex attribute, see “pthread_mutexattr_setkind_np() — Set Kind for a Mutex Attribute Object” on page 599.

The pthread_mutexattr_init() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

If successful, pthread_mutexattr_init() returns a value of `0`.

If unsuccessful, pthread_mutexattr_init() returns a value of -1 and sets errno to ENOMEM, which indicates that insufficient memory is available 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(&attr, __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 36
- "pthread_cond_init() — Initialize a Condition Variable" on page 562
- "pthread_create() — Create a Thread" on page 573
- "pthread_mutex_init() — Initialize a Mutex Object" on page 589

pthread_mutexattr_setkind_np() — Set Kind for a Mutex Attribute Object

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	POSIX(ON)

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. A mutex attribute object 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.

The valid values for the attribute *kind* are as follows:

__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.

The pthread_mutexattr_setkind_np() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

If successful, pthread_mutexattr_setkind_np() returns a value of 0.

If unsuccessful, pthread_mutexattr_setkind_np() returns a value of -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))

#ifndef _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 36
- "pthread_mutexattr_getkind_np() — Get Kind from a Mutex Attribute Object" on page 596
- "pthread_mutexattr_init() — Initialize a Mutex Attribute Object" on page 598

pthread_once() — Invoke a Function Once

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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 multiple threads call the function simultaneously. 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.

You use pthread_once() 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 and then pass as a parameter on the pthread_once() function call.

init_routine is a normal function and can be invoked directly outside of pthread_once(). The *once_control* variable 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.

The pthread_once() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, pthread_once() returns a value of `0`.

If unsuccessful, pthread_once() returns a value of `-1`.

No errnos are defined for pthread_once(). You should use perror() or strerror() to determine the cause of the error.

Example

CBC3BP46

```
/* CBC3BP46 */
#ifndef _OPEN_THREADS
#define _OPEN_THREADS
#endif

#include <stdio.h>
#include <errno.h>
#include <pthread.h>

#define threads 3

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 36

pthread_self() — Get the Caller

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

Format

```
#define _OPEN_THREADS
#include <pthread.h>

pthread_t pthread_self(void);
```

General Description

Returns the thread ID of the calling thread.

The pthread_self() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

No errnos are defined for pthread_self(). You should 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_create() 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 36
- "pthread_create() — Create a Thread" on page 573
- "pthread_equal() — Compare Thread IDs" on page 576

pthread_setintr() — Set the Cancelability State of a Thread

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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. For information on the cancelability types, see "pthread_setintrtype() — Set the Cancelability Type of a Thread" on page 606.

The pthread_setintr() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, pthread_setintr() returns the previous state.

If unsuccessful, pthread_setintr() returns a value of -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 36
- “pthread_cancel() — Cancel a Thread” on page 555
- “pthread_setintrtype() — Set the Cancelability Type of a Thread” on page 606

- "pthread_testintr() — Establish a Cancelability Point" on page 610

pthread_setintrtype() — Set the Cancelability Type of a Thread

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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()

The pthread_setintrtype() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, pthread_setintrtype() returns the previous type.

If unsuccessful, pthread_setintrtype() returns a value of -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            *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 36
- “pthread_cancel() — Cancel a Thread” on page 555
- “pthread_setintr() — Set the Cancelability State of a Thread” on page 604
- “pthread_testintr() — Establish a Cancelability Point” on page 610

pthread_setspecific() — Set the Thread-Specific Value for a Key

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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(), which 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().

The pthread_setspecific() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

If successful, pthread_setspecific() returns a value of `0`.

If unsuccessful, pthread_setspecific() returns a value of -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 36
- “pthread_getspecific() — Get the Thread-Specific Value for a Key” on page 579
- “pthread_key_create() — Create Thread-Specific Data Key” on page 584

pthread_testintr() — Establish a Cancelability Point

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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 set the cancelability state to enabled (PTHREAD_INTR_ENABLE) for this function to have any effect.

The pthread_testintr() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

The pthread_testintr() function has no returned value.

No errnos are defined for pthread_testintr(). You should 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 *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            *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 36
- “pthread_cancel() — Cancel a Thread” on page 555
- “pthread_setintr() — Set the Cancelability State of a Thread” on page 604
- “pthread_setintrtype() — Set the Cancelability Type of a Thread” on page 606

pthread_yield() — Release the Processor to Other Threads

Standards / Extensions	C or C++	Dependencies
POSIX.4a	both	POSIX(ON)

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.

The pthread_yield() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

The pthread_yield() function has no returned value.

No errnos are defined for pthread_yield().

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);
  }

  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 36

putc() - putchar() — Write a Character

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

Format

```
#include <stdio.h>

int putc(int c, FILE *stream);
int putchar(int c);
```

General Description

Converts *c* to an `unsigned char` and then writes *c* to the output *stream* at the current position. The putchar() function is identical to:

```
putc(c, stdout);
```

Recommendation: These functions are also available as macros. For performance purposes, you should use the macro forms rather than the functional forms.

By default, if the stdio.h header file is included, the macro is invoked; therefore, *stream* should never be an expression with side effects.

You can access the actual function 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: `(putchar)('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.

The putc() and putchar() functions are not supported for files opened with `type=record`.

The putc() and putchar() functions 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition must also occur, unless an EOF has been reached.

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 40
- "fputc() — Write a Character" on page 272
- "fwrite() — Write Items" on page 308
- "getc() - getchar() — Read a Character" on page 312

putenv() — Change or Add an Environment Variable

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 *envvar* argument is a pointer to a null-terminated character string in the form:

name=value

where:

1. The *name* part is a character string that represents the environment variable; putenv() will use this name when it searches the array of environment variables to determine whether to add or change this environment variable.
2. The = part is a separator character; it cannot appear in the *name*.

3. The *value* part is a null-terminated character string that represents the value to which the environment variable, *name*, will be set.

The putenv() function is a simplified form of setenv() and is equivalent to

setenv(*&name*, *&value*, 1)

Note: On this implementation, the system does not add the storage pointed to by *envvar* to the array of environment variables. Instead, it copies the string into system allocated storage. You should assume the system will use user storage and should not define this environment variable with storage that is part of your automatic storage.

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 value of `0`.

If unsuccessful, putenv() returns a value of -1 and sets errno to one of the following values under the conditions described.

ENOMEM
Insufficient memory was available.

Special Behavior for OpenExtensions Services

EINVAL
The environment variable pointed to by *envvar* does not follow the prescribed format. The equal-sign (=) separating the environment variable name from the value was not found.

Related Information

- "stdlib.h" on page 42
- "clearenv() — Clear Environment Variables" on page 122
- "getenv() — Get Value of Environment Variables" on page 326
- "setenv() — Add, Delete, and Change Environment Variables" on page 708

puts() — Write a String

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. It does not write the terminating null character.

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, puts() fills the record with the output characters, sets the last character of

the record to a new-line character, and writes the remaining output characters to the next record. Such wrapping is repeated until the remaining output characters fit into the record.

Note: The puts() function appends the new-line character 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`.

The puts() function 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition must also occur, unless an EOF has been reached.

Returned Value

Returns EOF if an error occurs; otherwise, puts() returns the number of bytes written. However, puts() does not count new-line characters used to wrap the data.

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)
{
  if ( puts("Hello World") == EOF )
    printf( "Error in puts\n" );
}
```

Output

```
Hello World
```

Related Information

- “stdio.h” on page 40
- “fputs() — Write a String” on page 273
- “gets() — Read a String” on page 364

pututxline() — Write Entry to utmpx Database

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

Format

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

struct utmpx *pututxline(const struct utmpx *utmpx);
```

General Description

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 pututxline() replaces the entry; otherwise, it makes a new entry 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 is EMPTY in the entry being added, pututxline() always places it at the start of the `utmpx` database. For this reason, you should not use pututxline() to place EMPTY entries in the `utmpx` database.

The getutxent(), getutxid(), and getutxline() functions obtain a shared lock on the database. The pututxline() function 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 using the fcntl() command F_SETLKW on the first record in the database.

Because pututxline() returns thread-specific data, you can use it safely from a multithreaded 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 calls endutxent().

The name of the database file defaults to /etc/utmpx. To process a different database file name, you must use the __utmpxname() function.

The pututxline() function 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.
- The 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 entry. Other members of the entry will contain meaningful data based on the value of the `ut_type` member as follows:

EMPTY
No other members have meaningful data.

BOOT_TIME
`ut_tv` is meaningful.

__RUN_LVL
`ut_tv` and `ut_line` are meaningful.

OLD_TIME
`ut_tv` is meaningful.

NEW_TIME
`ut_tv` is meaningful.

USER_PROCESS
`ut_id`, `ut_user` (login name of the user), `ut_line`, `ut_pid`, and `ut_tv` are meaningful.

INIT_PROCESS
`ut_id`, `ut_pid`, and `ut_tv` are meaningful.

LOGIN_PROCESS
`ut_id`, `ut_user` (implementation-specific name of the login process), `ut_pid`, and `ut_tv` are meaningful.

DEAD_PROCESS
`ut_id`, `ut_pid`, and `ut_tv` are meaningful.

Returned Value

If successful, pututxline() returns a pointer to a `utmpx` structure, containing a copy of the entry written to the database.

If unsuccessful, pututxline() returns a null pointer.

The pututxline() function may fail if the process does not have appropriate privileges.

Related Information

- "endutxent() — Close the utmpx Database" on page 186
- "getutxent() — Read Next Entry in utmpx Database" on page 379
- "getutxid() — Search by ID utmpx Database" on page 380
- "getutxline() — Search by Line utmpx Database" on page 381
- "setutxent() — Reset to Start of utmpx Database" on page 741
- "__utmpxname() — Change the utmpx Database Name" on page 915

putw() — Put a Machine Word on a Stream

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

Writes the word *w* to the output *stream* at the position at which the file offset is pointing. The size of the word is the size of an `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 a value of `0`.

If unsuccessful, putw() returns a nonzero value, sets the error indicators for *stream*, and sets errno to indicate the error. See "fread() — Read Items" on page 277 for errno values.

Related Information

- "stdio.h" on page 40
- "fopen() — Open a File" on page 254
- "fwrite() — Write Items" on page 308
- "getw() — Get a Machine Word from a Stream" on page 383

putwc() — Output a Wide Character

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 you implement it 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.

The 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition must also occur, 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.

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, putwc() sets the error indicator for the stream and returns WEOF. If an encoding error occurs when converting from a wide character to a multibyte character, putwc() returns WEOF and stores the value of the macro EILSEQ in errno.

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 40
- “wchar.h” on page 53
- “fputwc() — Output a Wide Character” on page 274

putwchar() — Output a Wide Character to Standard Output

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

Format

Non-XPG4

```
#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: putwc()(*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.

The putwchar() function 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition must also occur, unless an EOF has been reached.

You may not use putwc() 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.

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, putwchar() returns WEOF and sets the error indicator for the stream. If an encoding error occurs when converting from a wide character to a multibyte character, putwchar() returns WEOF and stores the value of the macro EILSEQ in errno.

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 53
- "fputwc() — Output a Wide Character" on page 274

qsort() — Sort Array

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 specifies the array to be sorted. The qsort() function overwrites the contents of the array with the sorted elements.

The *compare* pointer specifies 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`
element1 less than *element2*

`0`
element1 equal to *element2*

Greater than `0`
element1 greater than *element2*

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

The qsort() function has 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 */
int compare(const void *, const void *);

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 42
- "bsearch() — Search Arrays" on page 100

raise() — Raise Signal

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 process that issued raise(). If the signal is not blocked, it is delivered to the sender before raise() returns. The signal sent is `int sig`.

You can use signal() to specify how a signal will be handled when raise() is invoked. In an OpenExtensions C application, you can also use sigaction() to affect the resulting behavior.

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.

Table 21. Control Signals

Value	Default Action	Meaning
SIGABND	1	Abend.
SIGABRT	1	Abnormal termination (sent by abort()).
SIGALRM	1	A timeout signal (sent by alarm()). This signal is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.
SIGFPE	1	Arithmetic exceptions that are not masked, for example, overflow, division by `0`, and incorrect operation.
SIGHUP	1	A controlling terminal is hung up, or the controlling process ended. This signal is supported only in a POSIX program.
SIGILL	1	Detection of an incorrect function image.
SIGINT	1	Interactive attention.
SIGKILL	1	A termination signal that cannot be caught or ignored. This signal is supported only in a POSIX program.
SIGPIPE	1	A write to a pipe that is not being read. This signal is supported only in a POSIX program.
SIGQUIT	1	A quit signal for a terminal. This signal is supported only in a POSIX program.
SIGSEGV	1	Incorrect access to memory.
SIGTERM	1	Termination request sent to the program.
SIGTRAP	1	Internal for use by dbx or ptrace. This signal is supported only in a POSIX program.
SIGUSR1	1	Intended for use by user applications.
SIGUSR2	1	Intended for use by user applications.
SIGCHLD	2	An ended or stopped child process (SIGCLD is an alias name for this signal). This signal is supported only in a POSIX program.
SIGDCE	2	Signal is used by DCE. This signal is supported only in a POSIX program.

Table 21. Control Signals (continued)

Value	Default Action	Meaning
SIGIO	2	Completion of input or output. This signal is supported only in a POSIX program.
SIGIOERR	2	A serious I/O error was detected.
SIGSTOP	3	A stop signal that cannot be caught or ignored. This signal is supported only in a POSIX program.
SIGTSTP	3	A stop signal for a terminal. This signal is supported only in a POSIX program.
SIGTTIN	3	A background process attempted to read from a controlling terminal. This signal is supported only in a POSIX program.
SIGTTOU	3	A background process attempted to write to a controlling terminal. This signal is supported only in a POSIX program.
SIGCONT	4	If stopped, continue. This signal is supported only in a POSIX program.

Default Actions

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.

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

If successful, raise() returns a value of `0`.

If unsuccessful, raise() returns a nonzero value.

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 */

void sig_hand(int);

{
   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 38
- “kill() — Send a Signal to a Process” on page 439
- “pthread_kill() — Send a Signal to a Thread” on page 586
- “sigaction() — Examine or Change a Signal Action” on page 749
- “signal() — Handle Interrupts” on page 767
- “sigprocmask() — Examine or Change a Thread” on page 773

rand() — Generate Random Number

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`. You should 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 42
- “srand() — Set Seed for rand() Function” on page 800

random() — Generate a Better Random Number

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

Format

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

long random(void);
```

General Description

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 $16 \times (2^{31}-1)$ and is determined by the size of the state array. 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

Returns the generated pseudorandom number.

Related Information

- “stdlib.h” on page 42

- “drand48() — Generate Pseudo-Random Number” on page 178
- “initstate() — Initialize Generator for random()” on page 415
- “setstate() — Change Generator for random()” on page 739
- “srandom() — Use Seed to Initialize Generator for random()” on page 801

read() — Read From a File

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

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 specified by *fs*, reads *N* bytes of input into the memory area specified 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. No file offset is associated with such a file.

If *fs* refers to a socket, read() is equivalent to recv() with no flags set.

fs
The file or socket descriptor.

buf
The pointer to the buffer that receives the data.

N
The length in bytes of the buffer pointed to by the *buf* 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 fewer bytes are 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 value of -1 and sets errno to EWOULDBLOCK. For a description of how to set nonblocking mode, see "ioctl() — Control Device" on page 417 or "fcntl() — Control Open File Descriptors" on page 207.

For datagram sockets, this call returns the entire datagram that was sent, provided that the datagram fits into the specified buffer. The call discards excess datagram data. 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, AF_INET6, and SOCK_DGRAM type sockets, bulk mode I/O will be supported only after the socket has been connected and the setibmsockopt() or sock_do_bulkmode() function is issued to set a socket for bulk mode use.

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:

- The read() function reached the end of the file before reading the requested number of bytes.
- The read() function 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.

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 a value of -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 a value of 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 a value of 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 a value of -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 a value of -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.

The read() function 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 bytes of binary zero 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.

If read() fails, it returns a value of -1 and sets errno to one of the following values under the conditions described:

EAGAIN
O_NONBLOCK is set to 1, but data was not available for reading.

EBADF
fs is not a valid open file descriptor.

ECONNRESET
A connection was forcibly closed by a peer.

EFAULT
Using the *buf* and *N* parameters would result in an attempt to access memory outside the caller's virtual machine.

EINTR
The read() function was interrupted by a signal before it could read any data.

EINVAL
N contains a value that is less than 0, or the request is invalid or not supported, or the STREAM or multiplexer referenced by *fs* is linked (directly or indirectly) downstream from a multiplexer.

EIO
One of the following conditions applies:

- 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 SIGTTIN signal or the process group of the process is orphaned.
- For sockets, an I/O error has occurred.

ENOBUFS
Insufficient system resources are available to complete the call.

ENOTCONN
A receive was attempted on a connection-oriented socket that is not connected. For AF_INET or AF_INET6, 0 is returned instead of recognizing this error.

ESHUTDOWN
There is no data to read on the socket, and it has been shut down for reading. For AF_INET or AF_INET6, 0 is returned instead of recognizing this error.

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.

Example

CBC3BR03

```
/* CBC3BR03
   This example opens a file and reads input.
 */
#define _OPEN_SYS
#include <fcntl.h>
#include <unistd.h>
#undef _OPEN_SYS
#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 28
- “unistd.h” on page 51
- “close() — Close a File” on page 127
- “connect() — Connect a Socket” on page 138
- “creat() — Create a New File or Rewrite an Existing One” on page 146
- “dup() — Duplicate an Open File Descriptor” on page 180
- “fcntl() — Control Open File Descriptors” on page 207
- “fread() — Read Items” on page 277
- “getsockopt() — Get Options Associated with a Socket” on page 370
- “ioctl() — Control Device” on page 417
- “lseek() — Change the Offset of a File” on page 469
- “open() — Open a File” on page 527
- “pipe() — Create an Unnamed Pipe” on page 539
- “readv() — Read Data on a Socket and Store in a Set of Buffers” on page 635
- “recv() — Receive Data on a Socket” on page 640
- “recvfrom() — Receive Messages on a Socket” on page 642
- “recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers” on page 645
- “select() — Monitor Activity on Files/Sockets and Message Queues” on page 680
- “selectex() — Monitor Activity on Files/Sockets and Message Queues” on page 685
- “send() — Send Data on a Socket” on page 694
- “sendmsg() — Send Messages on a Socket” on page 697
- “sendto() — Send Data on a Socket” on page 700
- “setsockopt() — Set Options Associated with a Socket” on page 734

- "socket() — Create a Socket" on page 791
- "write() — Write Data on a File or Socket" on page 975
- "writev() — Write Data on a Socket from an Array" on page 979

readdir() — Read an Entry from a Directory

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

Format

```
#define _OPEN_SYS
#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() 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.

If required, you should save the data from readdir() before calling closedir() because closedir() frees the data.

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 values under the conditions described:

EBADF
dir does not yield an open directory stream.

EINVAL
The buffer was too small to contain any directories.

Example

CBC3BR04

```
/* CBC3BR04
   This example reads the contents of a root directory.
 */
#define _OPEN_SYS
#include <dirent.h>
#include <errno.h>
#include <sys/types.h>
#undef _OPEN_SYS
#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 23
- “sys/types.h” on page 46
- “closedir() — Close a Directory” on page 129
- “opendir() — Open a Directory” on page 531
- “rewinddir() — Reposition a Directory Stream to the Beginning” on page 672

readlink() — Read the Value of a Symbolic Link

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

Format

```
#define _OPEN_SYS
#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, you should 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 value of 0 for *bufsiz*.

If readlink() is unsuccessful, it returns a value of -1 and sets errno to one of the following values under the conditions described:

EACCES
Search permission is denied for a component of the path prefix.

EINVAL
The named file is not a symbolic link.

ELOOP
A loop exists in symbolic links. The readlink() function issues this error if it detects more than POSIX_SYMLOOP symbolic links during resolution of the *path* 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 using pathconf().

ENOENT
The named file does not exist.

ENOTDIR
A component of the path prefix is not a directory.

Example

CBC3BR05

```
/* CBC3BR05 */
#define _OPEN_SYS
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _OPEN_SYS

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 51

- “lstat() — Get Status of File, External Link, or Symbolic Link” on page 471
- “stat() — Get File Information” on page 803
- “symlink() — Create a Symbolic Link to a Path Name” on page 854
- “unlink() — Remove a Directory Entry” on page 910

readv() — Read Data on a Socket and Store in a Set of Buffers

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

Format

X/Open

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

ssize_t readv(int socket, const struct iovec *iov, int iovcnt);
```

Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/uio.h>

int readv(int socket, struct iovec *iov, int iovcnt);
```

General Description

Reads data from a socket with descriptor *socket* and stores it in a set of buffers. The data is scattered into the buffers specified by iov[0]...iov[iovcnt-1]

socket
The socket descriptor.

iov
A pointer to an `iovec` structure.

iovcnt
The number of buffers pointed to by the *iov* parameter.

The `iovec` structure is defined in uio.h and contains the following fields:

iov_base
The pointer to the buffer.

iov_len
The length of the buffer.

If the descriptor refers to a socket, 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 requested bytes is available, the call returns the number currently available. If data is not available for the socket *socket* and the socket is in blocking mode, readv() blocks the caller until data arrives. If data is not available and *socket* is in nonblocking mode, readv() returns a value of -1 and sets errno to EWOULDBLOCK. For a description of how to set nonblocking mode, see “fcntl() — Control Open File Descriptors” on page 207 or “ioctl() — Control Device” on page 417.

For datagram sockets, this call returns the entire datagram that was sent, provided that the datagram fits into the specified buffer. The call discards excess datagram data. 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 a value of 0. If a readv() is interrupted by a signal before it reads any data, it returns a value of -1 and sets errno 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, 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, 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, readv() returns the number of bytes read into the buffer.

If unsuccessful, readv() returns a value of -1 and sets errno to one of the following values under the conditions described:

EAGAIN
The O_NONBLOCK flag is set for the file descriptor, and the process would be delayed by the readv().

EBADF
socket is not a valid socket descriptor.

ECONNRESET
A connection was forcibly closed by a peer.

EFAULT
Using *iov* and *iovcnt* would result in an attempt to access storage outside the caller's virtual machine.

EINTR
The readv() call was interrupted by a signal that was caught before any data was available.

EINVAL
iovcnt was not valid, or one of the fields in the *iov* array was not valid.

ENOBUFS
Insufficient system resources are available to complete the call.

ENOTCONN
A receive was attempted on a connection-oriented socket that is not connected. For AF_INET and AF_INET6, 0 is returned instead of recognizing this error.

ENOTSOCK
socket is a valid file descriptor, but not a socket.

ESHUTDOWN
There is no data to read on the socket, and it has been shut down for reading. For AF_INET and AF_INET6, 0 is returned instead of recognizing this error.

EWOULDBLOCK
The socket is in nonblocking mode, and data is not available to read.

Related Information

- “connect() — Connect a Socket” on page 138
- “fcntl() — Control Open File Descriptors” on page 207
- “getsockopt() — Get Options Associated with a Socket” on page 370
- “ioctl() — Control Device” on page 417
- “read() — Read From a File ” on page 628

- “recv() — Receive Data on a Socket” on page 640
- “recvfrom() — Receive Messages on a Socket” on page 642
- “recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers” on page 645
- “select() — Monitor Activity on Files/Sockets and Message Queues” on page 680
- “selectex() — Monitor Activity on Files/Sockets and Message Queues” on page 685
- “send() — Send Data on a Socket” on page 694
- “sendmsg() — Send Messages on a Socket” on page 697
- “sendto() — Send Data on a Socket” on page 700
- “setsockopt() — Set Options Associated with a Socket” on page 734
- “socket() — Create a Socket” on page 791
- “write() — Write Data on a File or Socket” on page 975
- “writev() — Write Data on a Socket from an Array” on page 979

realloc() — Change Reserved Storage Block Size

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, with *ptr* pointing to the beginning of the block and *size* giving 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, realloc() frees the storage pointed to by *ptr* and returns NULL.

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.

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 not enough storage is available to expand the block to the given size, the original block is unchanged and realloc() returns a NULL pointer.

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

- For information about the System Programming Facilities, see *z/OS: XL C/C++ Programming Guide*.
- "spc.h" on page 39
- "stdlib.h" on page 42
- "calloc() — Reserve and Initialize Storage" on page 103
- "free() — Free a Block of Storage" on page 278
- "malloc() — Reserve Storage Block" on page 477

realpath() — Resolve Path Name

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

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <stdlib.h>
char *realpath(const char * file_name, char *resolved_name);
```

General Description

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

If successful, realpath() returns a pointer to the resolved name.

If unsuccessful, realpath() returns a null pointer and sets errno to indicate the error. 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 *file_name*.

EINVAL
Either the *file_name* or *resolved_name* argument is a null pointer.

EIO
An error occurred while reading from the file system.

ELOOP
Too many symbolic links were encountered in resolving *path*

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 are determined using pathconf().

ENOENT
A component of *file_name* does not name an existing file, or *file_name* points to an empty string.

ENOTDIR
A component of the path prefix is not a directory.

ERANGE
File system will return ERANGE if the result to be stored in 'resolved_name' is larger than PATH_MAX.

Related Information

- "getcwd() — Get Path Name of the Working Directory" on page 320
- "sysconf() — Determine System Configuration Options" on page 856

recv() — Receive Data on a Socket

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

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

Receives data on a socket with descriptor *socket* and stores it in a buffer. The recv() call applies only to connected sockets.

socket
The socket descriptor.

buffer
The pointer to the buffer that receives the data.

length
The length in bytes of the buffer pointed to by the *buffer* parameter.

flags
A parameter that can be set to one or more of the following flags. If you specify more than one flag, you must use the logical OR operator (|) to separate them. This parameter applies to AF_INET and AF_INET6 only. It is ignored for AF_IUCV and AF_UNIX.

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() call.

The recipient of the data determines whether or not to receive out-of-band data 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 out-of-band data is available, and the MSG_OOB flag is not set (SO_OOBINLINE can be on or off), 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, 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, see "ioctl() — Control Device" on page 417 and "setsockopt() — Set Options Associated with a Socket" on page 734.

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 value of -1 and sets errno to EWOULDBLOCK. For a description of how to set nonblocking mode, see "fcntl() — Control Open File Descriptors" on page 207 or "ioctl() — Control Device" on page 417.

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, AF_INET6, and SOCK_DGRAM type sockets, bulk mode I/O will be supported only after the socket has been connected and the setibmsockopt() 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, recv() returns the length of the message or datagram in bytes.

A value of 0 indicates the connection is closed. If unsuccessful, recv() returns a value of -1 and sets errno to one of the following values under the conditions described:

EBADF
socket is not a valid socket descriptor.

ECONNRESET
A connection was forcibly closed by a peer.

EFAULT
Using the *buffer* and *length* parameters would result in an attempt to access storage outside the caller's virtual machine.

EINTR
The recv() call was interrupted by a signal that was caught before any data was available.

EINVAL
The request is not valid or not supported. The MSG_OOB flag is set and no out-of-band data is available.

EIO
A network or transport failure has occurred.

ENOBUFS
Insufficient system resources are available to complete the call.

ENOTCONN
A receive was attempted on a connection-oriented socket that is not connected. For AF_INET or AF_INET6, 0 is returned instead of recognizing this error.

ENOTSOCK
The descriptor is for a file, not for a socket.

EOPNOTSUPP
The specified flags are not supported for this socket type or protocol.

ESHUTDOWN
There is no data to read on the socket, and it has been shut down for reading. For AF_INET or AF_INET6, `0` is returned instead of recognizing this error.

EWOULDBLOCK
The socket is in nonblocking mode, and data is not available to read.

Related Information

- "connect() — Connect a Socket" on page 138
- "fcntl() — Control Open File Descriptors" on page 207
- "getsockopt() — Get Options Associated with a Socket" on page 370
- "ioctl() — Control Device" on page 417
- "read() — Read From a File " on page 628
- "readv() — Read Data on a Socket and Store in a Set of Buffers" on page 635
- "recvfrom() — Receive Messages on a Socket" on page 642
- "recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers" on page 645
- "select() — Monitor Activity on Files/Sockets and Message Queues" on page 680
- "selectex() — Monitor Activity on Files/Sockets and Message Queues" on page 685
- "send() — Send Data on a Socket" on page 694
- "sendmsg() — Send Messages on a Socket" on page 697
- "sendto() — Send Data on a Socket" on page 700
- "setsockopt() — Set Options Associated with a Socket" on page 734
- "socket() — Create a Socket" on page 791
- "write() — Write Data on a File or Socket" on page 975
- "writev() — Write Data on a Socket from an Array" on page 979

recvfrom() — Receive Messages on a Socket

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

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_len);
```

Berkeley Sockets

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

int recvfrom(int socket, char *buffer, int length, int flags,
             struct sockaddr *address, int *address_len);
```

General Description

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.

socket
The socket descriptor.

buffer
The pointer to the buffer that receives the data.

length
The length in bytes of the buffer pointed to by the *buffer* parameter.

flags
A parameter that can be set to 0, MSG_PEEK, or MSG_OOB. This parameter applies to AF_INET and AF_INET6 only. It is ignored for AF_IUCV and AF_UNIX.

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() command.

The recipient of the data determines whether or not to receive out-of-band data 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 out-of-band data is available, and the MSG_OOB flag is not set (SO_OOBINLINE can be on or off), 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, 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, see "ioctl() — Control Device" on page 417 and "setsockopt() — Set Options Associated with a Socket" on page 734.

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.

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_len
The size of *name* in bytes.

If *address* is nonzero, the source address of the message is filled. *address_len* must first be initialized to the size of the buffer associated with *address* and is then modified, on return, to indicate the actual size of the address stored there.

If either *address* or *address_len* is a null pointer, *address* and *address_len* are unchanged.

If *address* is nonzero, the source address of the message is filled. *address_len* must first be initialized to the size of the buffer associated with *address* 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 value of -1 and sets errno to EWOULDBLOCK. For a description of how to set nonblocking mode, see "fcntl() — Control Open File Descriptors" on page 207 or "ioctl() — Control Device" on page 417.

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, AF_INET6, and SOCK_DGRAM type sockets, bulk mode I/O will be supported when the setibmsockopt() 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, recvfrom() returns the length of the message or datagram in bytes.

If unsuccessul, recvfrom() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EBADF
socket is not a valid socket descriptor.

ECONNRESET
The connection was forcibly closed by a peer.

EFAULT
Using the *buffer* and *length* parameters would result in an attempt to access storage outside the caller's virtual machine.

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
A network or transport failure has occurred.

ENOBUFS
Insufficient system resources are available to complete the call.

ENOTCONN
A receive was attempted on a connection-oriented socket that is not connected. For AF_INET or AF_INET6, `0` is returned instead of recognizing this error.

ENOTSOCK
The descriptor is for a file, not for a socket.

EOPNOTSUPP
The specified flags are not supported for this socket type.

ESHUTDOWN
There is no data to read on the socket, and it has been shut down for reading. For AF_INET or AF_INET6, `0` is returned instead of recognizing this error.

EWOULDBLOCK
The socket is in nonblocking mode, and data is not available to read.

Related Information

- “fcntl() — Control Open File Descriptors” on page 207
- “getsockopt() — Get Options Associated with a Socket” on page 370
- “ioctl() — Control Device” on page 417
- “read() — Read From a File ” on page 628
- “readv() — Read Data on a Socket and Store in a Set of Buffers” on page 635
- “recv() — Receive Data on a Socket” on page 640
- “recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers” on page 645

- “select() — Monitor Activity on Files/Sockets and Message Queues” on page 680
- “selectex() — Monitor Activity on Files/Sockets and Message Queues” on page 685
- “send() — Send Data on a Socket” on page 694
- “sendmsg() — Send Messages on a Socket” on page 697
- “sendto() — Send Data on a Socket” on page 700
- “setsockopt() — Set Options Associated with a Socket” on page 734
- “socket() — Create a Socket” on page 791
- “write() — Write Data on a File or Socket” on page 975
- “writev() — Write Data on a Socket from an Array” on page 979

recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers

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

Format

X/Open

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

ssize_t recvmsg(int socket, struct msghdr *msg,
int flags);
```

Berkeley Sockets

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

int recvmsg(int socket, struct msghdr *msg,
 int flags);
```

General Description

Receives messages on a socket with descriptor *socket* and stores them in a set of "scatter/gather" buffers. The sender's address can also be returned.

socket
The socket descriptor.

msg
A pointer to a message header structure, which contains pointers to an array of "scatter/gather" buffers and to a sockaddr structure.

flags
A parameter that can be set to one or more of the following flags. If you specify more than one flag, you must use the logical OR operator (|) to separate them. This parameter applies to AF_INET and AF_INET6 only. It is ignored for AF_IUCV and AF_UNIX.

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() call.

The recipient of the data determines whether or not to receive out-of-band data 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 out-of-band data is available, and

the MSG_OOB flag is not set (SO_OOBINLINE can be on or off), 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, 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, see "ioctl() — Control Device" on page 417 and "setsockopt() — Set Options Associated with a Socket" on page 734.

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.

A message header is defined by a `msghdr` structure. The `msghdr` structure is defined in the sys/socket.h header file and contains the following elements:

msg_iov
An array of *iovec* buffers into which the message is placed.

msg_iovlen
The number of elements in the *msg_iov* array.

msg_name
An optional pointer to a buffer where the sender's address is stored.

msg_namelen
The size of the address buffer.

caddr_t msg_accrights
Access rights sent/received (ignored if specified by the user).

int msg_accrightslen
The length of access rights data (ignored if specified by the user).

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 value of `-1` and sets errno to EWOULDBLOCK. For a description of how to set nonblocking mode, see "fcntl() — Control Open File Descriptors" on page 207 or "ioctl() — Control Device" on page 417.

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.

If successful, 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, AF_INET6, and SOCK_DGRAM type sockets, bulk mode I/O will be supported when the setibmsockopt() 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, recvmsg() returns the length of the message in bytes.

If unsuccessful, recvmsg() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EBADF
socket is not a valid socket descriptor.

ECONNRESET
The connection was forcibly closed by a peer.

EFAULT
Using *msg* would result in an attempt to access storage outside the caller's virtual machine.

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 `iov_len` values overflows a `ssize_t`.

EIO
A network or transport failure has occurred.

ENOBUFS
Insufficient system resources are available to complete the call.

ENOTCONN
A receive was attempted on a connection-oriented socket that is not connected. For AF_INET or AF_INET6, `0` is returned instead of recognizing this error.

EOPNOTSUPP
The specified flags are not supported for this socket type.

ENOTSOCK
The descriptor is for a file, not for a socket.

ESHUTDOWN
There is no data to read on the socket, and it has been shut down for reading. For AF_INET or AF_INET6, `0` is returned instead of recognizing this error.

EWOULDBLOCK
The socket is in nonblocking mode, and data is not available to read.

Related Information

- “connect() — Connect a Socket” on page 138
- “fcntl() — Control Open File Descriptors” on page 207
- “getsockopt() — Get Options Associated with a Socket” on page 370
- “ioctl() — Control Device” on page 417
- “read() — Read From a File ” on page 628
- “readv() — Read Data on a Socket and Store in a Set of Buffers” on page 635
- “recv() — Receive Data on a Socket” on page 640
- “recvfrom() — Receive Messages on a Socket” on page 642
- “select() — Monitor Activity on Files/Sockets and Message Queues” on page 680
- “selectex() — Monitor Activity on Files/Sockets and Message Queues” on page 685
- “send() — Send Data on a Socket” on page 694
- “sendmsg() — Send Messages on a Socket” on page 697
- “sendto() — Send Data on a Socket” on page 700
- “setsockopt() — Set Options Associated with a Socket” on page 734
- “socket() — Create a Socket” on page 791

- "write() — Write Data on a File or Socket" on page 975
- "writev() — Write Data on a Socket from an Array" on page 979

re_comp() — Compile Regular Expression

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 z/VM OpenExtensions	both	

Format

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

char *re_comp(const char *string);
```

General Description

Converts a regular expression string into an internal form suitable for pattern matching by re_exec().

The *string* parameter is a pointer to a character string defining a source regular expression for compiling.

If you call re_comp() 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 new-line characters.

Notes:

- 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 the ISO POSIX.2 standard.
- The OpenExtensions implementation of the re_comp() function supports only the POSIX locale. Any other locales will cause unpredictable results.
- The re_comp() and re_exec() functions are supported on the thread level. You must issue them from the *same* thread for them 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 a 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 (that is, *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]` would match any strings that contain a, b, or c.

   If the caret symbol (^) is the first character of the string within square brackets (as in `[^string]`), the one-character RE matches any characters except new-line and the remaining characters within the square brackets. Thus, `[^abc]` would *fail* to match any strings that contain even one `a, b, or c`.

   You may specify ranges as *c*–*c*. The hyphen symbol, within square brackets, means "through." You may use it to indicate a range of consecutive ASCII characters. For example, `[0–9]` is equivalent to `[0123456789]`.

   You can use the – (hyphen) by itself, but only if it is the first (after an initial ^, if any) or last character in the expression.

   You can use the right square bracket (]) 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.

Use the following rules 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, with *m* being the minimum number and *u* being the maximum number; *u* must be less than 256. Specifying only *m* 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, resulting in 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 <= *n* <= 9) matches the same string of characters as an expression enclosed between \(and \) matched 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, \(*a*\)*r*\(*e*\)\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 successful, re_comp() returns a null pointer.

If unsuccessful, re_comp() returns a pointer to an error message string.

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 application will not free the error message string; it will be freed when the thread terminates.

Related Information

- "re_comp.h" on page 37
- "fnmatch() — Match File Name or Path Name" on page 253
- "glob() — Generate Path names Matching a Pattern" on page 389
- "regcomp() — Compile Regular Expression" on page 651
- "regexec() — Execute Compiled Regular Expression" on page 655
- "re_exec() — Match Regular Expression" on page 650

re_exec() — Match Regular Expression

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

Attempts to match the character string, pointed to by *string*, with the last regular expression passed to re_comp().

Strings passed to re_exec() must be terminated by a null byte and may include new-line characters.

Notes:

- The re_exec() 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 the ISO POSIX.2 standard.
- The OpenExtensions implementation of the re_exec() function supports only the POSIX locale. Any other locales will cause unpredictable results.
- The re_comp() and re_exec() functions are supported on the thread level. You must issue them from the *same* thread for them to work properly.

The re_exec() function supports simple regular expressions, which are defined in "re_comp() — Compile Regular Expression" on page 648.

Returned Value

If the input *string* matches the last compiled regular expression, re_exec() returns a value of 1.

If the input *string* fails to match the last compiled regular expression, re_exec() returns a value of 0.

If the compiled regular expression is invalid (indicating an internal error), re_exec() returns a value of -1.

Related Information

- "re_comp.h" on page 37
- "fnmatch() — Match File Name or Path Name" on page 253
- "glob() — Generate Path names Matching a Pattern" on page 389
- "re_comp() — Compile Regular Expression" on page 648
- "regcomp() — Compile Regular Expression" on page 651
- "regexec() — Execute Compiled Regular Expression" on page 655

regcomp() — Compile Regular Expression

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2 z/VM OpenExtensions	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 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.

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.

[*character–character*]
The hyphen symbol, within square brackets, means "through". It fills in the intervening characters according to the current collating sequence. For example, [a–z] can be equivalent to [abc...xyz] or, with a different collating sequence, it can be equivalent to [aAbBcC...xXyYzZ].

[*string*]
A string within square brackets specifies any of the characters in *string*. Thus `[abc]`, 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.

[*m*] [*m*,] [*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*,] is equivalent to [*m*,*u*]. They both match *m* or more occurrences of the expression. The **+** (plus) and ***** (asterisk) operations are equivalent to [1,] and [0,] respectively.

*****
The asterisk symbol indicates 0 or more of any characters. For example, [a*e] is equivalent to any of the following: 99ae9, aaaaae, a999e99.

$
The dollar symbol matches the end of the string. (Use \n to match a new-line character.)

character+
The plus symbol specifies one or more occurrences of a character. Thus, `smith+ern` is equivalent to, for example, `smithhhern`.

[^*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 that contains even one a, b, or c.

(*expression*)$*n*
Stores the value matched by the enclosed regular expression in the $(n+1)^{th}$ *ret* parameter. Ten enclosed regular expressions are allowed. Assignments are made unconditionally.

(*expression*)
Groups a subexpression allowing an operator, such as *****, **+**, or [ ].], to work on the subexpression enclosed in parentheses, for example, `(a*(cb+)*)$0`.

Rules:

- You should *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 should 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, you should use \–.

Returned Value

If successful, regcomp() returns a value of 0.

If unsuccessful, regcomp() returns a nonzero value, 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 37
- "regerror() — Return Error Message" on page 653
- "regexec() — Execute Compiled Regular Expression" on page 655
- "regfree() — Free Memory for Regular Expression" on page 657

regerror() — Return Error Message

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 651.)

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 regerror() 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

- For information about internationalization, see *z/OS: XL C/C++ Programming Guide*.
- "regex.h" on page 37
- "regcomp() — Compile Regular Expression" on page 651
- "regexec() — Execute Compiled Regular Expression" on page 655
- "regfree() — Free Memory for Regular Expression" on page 657

regexec() — Execute Compiled Regular Expression

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* against the compiled regular expression, *preg*. (For a description of regular expressions, see "Regular Expressions" on page 651.)

nmatch is the number of subexpressions to match.

pmatch is an array of offsets into *string*; which matched the corresponding sub-expressions 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* is `0` or REG_NOSUB was set on the call to regcomp(), regexec() ignores *pmatch*. Otherwise, *pmatch* 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 more than *nmatch* subexpressions exist, regexec() only records the first *nmatch* -1.

Rules: 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, newline characters are allowed in *string*.

Notes:

- The string passed to regexec() 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 regexec().
- The information returned by regexec() in the `regmatch_t` structure has the shift-state at the start and end of the added string. This will help an application replace or process 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 a value of `0`; otherwise, it returns a nonzero value, 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 = regexec(&preg, string, nmatch, pmatch, 0)) != 0) {
       printf("failed to ERE match '%s' with '%s',returning %d.\n",
       string, pattern, rc);
    }

    regfree(&preg);
}
```

Related Information

- For information about internationalization, see *z/OS: XL C/C++ Programming Guide*.
- “regex.h” on page 37
- “regcomp() — Compile Regular Expression” on page 651
- “regerror() — Return Error Message” on page 653
- “regfree() — Free Memory for Regular Expression” on page 657

regfree() — Free Memory for Regular Expression

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 651.)

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

- For information about internationalization, see *z/OS: XL C/C++ Programming Guide*.
- "regex.h" on page 37
- "regcomp() — Compile Regular Expression" on page 651
- "regerror() — Return Error Message" on page 653
- "regexec() — Execute Compiled Regular Expression" on page 655

release() — Delete a Load Module

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	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.

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 specifies it.

You cannot release any module that contains any code written in another Language Environment-conforming language. See also notes in "fetch() — Get a Load Module" on page 221. Such Language Environment-conforming languages include: COBOL linked with Language Environment and assembler using the ASM15 macro with Language Environment. However, this restriction does not apply to VS COBOL II or to C itself.

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

If successful, release() returns a value of `0`

If unsuccessful, release() returns a nonzero value.

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 42
- "fetch() — Get a Load Module" on page 221
- "fetchep() — Share Writable Static" on page 231

remainder() — Remainder Function

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

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 63 for more information about IEEE floating-point.

Returned Value

If successful, remainder() returns the remainder of the division of *x* by *y*, as described.

If *y* is `0`, remainder() returns a value of `0` and sets errno to EDOM.

Special Behavior for IEEE

If successful, remainder() returns the remainder of the division of *x* by *y*.

If *y* is 0, remainder() returns NaNQ and sets errno to EDOM.

Related Information

- "math.h" on page 32
- "abs(), absf(), absl() — Calculate Integer Absolute Value" on page 68

remove() — Delete File

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 files. It also removes memory files that simulate PDSs.

The interpretation of the file name passed to remove() depends on whether the program is C and whether POSIX(ON) is specified. For more information about using POSIX support, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7. For full details about *filename* considerations, see the information about opening files in *z/OS: XL C/C++ Programming Guide*.

Memory files must exist and they must be closed. For an OpenExtensions application running POSIX(ON), however, memory files do not need to be closed when removing an HFS memory file. The OpenExtensions rules of interoperability apply.

For information about specifying file names for VM and byte file system files, see *z/OS: XL C/C++ Programming Guide*.

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

If successful, remove() deletes the file and returns a value of `0`. Otherwise, remove() returns a nonzero value, indicating 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>

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 40
- "fopen() — Open a File" on page 254
- "rename() — Rename File" on page 661

remque() — Remove an Element from a Doubly Linked List

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

Format

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

void remque(void *element);
```

General Description

Removes the *element* from a doubly linked list. The remque() function operates on structures that 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 has no returned value.

Related Information

- "search.h" on page 38
- "insque() — Insert an Element into a Doubly Linked List" on page 416

rename() — Rename File

Standards / Extensions	C or C++	Dependencies
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 specify the name of an existing file. The *newname* pointer must not specify the name of an existing file. You cannot rename an open CMS file; you can rename an open byte file system (BFS) file. In case of an error, the name of the file is not changed.

The rename() function renames memory files and DASD files. It also renames individual members of PDSs. It even renames files that simulate PDSs.

Special Behavior for POSIX C

Memory files must be closed unless you are working under OpenExtensions.

The interpretation of the file name passed to rename() depends on whether the program is running POSIX(ON) or POSIX(OFF). For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

You cannot rename a BFS file to a CMS file or rename a CMS file to a BFS file.

Both *oldname* and *newname* must be of the same type, that is, both directories or both files.

If *newname* already exists, rename() removes it before renaming *oldname* to *newname*. Thus, if *newname* specifies the name of an existing directory, it must be an empty directory.

If *oldname* points to a symbolic link, rename() renames the symbolic link. If *newname* points to a symbolic link, rename() removes the link and renames *oldname* to *newname*. The rename() function 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

If successful, rename() returns a value of `0`.

On an error, rename() returns a nonzero value and sets errno to one of the following values under the conditions described:

EACCES
An error occurred for one of these reasons:

- The process did not have search permission on some component of the old or new path name.
- The process did not have write permission on the parent directory of the file or directory to be renamed.
- *oldname* or *newname* were directories.
- The process did not have write permission on *oldname* or *newname*.

EBUSY
oldname and *newname* 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:

- *oldname* is part of the path name prefix of *newname*.
- *oldname* or *newname* refers to either `.` (dot) or `..` (dot-dot).
- *oldname* or *newname* starts with "//", which is not allowed for BFS files.

EISDIR
newname is a directory, but *oldname* is not a directory.

ELOOP
A loop exists in symbolic links. The rename() function issues this error if it detects the number of symbolic links during resolution of *oldname* or *newname* is greater than POSIX_SYMLOOP

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
No file or directory named *oldname* was found, or either *oldname* or *newname* was not specified.

ENOSPC
The directory intended to contain *newname* cannot be extended.

ENOTDIR
A component of the path name prefix for *oldname* or *newname* is not a directory, or *oldname* is a directory and *newname* is a file that is not a directory.

ENOTEMPTY
newname specifies a directory, but the directory is not empty.

EROFS
Renaming would require writing on a read-only file system.

EXDEV
oldname and *newname* identify files or directories on different file systems. OpenExtensionsdoes not support links between different file 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 40
- "fopen() — Open a File" on page 254
- "remove() — Delete File" on page 659

res_init() — Domain Name Resolver Initialization

Standards

Standards / Extensions	C or C++	Dependencies
BSD 4.3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_init(void);

struct __res_state _res;
```

General Description

The res_init() function is the Resolver function that initializes the __res_state structure for use by other Resolver functions. Initialization normally occurs on the first call to any of the IP address resolution routines commonly called the C/C++ Run-Time Library Resolver.

The res_init() routine does its initialization by passing the __res_state structure to the z/VM Resolver. The Resolver reads the first TCPIP DATA configuration file found in the CMS search order and updates the __res_state structure. The global configuration and state information in the __res_state structure is filled in based on the contents of the TCPIP DATA configuration file. The contents of the __res_state structure can be referenced using the *_res* macro, which calls internal function *__res()* and returns a pointer to the __res_state structure. Most of the values have reasonable defaults and can be left unchanged.

Value
  Description

_res.retrans
  Retransmission time interval is taken from the RESOLVERTIMEOUT statement found in the TCPIP DATA configuration file.

_res.retry
  The number of times to retransmit a request. It is taken from the RESOLVERUDPRETRIES statement found in the TCPIP DATA configuration file.

_res.options
  Options stored in _res.options are defined in <resolv.h> and are listed below. Options are stored as a simple bit mask containing the bitwise OR of the options enabled.

  Option
    Description

RES_INIT
True after the initial name server address and default domain name are initialized, because res_init() has been called. This option should only be tested but not set, except by the res_init() function.

RES_DEBUG
Print debugging messages.

RES_AAONLY
Accept authoritative answers only. With this option, res_send() should continue until it finds an authoritative answer or finds an error. Currently this is not implemented.

RES_USEVC
Use TCP connections for queries instead of UDP datagrams.

RES_STAYOPEN
Used with RES_USEVC to keep the TCP connection open between queries. This is useful only in programs that regularly do many queries. UDP should be the normal mode used.

RES_IGNTC
Ignore truncation errors, that is, don't retry with TCP. Currently unused.

RES_RECURSE
Set the recursion-desired bit in queries. This is the default. (res_send() does not do iterative queries and expects the name server to handle recursion.)

RES_DEFNAMES
If set, res_search() will append the default domain name to single-component names (those that do not contain a dot). This option is enabled by default.

RES_DNSRCH
If this option is set, res_search() will search for host names in the current domain and in parent domains. This is used by the standard host lookup routine gethostbyname(). This option is enabled by default.

_res.nscount
The number of name servers specified in the TCPIP DATA configuration file.

_res.*nsaddr_list[0]
The addresses of name servers specified by the NSINTERADDR or NAMESERVER statements found in the TCPIP DATA configuration file.

_res.dnsrch[0]
The beginning of the list of domains to be searched, as specified in the DOMAINSEARCH statement found in the TCPIP DATA configuration file. The structure will have either a Default DOMAIN or DOMAINSEARCH.

_res.defdname[0]
The Default Domain name, as specified in the DOMAINORIGIN statement found in the TCPIP DATA configuration file. The structure will have either a Default DOMAIN or SEARCH.

_res.pfcode
Currently this is not implemented.

Returned Value

If successful, res_init() returns 0.

If unsuccessful, res_init() returns -1 and sets h_errno to one of the following values:

Error Code
Description

NO_RECOVERY
An error occurred that will continue to fail if tried again. Storage could not be obtained for this thread to contain the _res structure.

TRY_AGAIN
An error occurred while initializing the __res_state structure name selected, which can be retried.

If successful, *_res* returns the address of __res_state structure.

If unsuccessful, *_res* returns NULL and sets errno to the following value:

Error Code
Description

ENOMEM
The storage needed to define the *_res* structure could not be obtained.

Related Information

- For additional information on the TCPIP DATA configuration, see *z/VM: TCP/IP Planning and Customization*.
- "arpa/nameser.h" on page 18
- "netinet/in.h" on page 34
- "resolv.h" on page 38
- "sys/types.h" on page 46
- "dn_comp() — Resolver Domain Name Compression" on page 174
- "dn_expand() — Resolver Domain Name Expansion" on page 175
- "dn_find() — Resolver Domain Name Find" on page 176
- "dn_skipname() — Resolver Domain Name Skipping " on page 177
- "gethostbyname() — Get a Host Entry by Name" on page 337
- "res_mkquery() — Make Resolver Query for Domain Name Servers (DNS)" on page 665
- "res_query() — Resolver Query for Domain Name Servers (DNS)" on page 666
- "res_querydomain() — Build Domain Name and Resolver Query " on page 667
- "res_search() — Resolver Query for Domain Name Servers (DNS) With Optional Rules" on page 668
- "res_send() — Send Resolver Query for Domain Name Servers (DNS)" on page 669

res_mkquery() — Make Resolver Query for Domain Name Servers (DNS)

Standards

Standards / Extensions	C or C++	Dependencies
BSD 4.3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_mkquery(int op, const char *dname, int class, int type, const u_char *data,
                int datalen, const u_char *newrr_in, u_char *buf, int buflen);
```

General Description

This routine is one of several functions used for making, sending and interpreting query and reply messages with Internet domain name servers (DNS).

The res_mkquery() function constructs a standard query message and places it in *buf*. It returns the size of the query, or -1 if the query is larger than *buflen*. The query type *op* is usually QUERY, but can be any

of the query types defined in <arpa/nameser.h>. The domain name for the query given by *dname*. The argument *newrr_in* is currently unused but is intended for making update messages.

Returned Value

If successful, res_mkquery() returns the size of the query.

If unsuccessful, res_mkquery() returns -1. The errors defined in <arpa/nameser.h> can be found in the buf.rcode, if an answer was supplied in the *buf* buffer. For the buffer mapping, see nameser.h.

Related Information

- "arpa/nameser.h" on page 18
- "netinet/in.h" on page 34
- "resolv.h" on page 38
- "sys/types.h" on page 46
- "dn_comp() — Resolver Domain Name Compression" on page 174
- "dn_expand() — Resolver Domain Name Expansion" on page 175
- "dn_find() — Resolver Domain Name Find" on page 176
- "dn_skipname() — Resolver Domain Name Skipping " on page 177
- "res_init() — Domain Name Resolver Initialization" on page 663
- "res_query() — Resolver Query for Domain Name Servers (DNS)" on page 666
- "res_querydomain() — Build Domain Name and Resolver Query " on page 667
- "res_search() — Resolver Query for Domain Name Servers (DNS) With Optional Rules" on page 668
- "res_send() — Send Resolver Query for Domain Name Servers (DNS)" on page 669

res_query() — Resolver Query for Domain Name Servers (DNS)

Standards

Standards / Extensions	C or C++	Dependencies
BSD 4.3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_query(const char *dname, int class, int type, u_char *answer, int anslen);
```

General Description

This routine is one of several functions used for making, sending and interpreting query and reply messages with Internet domain name servers (DNS).

The res_query() function provides an interface to the server query mechanism. It constructs a query, sends it to the local server, awaits a response, and makes preliminary checks on the reply. The query requests information of the specified *type* and *class* for the specified fully-qualified domain name *dname*. The reply message is placed in the *answer* buffer, whose length *anslen* is supplied by the caller.

Returned Value

If successful, res_query() returns the reply message in the *answer* buffer with length *anslen*.

If unsuccessful, res_query() returns -1 and sets h_errno to one of the following values:

Error Code
Description

HOST_NOT_FOUND
The host name provided is not known at any of the domain name servers queried for this request.

NO_DATA
An answer was received but no data was supplied in the *answer* buffer.

NO_RECOVERY
An error occurred that will continue to fail if tried again.

TRY_AGAIN
A error occurred querying the name selected, which can be retried.

Related Information

- “arpa/nameser.h” on page 18
- “netinet/in.h” on page 34
- “resolv.h” on page 38
- “sys/types.h” on page 46
- “dn_comp() — Resolver Domain Name Compression” on page 174
- “dn_expand() — Resolver Domain Name Expansion” on page 175
- “dn_find() — Resolver Domain Name Find” on page 176
- “dn_skipname() — Resolver Domain Name Skipping ” on page 177
- “res_init() — Domain Name Resolver Initialization” on page 663
- “res_mkquery() — Make Resolver Query for Domain Name Servers (DNS)” on page 665
- “res_querydomain() — Build Domain Name and Resolver Query ” on page 667
- “res_search() — Resolver Query for Domain Name Servers (DNS) With Optional Rules” on page 668
- “res_send() — Send Resolver Query for Domain Name Servers (DNS)” on page 669

res_querydomain() — Build Domain Name and Resolver Query

Standards

Standards / Extensions	C or C++	Dependencies
BSD 4.3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_querydomain(const char *name, const char *domain, int class, int type,
                    u_char *answer, int anslen);
```

General Description

This routine is one of several functions used for making, sending and interpreting query and reply messages with Internet domain name servers (DNS).

The res_querydomain() function builds a fully qualified domain name and returns a res_query() to the caller. The query requests information of the specified *type* and *class* for the specified host *name* and domain *domain*. The reply message is placed in the *answer* buffer, whose length *anslen* is supplied by the caller.

Returned Value

If successful, res_querydomain() returns a res_query() to the caller.

If unsuccessful, res_querydomain() returns -1 and sets h_errno to one of the following values:

Error Code
Description

HOST_NOT_FOUND
The host name provided is not known at any of the domain name servers queried for this request.

NO_DATA
An answer was received but no data was supplied in the *answer* buffer.

NO_RECOVERY
An error occurred that will continue to fail if tried again.

TRY_AGAIN
A error occurred querying the name selected, which can be retried.

Related Information

- “arpa/nameser.h” on page 18
- “netinet/in.h” on page 34
- “resolv.h” on page 38
- “sys/types.h” on page 46
- “dn_comp() — Resolver Domain Name Compression” on page 174
- “dn_expand() — Resolver Domain Name Expansion” on page 175
- “dn_find() — Resolver Domain Name Find” on page 176
- “dn_skipname() — Resolver Domain Name Skipping ” on page 177
- “res_init() — Domain Name Resolver Initialization” on page 663
- “res_mkquery() — Make Resolver Query for Domain Name Servers (DNS)” on page 665
- “res_query() — Resolver Query for Domain Name Servers (DNS)” on page 666
- “res_search() — Resolver Query for Domain Name Servers (DNS) With Optional Rules” on page 668
- “res_send() — Send Resolver Query for Domain Name Servers (DNS)” on page 669

res_search() — Resolver Query for Domain Name Servers (DNS) With Optional Rules

Standards

Standards / Extensions	C or C++	Dependencies
BSD 4.3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_search(const char *dname, int class, int type, u_char *answer, int anslen);
```

General Description

This routine is one of several functions used for making, sending and interpreting query and reply messages with Internet domain name servers (DNS).

The res_search() routine makes a query and awaits a response like res_query() but, in addition, it implements the default and search rules controlled by the RES_DEFNAMES and RES_DNSRCH options. It returns the first successful reply. The reply message containing information of the specified *type* and *class* for the specified fully-qualified domain name *dname* is placed in the *answer* buffer, whose length *anslen* is supplied by the caller.

Returned Value

If successful, res_search() returns the first successful reply.

If unsuccessful, res_search() returns -1 and sets h_errno to one of the following values:

Error Code
Description

HOST_NOT_FOUND
The host name provided is not known at any of the domain name servers queried for this request.

NO_DATA
An answer was received but no data was supplied in the *answer* buffer.

NO_RECOVERY
An error occurred that will continue to fail if tried again.

TRY_AGAIN
A error occurred querying the name selected, which can be retried.

Related Information

- “arpa/nameser.h” on page 18
- “netinet/in.h” on page 34
- “resolv.h” on page 38
- “sys/types.h” on page 46
- “dn_comp() — Resolver Domain Name Compression” on page 174
- “dn_expand() — Resolver Domain Name Expansion” on page 175
- “dn_find() — Resolver Domain Name Find” on page 176
- “dn_skipname() — Resolver Domain Name Skipping ” on page 177
- “res_init() — Domain Name Resolver Initialization” on page 663
- “res_mkquery() — Make Resolver Query for Domain Name Servers (DNS)” on page 665
- “res_query() — Resolver Query for Domain Name Servers (DNS)” on page 666
- “res_querydomain() — Build Domain Name and Resolver Query ” on page 667
- “res_send() — Send Resolver Query for Domain Name Servers (DNS)” on page 669

res_send() — Send Resolver Query for Domain Name Servers (DNS)

Standards

Standards / Extensions	C or C++	Dependencies
BSD 4.3	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_send(const u_char *msg, int msglen, u_char *answer, int anslen);
```

General Description

This routine is one of several functions used for sending query and reply messages with Internet domain name servers (DNS).

The res_send() routine sends a pre-formatted query in a message *msg* of length *msglen*, and returns an answer in the *answer* buffer, whose length *anslen* is supplied by the caller. The routine will call res_init() if RES_INIT is not set, send the query to the local name server, and handle timeouts and retries. The length of the reply message is returned, or -1 if there were errors.

Returned Value

If successful, res_send() returns the length of the reply message.

If unsuccessful, res_send() returns -1. The errors defined in <arpa/nameser.h> can be found in the buf.rcode, if an answer was supplied in the *answer* buffer. For the buffer mapping, see nameser.h.

Related Information

- “arpa/nameser.h” on page 18
- “netinet/in.h” on page 34
- “resolv.h” on page 38
- “sys/types.h” on page 46
- “dn_comp() — Resolver Domain Name Compression” on page 174
- “dn_expand() — Resolver Domain Name Expansion” on page 175
- “dn_find() — Resolver Domain Name Find” on page 176
- “dn_skipname() — Resolver Domain Name Skipping ” on page 177
- “res_init() — Domain Name Resolver Initialization” on page 663
- “res_mkquery() — Make Resolver Query for Domain Name Servers (DNS)” on page 665
- “res_query() — Resolver Query for Domain Name Servers (DNS)” on page 666
- “res_querydomain() — Build Domain Name and Resolver Query ” on page 667
- “res_search() — Resolver Query for Domain Name Servers (DNS) With Optional Rules” on page 668

rewind() — Set File Position to Beginning of File

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

The rewind() function has no returned value. If an error occurs, rewind() sets errno. 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 a value of -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",
       data3, data4);
}
```

Output

```
The values read back in are: 1 and -37
```

Related Information

- "stdio.h" on page 40
- "fgetpos() — Get File Position" on page 236
- "fseek() — Change File Position" on page 292
- "fsetpos() — Set File Position" on page 294
- "ftell() — Get Current File Position" on page 300

rewinddir() — Reposition a Directory Stream to the Beginning

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

Format

```
#define _OPEN_SYS
#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, calling rewinddir() updates the directory stream so that a subsequent readdir() can read the new contents.

Returned Value

The rewinddir() function has no returned value.

Example

CBC3BR15

```
/* CBC3BR15
   This example produces the contents of a directory by opening it,
   rewinding it, and closing it.
 */
#define _OPEN_SYS
#include <dirent.h>
#include <errno.h>
#include <sys/types.h>
#undef _OPEN_SYS
#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 23
- “sys/types.h” on page 46
- “closedir() — Close a Directory” on page 129
- “opendir() — Open a Directory” on page 531
- “readdir() — Read an Entry from a Directory” on page 632

rindex() — Search for a Character

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

Locates the last occurrence of *c* (converted to an unsigned character) in *string*.

The *string* argument 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 836.

Returned Value

Returns a pointer to the first occurrence of *c*.

If *c* was not found, rindex() returns a null pointer.

No errnos are defined for rindex().

Related Information

- “strings.h” on page 44
- “memchr() — Search Buffer” on page 492
- “index() — Search for Character” on page 407
- “strchr() — Search for Character” on page 811
- “strrchr() — Find Last Occurrence of Character in String” on page 836
- “strspn() — Search String” on page 837

- "strstr() — Locate Substring" on page 838

rint() — Round to Nearest Integral Value

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

Format

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

double rint(double x);
```

General Description

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 63 for more information about IEEE floating-point.

Returned Value

The rint() function is always successful.

Related Information

- "math.h" on page 32
- "abs(), absf(), absl() — Calculate Integer Absolute Value" on page 68
- "isnan() — Test for NaN" on page 431

rmdir() — Remove a Directory

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

Format

```
#define _OPEN_SYS
#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. The rmdir() function does not remove a directory that still contains files or subdirectories.

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.

The rmdir() function 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 a value of 0.

If unsuccessful, rmdir() returns a value of -1 and sets errno to one of the following values under the conditions described:

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), or, *pathname* starts with "//", which is not allowed for byte file system files.

ELOOP
A loop exists in symbolic links. More than POSIX_SYMLOOP (an integer defined in the limits.h header file) 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.

ENOTEMPTY
The directory still contains files or subdirectories.

EROFS
The directory to be removed is on a read-only file system.

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>
#undef _OPEN_SYS
#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 51
- "mkdir() — Make a Directory" on page 498
- "unlink() — Remove a Directory Entry" on page 910

rpmatch() — Test for a Yes/No Response Match

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	

Format

```
#include <stdlib.h>

int rpmatch(const char *response);
```

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.

Returned Value

If the string pointed to by *response* matches the affirmative expression in the current locale, the rpmatch() function returns one of the following values under the conditions described:

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 42

sbrk() — Change Space Allocation

Standards / Extensions	C or C++	Dependencies
XPG4.2 Single UNIX Specification, Version 2	both	

Format

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

void *sbrk(int incr);
```

General Description

Changes the space allocated for the calling process 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 `0`, sbrk() returns the current value of the program break and sets the newly-allocated space to `0`. However, if the application first decrements and then increments the break value, the contents of the reallocated space are not zeroed.

The sbrk() function allocates separate storage space from the storage space used by other memory allocation functions such as malloc() and calloc(). Because this storage space must be a contiguous segment, sbrk() allocates it from the initial heap segment only. Thus, it 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 sbrk() call. Since this is a separate segment of storage, an application that uses other memory allocation functions can use the sbrk() function, provided that the user's region is large enough to support extensive use of both types of memory allocation.

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, you may want to use other memory allocation functions, such as mmap(), to assure portability with all other memory allocation functions and with any function that uses other allocation functions. However, you should not use the other memory allocation functions for applications that require sbrk(). and should run these with an initial heap size that will satisfy the maximum storage requirements of the programs.

Note: The sbrk() function is not supported from a multithreaded environment; it will return in error if you invoke it in this environment.

Returned Value

If successful, sbrk() returns a value of `0`.

If unsuccessful, sbrk() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EINVAL
The caller is running in a multithreaded environment, which is not a valid environment for this function.

ENOMEM
The requested change would allocate more space than allowed for the calling process.

Related Information

- "brk() — Change Space Allocation" on page 98

scalb() — Load Exponent

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

Computes $x \cdot 16^n$, 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. The *radix* is 16 for hexadecimal floating-point and 2 for IEEE floating-point. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

Returned Value

If successful, scalb() returns the function of its arguments as described above.

The scalb() function will fail under the following conditions:

- If the result would underflow, scalb() will return a value of 0 and set errno to ERANGE.
- If the result would overflow, scalb() will return plus or minus HUGE_VAL according to the sign of *x* and set errno to ERANGE.

Special Behavior for IEEE

If successful, scalb() returns the value of the *x* parameter times 2 to the power of the *y* parameter.

If the result would overflow, scalb() returns ±HUGE_VAL according to the sign of *x* and sets errno to ERANGE. No other errors can occur.

Related Information

- "math.h" on page 32
- "ldexp(), ldexpf(), ldexpl() — Multiply by a Power of Two" on page 445

scanf() — Read and Format Data

The information for this function is included in "fscanf() - scanf() - sscanf() — Read and Format Data" on page 283.

seed48() — Initialize Pseudorandom Number

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 lcong48(), seed48(), and srand48() functions are initialization functions. You should invoke one of these before calling either the drand48(), lrand48(), or mrand48() function.

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 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)mod(2**48)     n>=0
```

The initial values of X, a, and c are:

```
X(0)= 1
a   = 5deece66d (base 16)
c   = b         (base 16)
```

IBM XL C/C++ for z/VM provides storage to save the most recent 48-bit integer value of the sequence, `X(i)`. The drand48(), lrand48(), and mrand48() functions share this storage. You may re-initialize the most recent 48-bit value in this storage by calling seed48(). 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].

You may change the values of a and c, by calling lcong48(). The seed48() function restores the initial values of a and c.

Special Behavior for OpenExtensions Services

You can make seed48() 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, IBM XL C/C++ for z/VM serializes access to the storage for `X(n)`, a, and c by functions in the `drand48` family when a multithreaded application calls them.

If you request thread-specific behavior, making calls to drand48(), lrand48(), and mrand48() from thread `t` generates 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))mod(2**48)     n>=0
```

IBM XL C/C++ for z/VM provides thread-specific storage to save the most recent 48-bit integer value of the sequence, `X(t,i)`. When you call seed48() from thread `t`, seed48() 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].

You may change the values of a(t) and c(t), for the thread t by calling lcong48(). When you call seed48() 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

Saves the most recent 48-bit integer value in the sequence, X(i), in an array of unsigned short ints provided by IBM XL C/C++ for z/VM 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 OpenExtensions Services

If you request thread-specific behavior for the drand48 family and call seed48() on thread t, seed48() 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 42
- “drand48() — Generate Pseudo-Random Number” on page 178
- “erand48() — Generate Pseudo-Random Number” on page 187
- “jrand48() — Generate Pseudorandom Number” on page 436
- “lcong48() — Initialize Pseudorandom Number” on page 444
- “lrand48() — Generate Pseudorandom Number” on page 467
- “mrand48() — Generate Pseudorandom Number” on page 510
- “nrand48() — Pseudo-Random Number Generator” on page 523
- “srand48() — Initialize Pseudorandom Number” on page 801

select() — Monitor Activity on Files/Sockets and Message Queues

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

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

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 and unnamed pipes. FIFOs are not supported.

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 32767.

The second half (the low-order 16 bits) gives the number of bits within a bit set that corresponds to the file or socket descriptors to check. This value should equal the greatest descriptor number to check +1.

If either half of the *nmsgsfds* parameter is equal to a value of `0`, the corresponding bit sets or arrays are assumed to be absent.

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 `0`. 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, set the second half of *nmsgsfds* to 8, the highest descriptor you specified +1. If your application checks sockets 3 and 4, set the second half of *nmsgsfds* 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 the 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 `fd_set` type 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

A 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 24.85 days (2,147,483 seconds). 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 the `fd_set` type. 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. You should increase the size of the bit sets 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 `sellist` structure will be used.

Note: The 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 runtime 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 allocating bit strings large enough to hold the largest descriptor value in your application program, rather than relying 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)`. You can use the other macros 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*. OpenExtensions 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.

FD_ZERO*(&fdset)*
Sets all bits in the bit set *fdset* to `0`. After this operation, the bit set does not contain sockets as elements. You should call this macro to initialize the bit set before calling FD_SET() to set a socket as a member.

Note: If you used malloc() to dynamically allocate a new area, you should *not* use the FD_ZERO() macro, for it can cause unpredictable results. You should zero the area using the memset() function.

FD_SET*(sock, &fdset)*
Sets the bit for the socket *sock* to `1`, making *sock* a member of the bit set *fdset*.

FD_CLR*(sock, &fdset)*
Clears the bit for the socket *sock* in the bit set *fdset*. This operation sets the appropriate bit to `0`.

FD_ISSET*(sock, &fdset)*
Returns > `0` if *sock* is a member of the bit set *fdset*. Returns `0` if *sock* is not a member of *fdset*. (This operation returns the bit representing *sock*.)

The following macros are provided to manipulate the *nmsgsfds* parameter and the return value from select():

_SET_FDS_MSGS*(nmsgsfds, nmsgs, nfds)*
Sets the high-order halfword of *nmsgsfds* to *nmsgs* and sets the low-order halfword of *nmsgsfds* to *nfds*.

_NFDS*(n)*
If the return value *n* from select() 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.

_NMSGS*(n)*
If the return value *n* from select() 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 bit set *fdset* in *readlist* and invoke FD_SET() for each socket to test.

A socket is ready for writing if buffer space is available for outgoing data. A nonblocking 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 bit set *fdset* in *writelist* with either FD_ZERO() or memset(), if dynamically allocated, and use FD_SET() for each socket to test.

A socket has an exceptional condition pending if it has received out-of-band data. A socket that was given to another program with the givesocket() call also has an exceptional condition pending when the target program successfully takes the socket using takesocket(). In this case, when select() indicates a pending exception condition, your program should call close() to close the given socket. A pipe has an exceptional condition pending when it is empty and not open for writing, or when it is not open for reading (for example, if either end of the pipe has been closed). A file descriptor referring to a tty, regular file, directory, character-special file, or a block-special file never has an exceptional condition pending.

A message queue is ready for reading when it has a message on it. It is considered ready for writing 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.

You 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 you want to check a set of file descriptors for read status only, you must set the appropriate bits in the bit set in the `sellist` structure pointed to by the *writelist* to `0`. If you want to check a set of message queues for write status only, you must set the appropriate elements in the array in the `sellist` structure pointed to by the *readlist* 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

Returns a value of `-1` to indicate an error set in errno. Returns a value of `0` to indicate an expired time limit.

When the return value is greater than 0, 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 32767, only 32767 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 65535, only 65535 will be reported.

If the return value is greater than 0, 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 0. Use the FD_ISSET() macro 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.

EBADF
One of the bit sets specified an invalid socket, or a message queue identifier is invalid. FD_ZERO() was probably not called to clear the bit set before the sockets were set.

EFAULT
One of the parameters contained an invalid address.

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 timeval structure was invalid or an *nmsgsfds* value was invalid.

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 shut down. To find out which descriptor is bad, you can code a loop to fstat() or select() individually 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 512
- “msgget() — Get Message Queue” on page 513
- “msgrcv() — Read and Receive a Message” on page 515
- “msgsnd() — Send a Message” on page 516
- “selectex() — Monitor Activity on Files/Sockets and Message Queues” on page 685

selectex() — Monitor Activity on Files/Sockets and Message Queues

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	

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

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.

For more information, see “select() — Monitor Activity on Files/Sockets and Message Queues” on page 680.

nmsgsfds
The number of message queues and the number of file or socket descriptors to check. (For a full description of this parameter and the following ones, see “select() — Monitor Activity on Files/Sockets and Message Queues” on page 680).

readlist
A pointer to an `fd_set` type, array of message queue identifiers, or `sellist` structure specifying descriptors and message queues to check for reading.

writelist
A pointer to an `fd_set` type, array of message queue identifiers, or `sellist` structure specifying descriptors and message queues to check for writing.

exceptlist
A pointer to an `fd_set` type, array of message queue identifiers, or `sellist` structure specifying descriptors and message queues to be checked for exceptional pending conditions.

timeout
A pointer to the time to wait for the selectex() call to complete.

ecbptr
This variable can contain one of the following values:

1. A pointer to a user event control block. To specify this usage of *ecbptr*, set the high-order bit to `'0'B`.
2. A pointer to a list of ECBs. To specify this usage of *ecbptr*, set the high order bit to `'1'B`.
3. A NULL pointer, indicating that no ECBs are specified.

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 value of `-1` to indicate an error set in errno. Returns a value of `0` to indicate an expired time limit or that an ECB is posted.

When the return value is greater than `0`, 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 `32767`, only `32767` 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 `65535`, only `65535` will be reported.

If the return value is greater than `0`, 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 `0`. Use the FD_ISSET() macro 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`.

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 selectex() function was interrupted before any of the selected events occurred and before the timeout interval expired.

EINVAL
One of the fields in the `timeval` 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 shut down. To find out which descriptor is bad, you can code a loop to fstat() or select() individually on each descriptor, without waiting, until you get a failure.

Related Information

- “accept() — Accept a New Connection on a Socket” on page 69
- “connect() — Connect a Socket” on page 138
- “msgctl() — Message Control Operations” on page 512
- “msgget() — Get Message Queue” on page 513
- “msgrcv() — Read and Receive a Message” on page 515
- “msgsnd() — Send a Message” on page 516
- “recv() — Receive Data on a Socket” on page 640
- “send() — Send Data on a Socket” on page 694

semctl() — Perform Semaphore Control Operations

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

Format

```
#define _XOPEN_SOURCE
#include <sys/sem.h>
```

```
int  semctl(int semid, int semnum, int cmd, ...);
```

General Description

Performs control operations in the semaphore set, *semid*, as specified by *cmd*.

Depending on the value of *cmd*, semctl() may ignore the *semnum* argument or identify one specific semaphore number.

The fourth argument is optional and depends upon the requested operation. 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:

Data Type	Parameter	Value
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 you use semctl() to identify one specific semaphore number for the GETVAL, SETVAL, GETPID, GETNCNT, and GETZCNT commands, references are made to this anonymous data structure for the semaphore *semnum*.

The *cmd* argument may specify the following semaphore control operations, which require the level of permission indicated. The <sys/sem.h> header: defines these symbolic constants.

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(). Successful execution clears the `semadj` value corresponding to the specified semaphore in all processes. This command requires alter permission.

GETPID
Returns the most recent process to update the semaphore (`sempid`) 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.

GETALL
Stores `semvals` for each semaphore in the semaphore set and places them into the array pointed to by `arg.array.`, where `arg.` is the fourth argument to semctl(). This command requires read permission. You must insure that the allocated storage is large enough to hold the number of semaphore elements. The number of semaphore values stored is `sem_nsems`, which you may obtain using the IPC_STAT command.

SETALL
Sets `semval` values for each semaphore in the sempahore set according to the array pointed to by arg.array, where `arg` is the fourth argument to semctl(). This command requires alter permission. Each `semval` value must be zero or positive. Successful execution clears the `semadj` values corresponding to each specified semaphore in all processes. You must insure that the allocated storage is large enough to hold the number of semaphore elements. The number of semaphore values set is `sem_nsems`, which you may obtain using the IPC_STAT command.

IPC_STAT
Obtains status information for the semaphore identifier, *semid*. This command requires read permission. The information is stored in the address specified by the fourth argument, defined by the `semid_ds` data structure.

IPC_SET
Sets the value of the `sem_perm.uid`, `sem_perm.gid`, and `sem_perm.mode` in `semid_ds` data structure for the semaphore identifier *semid* to those specified in the `semid_ds` structure pointed to by the fourth argument.

Any value for `sem_perm.uid` and `semperm.gid` may be set.

Only mode bits defined under the semget() function *semflg* argument may be set in `sem_perm.mode`.

IPC_SET can only be executed by a process with the appropriate privileges or that has an effective user ID equal to the value of `sem_perm.cuid` or `sem_perm.uid` in the `semid_ds` structure associated with *semid*.

IPC_RMID
Removes the semaphore identifier, *semid*, from the system and frees the storage for the set of semaphores in the `semid_ds` structure.

IPC_RMID can only be executed by a process with the appropriate privileges or that has an effective user ID equal to the value of `sem_perm.cuid` or `sem_perm.uid` in the `semid_ds` structure associated with *semid*.

Returned Value

If successful, the value returned by semctl() depends on the value of *cmd* as follows:

GETVAL
The value of `semval` is returned.

GETPID
The value of `sempid` is returned.

GETNCNT
The value of `semncnt` is returned.

GETZCNT
The value of `semzcnt` is returned.

All others
The value of zero is returned.

If unsuccessful, semctl() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EACCES
Operation permission (read or write) is denied to the calling process.

EINVAL
The value of *semid* is not a valid semaphore identifier; or the value of *semnum* is less than `0`, or greater than, or equal to the number of semaphores in the set; or *cmd* is not a valid command; or the bits specified for `sem_perm.mode` are undefined.

Note: The valid range of *semnum* is `0` to the number of semaphores in the set `-1`.

EPERM
The *cmd* argument has a value of IPC_RMID or IPC_SET and the effective user ID of the caller is not a process with appropriate privileges and is not the value of `sem_perm.cuid` or `sem_perm.uid` in the `semid_ds` data structure associated with *semid*.

ERANGE
The *cmd* argument has a value of SETVAL or SETALL, and the `semval` value to be set exceeds the system limit as defined in <sys/sem.h>.

Related Information

- "sys/sem.h" on page 45
- "sys/ipc.h" on page 45
- "semget() — Get a Set of Semaphores" on page 690
- "semop() — Perform Semaphore Operations" on page 692

semget() — Get a Set of Semaphores

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

Format

```
#define _XOPEN_SOURCE
#include <sys/sem.h>

int  semget(key_t key, int nsems, int semflg);
```

General Description

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:

- The *key* argument has a value of IPC_PRIVATE.
- The *key* argument is not associated with a semaphore ID and (`semflg` & IPC_CREAT) is nonzero.

Valid values for the *semflg* field include any combination of the following defined in <sys/ipc.h> and <modes.h>:

IPC_CREAT
Creates a semaphore if the *key* specified does not already have an associated ID. IPC_CREATE is ignored when IPC_PRIVATE is specified.

IPC_EXCL
Causes the semget() 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.

S_IRUSR
Permits read access when the effective user ID of the caller matches either `sem_perm.cuid` or `sem_perm.uid`.

S_IWUSR
Permits write access when the effective user ID of the caller matches either `sem_perm.cuid` or `sem_perm.uid`.

S_IRGRP
Permits read access when the effective group ID of the caller matches either `sem_perm.cgid` or `sem_perm.gid`.

S_IWGRP
Permits write access when the effective group ID of the caller matches either `sem_perm.cgid` or `sem_perm.gid`.

S_IROTH
Permits others read access.

S_IWOTH
Permits others write access.

When a semaphore set associated with *key* already exists, setting IPC_EXCL and IPC_CREAT in *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:

Data Type	Parameter	Value
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.

The following fields are initialized when a `sem_ds` data structure is created:

- The `sem_perm.cuid` and `sem_perm.uid` fields are set equal to the effective user ID of the calling process.
- The `sem_perm.cgid.` and `sem_perm.gid` fields are set equal to the 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 `sem_nsems` field is set to the value of *nsems*.
- The `sem_otime` field is set to `0`.
- The `sem_ctime` field is set to the current time.
- The anonymous data structure containing `semval` for each semaphore is not initialized. You should use the semctl() commands SETVAL and SETALL to initialize each semaphore's `semval` value.

Returned Value

If successful, semget() returns a non-negative semaphore identifier.

If unsuccessful, semget() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EACCES
A semaphore identifier exists for *key*, but access permission as specified by the low order 9 bits of *semflg* could not be granted.

EEXIST
A semaphore identifier exists for *key*, and both IPC_CREAT and IPC_EXCL are specified in *semflg*.

EINVAL
The value of *nsems* is either less than `0` or greater than the system limit. A semaphore identifier associated with *key* does not exist, and the *nsems* is `0`. A semaphore identifier associated with *key* already exists, and the *nsems* value specified on semget(), when the semaphore identifier was created, is less than the *nsems* value on the current semget(). The *semflg* argument specified flags not currently supported.

ENOENT
A semaphore identifier does not exist for *key*, and IPC_CREAT is not specified.

ENOSPC
A system limit of number of semaphore identifiers has been reached.

When *semflg* is `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, 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, semget() returns a value of -1 and sets errno to EACCES.
- If a semaphore identifier has not been created with *key* earlier, semget() returns -1 and sets errno to ENOENT.

Related Information

- "sys/ipc.h" on page 45
- "sys/sem.h" on page 45
- "sys/stat.h" on page 46
- "sys/types.h" on page 46
- "ftok() — Generate an Interprocess Communication (IPC) Key" on page 302
- "semctl() — Perform Semaphore Control Operations" on page 687
- "semop() — Perform Semaphore Operations" on page 692

semop() — Perform Semaphore Operations

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

Format

```
#define _XOPEN_SOURCE
#include <sys/sem.h>

int  semop(int semid, struct sembuf *sops, size_t nsops);
```

General Description

Performs semaphore operations atomically on a set of semaphores associated with the *semid* argument. The *sops* argument is a pointer to an array of `sembuf` data structures. The *nsops* argument is the number of `sembuf` structures in the array.

The `sembuf` structure 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:

Data Type	Parameter	Value
unsigned short int	`semval`	Semaphore value
pid_t	`sempid`	Process ID of last operation

Data Type	Parameter	Value
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.

The semop() function performs each semaphore operation, specified by sem_op, on the corresponding semaphore, specified by *semid* and sem_num.

The sem_op variable 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, semop() subtracts the absolute value of sem_op 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 0, 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, semop() decrements the value of semncnt, associated with the specified semaphore, and subtracts the absolute value of sem_op from semval.
     - The *semid* for which the calling process is awaiting action is removed from the system. When this occurs, semop() returns a value of -1 and sets errno to EIDRM.
     - The calling process receives a signal to be caught. When this occurs, semop() decrements the value of semncnt, associated with the specified semaphore, 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, semop() adds the value of sem_op to semval.
3. If sem_op is 0 and the calling process has read permission, one of the following will occur:
   - If semval is 0, 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 value of 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 semop() decrements the value of semzcnt associated with the specified semaphore.
     - The *semid* for which the calling process is awaiting action is removed from the system. When this occurs, semop() returns -1 and sets errno to EIDRM.
     - The calling process receives a signal to be caught. When this occurs, semop() decrements the value of semzcnt, associated with the specified semaphore, and the calling process resumes execution in the manner prescribed in sigaction().
     - Upon successful completion, semop() sets the value of sempid for each semaphore specified in the array pointed to by *sops* equal to the process ID of the calling process.

sem_flg contains the IPC_NOWAIT and SEM_UNDO flags described as follows:

IPC_NOWAIT
Causes semop() to return EAGAIN rather than place the thread into wait state.

SEM_UNDO
Maintains `semadj` adjustment values for each semaphore on a per process basis. If `sem_op` is not equal to zero and SEM_UNDO is specified, then `sem_op` 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. Using SETALL clears all `semadj` values in all processes.

Returned Value

If successful, semop() returns a value of `0`. and sets *semid* for each semaphore operated upon to the process ID of the calling process.

If unsuccessful, semop() returns a value of `-1` and sets errno to one of the following values under the conditions described:

E2BIG
The value *nsops* is greater than the system limit.

EACCES
Operation permission is denied to the calling process. Read access is required when `sem_op` is `0`. Write access is required when `sem_op` is not `0`.

EAGAIN
The operation would result in suspension of the calling process, but IPC_NOWAIT in `sem_flg` was specified.

EFBIG
`sem_num` is less than zero or greater or equal to the number of semaphores in the set specified in the semget() argument *nsems*.

EIDRM
semid was removed from the system while the invoker was waiting.

EINTR
The semop() function was interrupted by a signal.

EINVAL
The value of *semid* is not a valid semaphore identifier.

ENOSPC
The limit on the number of individual processes requesting a SEM_UNDO would be exceeded.

ERANGE
An operation would cause `semval` or `semadj` to overflow the system limit as defined in <sys/sem.h>.

Related Information

- “sys/ipc.h” on page 45
- “sys/sem.h” on page 45
- “sys/types.h” on page 46
- “exec Functions” on page 192
- “_exit() — End a Process and Bypass the Cleanup” on page 195
- “exit() — End Program” on page 197
- “fork() — Create a New Process” on page 259
- “semget() — Get a Set of Semaphores” on page 690
- “semctl() — Perform Semaphore Control Operations” on page 687

send() — Send Data on a Socket

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

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

Sends data on the socket with the *socket* descriptor. The send() call applies to all connected sockets.

socket
The socket descriptor.

buffer
The pointer to the buffer containing the message to transmit.

length
The length of the message pointed to by the *buffer* parameter.

flags
A parameter that can be set to one or more of the following flags. If you specify more than one flag, you must use the logical OR operator (|) to separate them. This parameter applies to AF_INET and AF_INET6 only. It is ignored for AF_IUCV and AF_UNIX.

MSG_OOB
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.

The recipient of the data determines whether or not to receive out-of-band data inline by the setting of the SO_OOBINLINE option of setsockopt(). For more information on receiving out-of-band data, see “recv() — Receive Data on a Socket” on page 640, “recvfrom() — Receive Messages on a Socket” on page 642, “recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers” on page 645, and “setsockopt() — Set Options Associated with a Socket” on page 734.

MSG_DONTROUTE
Turns on for the duration of the operation. This option is usually used only by diagnostic or routing programs.

If not enough buffer space is available 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 value of -1 and sets errno to EWOULDBLOCK. For a description of how to set nonblocking mode, see “fcntl() — Control Open File Descriptors” on page 207 or “ioctl() — Control Device” on page 417.

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

Returns a value of `-1` to indicate locally detected errors and sets errno to one of the values listed below. No indication of delivery failure is implied in the return value of this call.

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 incomplete.

EBADF
socket is not a valid 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 *buffer* and *length* parameters would result in an attempt to access storage outside the caller's virtual machine.

EINTR
A signal interrupted send() before any data was transmitted.

EIO
A network or transport failure has occurred.

EMSGSIZE
The message was too big to be sent as a single datagram.

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 *socket* argument is associated with a socket that does not support one or more of the values set in *flags*.

EPIPE
For a connected stream socket, the connection to the peer socket has been lost. A SIGPIPE signal is sent to the calling process.

ESHUTDOWN
There is no data to read on the socket, and it has been shut down for reading.

EWOULDBLOCK
The socket is in nonblocking mode, and no data buffers are available.

Related Information

- “connect() — Connect a Socket” on page 138
- “fcntl() — Control Open File Descriptors” on page 207
- “getsockopt() — Get Options Associated with a Socket” on page 370
- “ioctl() — Control Device” on page 417
- “read() — Read From a File ” on page 628
- “readv() — Read Data on a Socket and Store in a Set of Buffers” on page 635
- “recv() — Receive Data on a Socket” on page 640
- “recvfrom() — Receive Messages on a Socket” on page 642

- “recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers” on page 645
- “select() — Monitor Activity on Files/Sockets and Message Queues” on page 680
- “selectex() — Monitor Activity on Files/Sockets and Message Queues” on page 685
- “sendmsg() — Send Messages on a Socket” on page 697
- “sendto() — Send Data on a Socket” on page 700
- “socket() — Create a Socket” on page 791
- “write() — Write Data on a File or Socket” on page 975
- “writev() — Write Data on a Socket from an Array” on page 979

sendmsg() — Send Messages on a Socket

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

Format

X/Open

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

ssize_t sendmsg(int socket, struct msghdr *msg, int flags);
```

Berkeley Sockets

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

int sendmsg(int socket, struct msghdr *msg, int flags);
```

General Description

Sends messages on a socket with the *socket* descriptor passed in a set of "scatter/gather" buffers. The receiver's address can also be specified.

socket
The socket descriptor.

msg
A pointer to a message header structure, which contains pointers to an array of "scatter/gather" buffers and to a sockaddr structure.

flags
A parameter that can be set to one or more of the following flags. If you specify more than one flag, you must use the logical OR operator (|) to separate them. This parameter applies to AF_INET and AF_INET6 only. It is ignored for AF_IUCV and AF_UNIX.

MSG_OOB
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.

The recipient of the data determines whether or not to receive out-of-band data inline by the setting of the SO_OOBINLINE option of setsockopt(). For more information on receiving out-of-band data, see “recv() — Receive Data on a Socket” on page 640, “recvfrom() — Receive Messages on a Socket” on page 642, “recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers” on page 645, and “setsockopt() — Set Options Associated with a Socket” on page 734.

MSG_DONTROUTE
Turns on for the duration of the operation. This option is usually used only by diagnostic or routing programs.

A message header is defined by the `msghdr` structure. The `msghdr` is defined in the sys/socket.h header file and contains the following elements:

msg_iov
An array of *iovec* buffers containing the message.

msg_iovlen
The number of elements in the *msg_iov* array.

msg_name
An optional pointer to the buffer containing the recipient's address.

msg_namelen
The size of the address buffer.

caddr_t msg_accrights
Access rights sent/received (ignored if specified by the user).

int msg_accrightslen
The length of access rights data (ignored if specified by the user).

The sendmsg() call applies to sockets regardless of whether or not they are in the connected state. For connected sockets, the sockaddr portion of the `msghdr` structure is ignored.

This call returns the length of the data sent. If not enough buffer space is available 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 value of `-1` and sets errno to EWOULDBLOCK. For a description of how to set nonblocking mode, see "fcntl() — Control Open File Descriptors" on page 207 or "ioctl() — Control Device" on page 417.

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, sendmsg() returns the length of the message in bytes.

If unsuccessful, sendmsg() returns a value of `-1` and sets errno to one of the values listed below.

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 incomplete.

EAFNOSUPPORT
The address family is not supported (it is not AF_UNIX, AF_INET, or AF_INET6).

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* parameter would result in an attempt to access storage outside the caller's virtual machine.

EINTR
A signal interrupted sendmsg() before any data was transmitted.

EINVAL
msg_namelength is not the size of a valid address for the specified address family.

EIO
A network or transport failure has occurred.

EMSGSIZE
The message was too big to be sent as a single datagram. The default is *large-envelope-size*. (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 *socket* argument is associated with a socket that does not support one or more of the values set in *flags*.

EPIPE
For a connected stream socket, the connection to the peer socket has been lost. A SIGPIPE signal is sent to the calling process.

ESHUTDOWN
There is no data to read on the socket, and it has been shut down for reading.

EWOULDBLOCK
The socket is in nonblocking mode, and no data buffers are available.

The following are for AF_UNIX only:

EACCES
The process does not have search permission on a component of the path prefix, or it does not have write access to the named socket.

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 path name in the socket address.

ENAMETOOLONG
A component of a path name exceeded NAME_MAX characters, or an entire path name exceeded PATH_MAX characters.

ENOENT
A component of the path name does not name an existing file, or the path name is an empty string.

ENOTDIR
A component of the path prefix of the path name in the socket address is not a directory.

Related Information

- “connect() — Connect a Socket” on page 138
- “fcntl() — Control Open File Descriptors” on page 207

- "getsockopt() — Get Options Associated with a Socket" on page 370
- "ioctl() — Control Device" on page 417
- "read() — Read From a File " on page 628
- "readv() — Read Data on a Socket and Store in a Set of Buffers" on page 635
- "recv() — Receive Data on a Socket" on page 640
- "recvfrom() — Receive Messages on a Socket" on page 642
- "recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers" on page 645
- "select() — Monitor Activity on Files/Sockets and Message Queues" on page 680
- "selectex() — Monitor Activity on Files/Sockets and Message Queues" on page 685
- "sendmsg() — Send Messages on a Socket" on page 697
- "sendto() — Send Data on a Socket" on page 700
- "setsockopt() — Set Options Associated with a Socket" on page 734
- "socket() — Create a Socket" on page 791
- "write() — Write Data on a File or Socket" on page 975
- "writev() — Write Data on a Socket from an Array" on page 979

sendto() — Send Data on a Socket

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

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

Sends data on the socket with the *socket* descriptor. The sendto() call applies to either connected or unconnected sockets.

socket
The socket descriptor.

buffer
The pointer to the buffer containing the message to transmit.

length
The length of the message in the buffer pointed to by the *msg* parameter.

flags
A parameter that can be set to one or more of the following for sockets in the AF_INET and AF_INET6 domains. It is ignored for AF_IUCV and AF_UNIX.

MSG_OOB
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.

The recipient of the data determines whether or not to receive out-of-band data inline by the setting of the SO_OOBINLINE option of setsockopt(). For more information on receiving out-of-band data, see "recv() — Receive Data on a Socket" on page 640, "recvfrom() — Receive Messages on a Socket" on page 642, "recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers" on page 645, and "setsockopt() — Set Options Associated with a Socket" on page 734.

MSG_DONTROUTE
Turns on for the duration of the operation. This option is usually used only by diagnostic or routing programs.

address
The address of the target. For connected sockets, this address is ignored.

address_len
The size of the address pointed to by *address*.

If not enough buffer space is available 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 value of -1 and sets errno to EWOULDBLOCK. For a description of how to set nonblocking mode, see "fcntl() — Control Open File Descriptors" on page 207 or "ioctl() — Control Device" on page 417.

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, sendto() returns the number of characters sent.

If unsuccessful, sendto() returns a value of -1 and sets errno to one of the values listed below. When used with datagram sockets, no indication of delivery failure is implied in the return value of this call.

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 incomplete.

EAFNOSUPPORT
The address family is not supported (it is not AF_INET, AF_INET6, AF_IUCV, or AF_UNIX).

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 *buffer* and *length* parameters would result in an attempt to access storage outside the caller's virtual machine.

EINTR
A signal interrupted sendto() before any data was transmitted.

EINVAL
address_len is not the size of a valid address for the specified address family.

EIO
A network or transport failure has occurred.

EMSGSIZE
The message was too big to be sent as a single datagram. The default is *large-envelope-size*. (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 *socket* argument is associated with a socket that does not support one or more of the values set in *flags*.

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.

ESHUTDOWN
There is no data to read on the socket, and it has been shut down for reading.

EWOULDBLOCK
The socket is in nonblocking mode, and no data buffers are available.

The following are for AF_UNIX only:

EACCES
The process does not have search permission on a component of the path prefix, or it does not have write access to the named socket.

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 path name in the socket address.

ENAMETOOLONG
A component of a path name exceeded NAME_MAX characters, or an entire path name exceeded PATH_MAX characters.

ENOENT
A component of the path name does not name an existing file, or the path name is an empty string.

ENOTDIR
A component of the path prefix of the path name in the socket address is not a directory.

Related Information

- “read() — Read From a File ” on page 628
- “readv() — Read Data on a Socket and Store in a Set of Buffers” on page 635
- “recv() — Receive Data on a Socket” on page 640
- “recvfrom() — Receive Messages on a Socket” on page 642
- “recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers” on page 645

- "select() — Monitor Activity on Files/Sockets and Message Queues" on page 680
- "send() — Send Data on a Socket" on page 694
- "sendmsg() — Send Messages on a Socket" on page 697
- "socket() — Create a Socket" on page 791
- "write() — Write Data on a File or Socket" on page 975
- "writev() — Write Data on a Socket from an Array" on page 979

setbuf() — Control Buffering

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 *buffer* 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 IBM XL C/C++ for z/VM, the buffer is not used.

The setvbuf() function is more flexible than setbuf(), because you can specify the type of buffering and size of the 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 entire 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, you should make the buffer a variable allocated at file scope, 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.

CMS and VSAM file types do not support unbuffered I/O, causing requests for unbuffered I/O to fail.

Returned Value

The setbuf() function has no returned value. For details about errno values and buffers you may have set, see discussions about buffering in *z/OS: XL 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 40
- “fclose() — Close File” on page 206
- “fflush() — Write Buffer to File” on page 233
- “fopen() — Open a File” on page 254
- “setvbuf() — Control Buffering” on page 742

setcontext() — Restore User Context

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

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(). A prior call to getcontext() or a pass to a signal handler should create the *ucp* argument.

If the *ucp* argument was created with getcontext(), program execution continues as if the corresponding call of getcontext() had just returned.

If *ucp* 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 `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 *ucp* is `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.

The setcontext() function is similar in some respects to siglongjmp(), longjmp(), and _longjmp(). You cannot intermix the getcontext()–setcontext() pair, the sigsetjmp()–siglongjmp() pair, the setjmp()–longjmp() pair, and the _setjmp()–_longjmp() pair. You should only use setcontext() to restore a context saved by getcontext().

Recommendation: You should not use siglongjmp() from a signal handler to restore a context created with getcontext(), even though some compatibility exists with siglongjmp().

Note: If makecontext() has not modified the *ucontext* input to setcontext(), you should 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.

The setcontext() 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         Apointer to the context that will
                                 resume 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 z/VM C++ program, the behavior in terms of the destruction of automatic objects is undefined. This applies to both z/VM C++ and C/VM 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.

Special Behavior for XPLINK-compiled C++

Restrictions concerning setjmp.h and ucontext.h:

Note:

1. All XPLINK programs compiled with the V2.10 or later C compilers that are to run with Language Environment V2.10 or later libraries and use the **jmp_buf**, **sigjmp_buf** or **ucontext_t** types must not be compiled with C headers from Language Environment V2.9 or earlier.
2. Non-XPLINK functions compiled with any level of Language Environment headers must not define **jmp_buf**, **sigjmp_buf** or **ucontext_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), _longjmp(), setjmp(), _setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.
3. When __XPLINK__ is defined, the Language Environment V2.10 and later headers define a larger **jmp_buf**, **sigjmp_buf** or **ucontext_t** area that is required by setjmp(), getcontext(), and related functions when they are called from a XPLINK routine. If __XPLINK__ is not defined, the Language Environment V2.10 and later headers define a shorter **jmp_buf**, **sigjmp_buf** or **ucontext_t** area. The Language Environment headers prior to V2.10 also define the shorter version of these data areas. If a XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp_buf**, **sigjmp_buf** or **ucontext_t** area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.

Returned Value

If successful, setcontext() has no returned value.

If unsuccessful, setcontext() returns a value of -1.

No errnos are defined for setcontext()

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 a value of `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 51
- “getcontext() — Get User Context” on page 318
- “_longjmp() — Nonlocal Goto” on page 463
- “longjmp() — Restore Stack Environment” on page 465
- “makecontext() — Modify User Context” on page 475
- “sigaction() — Examine or Change a Signal Action” on page 749
- “siglongjmp() — Restore the Stack Environment and Signal Mask” on page 764
- “swapcontext() — Save and Restore User Context” on page 849

setegid() — Set the Effective Group ID

Standards / Extensions	C or C++	Dependencies
POSIX.1a	both	

Format

```
#define _OPEN_SYS
#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.

The setegid() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, setegid() returns a value of `0`.

If unsuccessful, setegid() returns a value of -1 and sets errno to one of the following values under the conditions described.

ECMSERR
CMS detected a system error, most likely insufficient storage.

ECPERR
CP detected a system error.

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 _OPEN_SYS
#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 51
- "exec Functions" on page 192

- "getegid() — Get the Effective Group ID" on page 325
- "getgid() — Get the Real Group ID" on page 329
- "setgid() — Set the Group ID" on page 711

setenv() — Add, Delete, and Change Environment Variables

Standards / Extensions	C or C++	Dependencies
POSIX.1a Language Environment	both	

Format

```
#include <stdlib.h>

int setenv(const char *var_name, const char *new_value, int change_flag)
```

General Description

Adds, changes, and deletes environment variables.

var_name is a pointer to a character string containing the name of the environment variable to be added, changed, or deleted. If you call setenv() with *var_name* containing an equal sign ("="), setenv() will fail; and errno will be set to indicate that an invalid argument was passed to setenv().

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 a value of 1 or `0`:

1
: 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.

Note: The value of the *change_flag* is irrelevant if *new_value*=NULL.

Environment variables set with setenv() will only exist for the life of the program and are not saved before program termination. For other ways to set environment variables, see *z/OS: XL 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 OE option.

For more information about using POSIX support, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, setenv() returns a value of 0.

If unsuccessful, setenv() returns a value of -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

- For information about using environment variable, see *z/OS: XL C/C++ Programming Guide*.
- "stdlib.h" on page 42
- "clearenv() — Clear Environment Variables" on page 122
- "getenv() — Get Value of Environment Variables" on page 326
- "system() — Execute a Command" on page 858

seteuid() — Set the Effective User ID

Standards / Extensions	C or C++	Dependencies
POSIX.1a	both	

Format

```
#define _OPEN_SYS
#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 supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, seteuid() returns a value of 0.

If unsuccessful, seteuid() returns a value of -1 and sets errno to one of the following values under the conditions described:

ECMSERR
CMS detected a system error, most likely insufficient storage.

ECPERR
CP detected a system error.

EINVAL
The value specified for *uid* is incorrect and is not supported by the implementation.

EPERM
The process does not have appropriate privileges, and *uid* does not match the real UID or the saved set UID.

Example

CBC3BS05

```
/* CBC3BS05
   This example changes the effective UID.
 */
#define _OPEN_SYS
#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 51
- "getuid() — Get the Real User ID" on page 378
- "setuid() — Set the Effective User ID" on page 739

setgid() — Set the Group ID

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

Format

```
#define _OPEN_SYS
#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 or saved set GID.

If *gid* is not the same as the process's real GID or saved set 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*.

The setgid() function does not change any supplementary GIDs of the calling process.

This function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, setgid() returns a value of 0.

If unsuccessful, setgid() returns a value of -1 and sets errno to one of the following values under the conditions described:

ECMSERR
CMS detected a system error, most likely insufficient storage.

ECPERR
The CP detected a system error.

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 _OPEN_SYS
#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 46
- “exec Functions” on page 192
- “getegid() — Get the Effective Group ID” on page 325
- “getgid() — Get the Real Group ID” on page 329
- “setuid() — Set the Effective User ID” on page 739

sethostent() — Open Host Information Data Set

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

Opens and rewinds the HOSTS SITEINFO file. If the *stayopen* flag is nonzero, the HOSTS SITEINFO file remains open after each call.

You can use the X_SITE environment variable to specify a file other than HOSTS SITEINFO.

Note: HOSTS LOCAL, HOSTS ADDRINFO, and HOSTS SITEINFO are described in *z/VM: TCP/IP Planning and Customization*.

Special Behavior for C++

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

Returned Value

The sethostent() function has no returned value.

Related Information

- "endhostent() — Work with a Host Entry" on page 183
- "endnetent() — Close Network Information Data Set" on page 184
- "gethostbyaddr() — Get a Host Entry by Address" on page 335
- "gethostbyname() — Get a Host Entry by Name" on page 337
- "gethostent() — Get Next Host Entry" on page 339

setibmsockopt() — Set Options Associated with a Bulk Mode Socket

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	

Format

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

int setibmsockopt(int s,int level,int optname,
     char *optval, size_t optlen);
```

General Description

Use setibmsockopt() with the *optname* SO_BULKMODE to place the UDP socket in bulk mode. Normally, UNIT transactions occur between the application and the TCP/IP virtual machine 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 supported only for receive-type socket calls. Currently, send-type socket calls are not supported for bulk mode. If a setibmsockopt() request is made to set the send queue size, the send queue size is set to `0` automatically for the socket, and a value of `0` 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 virtual machine in one UNIT transaction.

Notes:

- Bulk mode is valid for send calls send(), sendmsg(), sendto(), and write() but is not valid for writev(). 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 (that is, the buffer should be large enough to contain the entire message).
- 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. To pass the socket descriptor to a child using fork() and exec(), the parent MUST issue ibmsflush() (if sending in bulk mode) to clear the send buffer and setibmsockopt() (with b_onoff in the `ibm_bulkmode_struct` set to `0`) to turn bulk mode off in the parent. The child can then issue setibmsockopt() against the same socket to turn bulk mode back on and begin operations in bulk mode in the new process. To pass the socket descriptor back, the child MUST follow the same sequence.

socket
The socket descriptor.

level
The level for which the option is set.

optname
The name of a specified socket option.

optval
The pointer to option data. If *optname* is SO_BULKMODE, *optval* should point to an `ibm_bulkmode_struct`. If *optname* is SO_NONBLOCKLOCAL or SO_IGNOREINCOMINGPUSH, *optval* should point to an `int`.

optlen
The pointer to 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:

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. Specifying a value of `0` prevents queuing for inbound datagrams. The maximum value allowed is 65768.

b_max_send_queue_size
The maximum sending queue size in bytes. Specifying a value of `0` prevents queuing for outbound datagrams. If specifying a value greater than `0`, the minimum value is 37; the maximum value is 65768. This value is set to `0` 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 `0`, 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. If either address or buffer cannot be addressed, errno is set to EFAULT. If this element is `0`, no checking is performed.

b_max_send_queue_size_avail
The maximum send queue size in bytes that can be set by *b_max_send_queue_size* option on setibmsockopt(). This value is set to `0` 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 `0` 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.

You do not need to change the socket calls that receive datagrams in your program (recvfrom(), recv(), read(), or recvmsg()). 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 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 with errno set 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.

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

If successful, setibmsockopt() returns a value of `0`.

If unsuccessful, setibmsockopt returns a value of `-1` and sets errno to one of the following values under the conditions described:

EBADF
The *socket* parameter is not a valid socket descriptor.

EFAULT
Using the *optval* and *optlen* parameters would result in an attempt to access storage outside the caller's virtual machine.

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, 65768. The requested send queue size is less than the minimum required, 37. The SO_NONBLOCKALL option is used for a socket that is not in bulk mode.

ENOMEM
Not enough storage is available in the caller's virtual machine to allocate the receive queue to the *b_max_receive_queue_size* specified in the `ibm_bulkmode_struct`. *b_max_receive_queue_size* is reset to `0`, turning bulk mode off. Subsequent requests will pass through as normal receives.
Not enough storage is available in the caller's virtual machine to allocate the send queue to the *b_max_send_queue_size* specified in the `ibm_bulkmode_struct`. *b_max_send_queue_size* is reset to `0`, turning bulk mode off. Subsequent requests will pass through as normal sends.

ENOPROTOOPT
The *optname* parameter is unrecognized. The *level* parameter is not SOL_SOCKET. The domain of the socket descriptor is not AF_INET or AF_INET6. The socket descriptor is not a datagram type socket.

Example

The following is an example of the setibmsockopt() 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 = getibmsockopt(s, SOL_SOCKET, SO_BULKMODE, (char *) &bulkstr, &optlen);
  if (rc < 0) {
      tcperror("on getibmsockopt()");
     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 = setibmsockopt(s, SOL_SOCKET, SO_BULKMODE, (char *) &bulkstr, optlen);
 if (rc < 0) {
     tcperror("on setibmsockopt()");
     exit(-1);
 }
}
```

Related Information

- “sys/socket.h” on page 46
- “fcntl() — Control Open File Descriptors” on page 207
- “getibmsockopt() — Get Options Associated with a Bulk Mode Socket” on page 341
- “getsockopt() — Get Options Associated with a Socket” on page 370
- “ibmsflush() — Flush Application-side Datagram Queue” on page 398
- “setsockopt() — Set Options Associated with a Socket” on page 734

_setjmp() — Set Jump Point for a Nonlocal Goto

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

Saves a stack environment that _longjmp() can subsequently restore. The _setjmp() and _longjmp() functions provide a way to perform a nonlocal *goto* and are often used in signal handlers.

A call to _setjmp() saves the current stack environment 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. All variables, except register variables and nonvolatile automatic variables, that are accessible to the function receiving control contain the values they had when _longjmp() was called. The values of register variables are unpredictable, as are nonvolatile *auto* variables that change between calls to _setjmp() and _longjmp().

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 _longjmp() and _setjmp() are functionally identical to longjmp() and setjmp(), respectively, with the additional restriction that _longjmp() and _setjmp() do not manipulate the signal mask. Since on this implementation longjmp() and setjmp() do not manipulate the signal mask, 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, you should use sigsetjmp() and siglongjmp() instead of _setjmp() and _longjmp(), or setjmp() and longjmp().

You cannot intermix the _setjmp()—_longjmp() pair, the setjmp()—longjmp() pair, the sigsetjmp()—siglongjmp() pair, and the getcontext()—setcontext() pair. A stack environment saved by _setjmp() can be restored only by _longjmp().

Recommendation: Although on this implementation the _setjmp()—_longjmp() pair are functionally identical to the setjmp()—longjmp() pair and it is possible to intermix them, you should not do so.

Note: You should 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 z/VM C++ program, the behavior in terms of the destruction of automatic objects is undefined. This applies both to z/VM C++ and C/VM ILC modules. The use of _setjmp() and _longjmp() in conjunction with try(), catch(), and throw() is also undefined.

Special Behavior for XPLINK-compiled C++

Restrictions concerning setjmp.h and ucontext.h:

Note:

1. All XPLINK programs compiled with the V2.10 or later C compilers that are to run with Language Environment V2.10 or later libraries and use the **jmp_buf**, **sigjmp_buf** or **ucontext_t** types must not be compiled with C headers from Language Environment V2.9 or earlier.
2. Non-XPLINK functions compiled with any level of Language Environment headers must not define **jmp_buf**, **sigjmp_buf** or **ucontext_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), _longjmp(), setjmp(), _setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.
3. When __XPLINK__ is defined, the Language Environment V2.10 and later headers define a larger **jmp_buf**, **sigjmp_buf** or **ucontext_t** area that is required by setjmp(), getcontext(), and related functions when they are called from an XPLINK routine. If __XPLINK__ is not defined, the Language Environment V2.10 and later headers define a shorter **jmp_buf**, **sigjmp_buf** or **ucontext_t** area. The Language Environment headers prior to V2.10 also define the shorter version of these data areas. If an XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp_buf**, **sigjmp_buf** or **ucontext_t** area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.

Returned Value

The _setjmp() function returns a value of `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 38
- “getcontext() — Get User Context” on page 318
- “_longjmp() — Nonlocal Goto” on page 463
- “longjmp() — Restore Stack Environment” on page 465
- “setcontext() — Restore User Context” on page 704
- “setjmp() — Preserve Stack Environment” on page 718
- “siglongjmp() — Restore the Stack Environment and Signal Mask” on page 764
- “sigsetjmp() — Save Stack Environment and Signal Mask” on page 778
- “swapcontext() — Save and Restore User Context” on page 849

setjmp() — Preserve Stack Environment

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 longjmp() can subsequently restore. The setjmp() and longjmp() functions provide a way to perform a nonlocal *goto* and are often used in signal handlers.

A call to setjmp() saves the current stack environment 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. All variables except register variables and nonvolatile automatic variables that are accessible to the function receiving control contain the values they had when longjmp() was called. The values of register variables are unpredictable, as are nonvolatile *auto* variables that change between calls to setjmp() and longjmp().

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).

Special Behavior for POSIX C

To save and restore a stack environment that includes a signal mask, you should use sigsetjmp() and siglongjmp(), instead of setjmp().

For more information about using POSIX support, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

You *cannot* intermix the sigsetjmp()—siglongjmp() pair and the setjmp()—longj() pair. 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 z/VM C++ program, the behavior in terms of the destruction of automatic objects is undefined. This applies both to z/VM C++ and C/VM 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. Although on this implementation, these calls are functionally identical to the setjmp()—_longjmp() pair and it is possible to intermix them, you should not do so.

Special Behavior for XPLINK-compiled C++

Restrictions concerning setjmp.h and ucontext.h:

Note:

1. All XPLINK programs compiled with the V2.10 or later C compilers that are to run with Language Environment V2.10 or later libraries and use the **jmp_buf**, **sigjmp_buf** or **ucontext_t** types must not be compiled with C headers from Language Environment V2.9 or earlier.
2. Non-XPLINK functions compiled with any level of Language Environment headers must not define **jmp_buf**, **sigjmp_buf** or **ucontext_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), _longjmp(), setjmp(), _setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.
3. When __XPLINK__ is defined, the Language Environment V2.10 and later headers define a larger **jmp_buf**, **sigjmp_buf** or **ucontext_t** area that is required by setjmp(), getcontext(), and related functions when they are called from an XPLINK routine. If __XPLINK__ is not defined, the Language Environment V2.10 and later headers define a shorter **jmp_buf**, **sigjmp_buf** or **ucontext_t** area. The Language Environment headers prior to V2.10 also define the shorter version of these data areas. If an XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp_buf**, **sigjmp_buf** or

ucontext_t area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.

Returned Value

Returns a value of `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`.

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 a value of `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 38
- “longjmp() — Restore Stack Environment” on page 465
- “sigsetjmp() — Save Stack Environment and Signal Mask” on page 778
- “siglongjmp() — Restore the Stack Environment and Signal Mask” on page 764

setlocale() — Set Locale

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 SAA Language Environment z/VM OpenExtensions	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 "Supplied Locales" appendix in *z/OS: XL 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: 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.

The Category Argument of setlocale()

The category argument may be set to one of these values:

Table 22. Values for Category Arguments of setlocale

Category	Purpose
LC_ALL	Specifies all categories associated with the program's locale.
LC_COLLATE	Defines the collation sequence, that is, the relative order of collation elements (characters and multicharacter collation elements) in the program's locale. The collation sequence definition is used by regular expression, pattern matching, and sorting functions. These string functions are affected by the defined collation sequence: strcoll(), strxfrm(), wcscoll(), and wcsxfrm(). LC_CTYPE, LC_COLLATE, and LC_SYNTAX should refer to the same locale. Changing just one of them may invalidate another.

Table 22. Values for Category Arguments of setlocale (continued)

Category	Purpose
LC_CTYPE	Defines character classification and case conversion for characters in the program's locale. Affects the behavior of character-handling functions defined in the ctype.h header file: csid(), isalnum(), isalpha(), isblank(), iswblank() iscntrl(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), iswalnum(), iswalpha(), iswcntrl(), iswctype(), iswdigit(), iswgraph(), iswlower(), iswprint(), iswpunct(), iswspace(), iswupper(), iswxdigit(), isxdigit(), tolower(), toupper(), towlowe(), towuppe(), wcsid(), and wctype(). Affects behavior of the printf() and scanf() families of functions: fprintf(), printf(), sprintf(), fscanf(), scanf(), and sscanf(). Affects the behavior of wide-character input/output functions: fgetwc(), fgetws(), getwc(), getwchar(), fputwc(), fputws(), putwc(), putwchar(), and ungetwc(). Affects the behavior of multibyte and wide-character functions: mblen(), mbtowc(), mbstowcs(), wctomb(), wcstombs(), mbrlen(), mbrtowc(), mbsrtowcs(), wcrtomb(), wcsrtombs(), wcswidth(), wcwidth(), wcstod(), wcstol(), and wcstoul(). LC_CTYPE, LC_COLLATE, and LC_SYNTAX should refer to the same locale. Changing just one of them may invalidate another.
LC_MESSAGES	Affects the messages returned by the nl_langinfo() function, and it also has an effect on rpmatch(). The LC_Messages category will *not* affect the messages for the following functions: perror(), strerror(), and regerror(). In the locale of a C program running with POSIX(ON), it defines affirmative and negative response patterns.
LC_MONETARY	Affects monetary information returned by localeconv() and the strfmon() 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. The localeconv() function returns pointers to these strings with names found in the locale.h header file.
LC_NUMERIC	Affects the decimal-point character for the formatted input/output and string conversion functions, and the nonmonetary formatting information returned by the localeconv() function, specifically: • The printf() family of functions • The scanf() family of functions • strtod() • atof() The formatting rules and symbols are strings. The localeconv() function returns pointers to the strings with names found in the locale.h header file.
LC_TIME	Defines time and date format information in the program's locale used by the strftime() strptime(), and wcsftime() functions.

Table 22. Values for Category Arguments of setlocale (continued)	
Category	**Purpose**
LC_SYNTAX	Affects the behavior of functions that use encoded values to format characters: • printf() family of functions • scanf() family of functions • regcomp() • strfmon() LC_CTYPE also affects values that may be retrieved using the getsyntx() function. LC_CTYPE, LC_COLLATE, and LC_SYNTAX should refer to the same locale. Changing just one of them may invalidate another.
LC_TOD	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().

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. For more information about using POSIX support, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Locale Argument of setlocale()

Identifies the locale. For a list of locales provided by IBM refer to the appendix about supplied locales in *z/OS: XL 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 locale definition mechanism, are valid.

The Null-String ("") Locale Category

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). For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

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 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. For more information about old format locales, see *z/OS: XL C/C++ Programming Guide*.

Effect of setlocale() on Language Environment

The current locale set with the setlocale() function affects only some C library functions. (See Table 22 on page 721). It does not affect the new CEE locale set and query functions available under Language Environment and described in *z/OS: Language Environment Programming Guide (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ceea200_v2r5.pdf)*.

Special Behavior for OpenExtensions Services

The LOCPATH environment variable specifies a colon separated list of byte file system (BFS) directories. If LOCPATH is defined, setlocale() searches BFS directories in the order specified by LOCPATH for locale object files it requires. Locale object files in the BFS are produced by the localedef utility running under the OpenExtensions Shell and Utilities. If LOCPATH is not defined and setlocale() is called by a POSIX program, setlocale() looks in the default BFS locale directory, /usr/lib/nls/locale, for locale object files it requires. If setlocale() does not find a locale object it requires in the BFS, it converts the locale name to a PDS member name and searches locale PDS load libraries associated with the program calling setlocale().

Locale names may be file names, relative path names, or absolute path names. LOCPATH is used if file name rather than path name is specified. Also, // preceding a file name tells setlocale() to skip BFS search, to convert the name to a load module name of the form EDC$xxxx, and to load the accessed CMS search order 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 IBM XL C/C++ for z/VM is installed. See the information about using environment variables in *z/OS: XL C/C++ Programming Guide*.

   The environment variables are not currently supported under all 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.

Query 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, you should give a NULL pointer as the second parameter. For example, to query all the categories of your locale, you should 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() returns a string whose contents depend on the values of the *category* and *locale* arguments as shown in the following table.

Table 23. 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: • locale-name • locale-name-list: locale-name1, locale-name2, ..., if different names for one or more categories.
	new-locale-name	new-locale-name (same for all categories)
	"" (Null string)	One of these: • new-locale-name: environment-variable-value or C • locale-name-list: environment-variable-value-list if different names for one or more categories. 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

The LC_MESSAGES locale-name is not included if setlocale() was called by a program running POSIX(OFF).

If setlocale() fails, it returns a NULL pointer and does not change the program's locale. Failure can result if:

- An incorrect *category* value is used. This occurs when the value LC_MESSAGES is specified by a program running POSIX(OFF) or when the value does not correspond to a valid category.
- An incorrect *locale* value is used. This occurs when the value POSIX or LC_C_POSIX is specified by a program running POSIX(OFF), or when the value of *locale* is an incorrect or undefined locale name.
- 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

```
/* CBC3BS08
   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);
}
```

CBC3BS08 Output

If the example is run with POSIX(OFF), the result of printf() is:

```
string = "S370"
```

If the example is run with POSIX(ON), the result of printf() is:

```
string = "C,C,C,C,FRAN,C,C"
```

EUROSAMP

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);
}
```

EUROSAMP Output

```
String = Fr_BE.IBM-1148
String = Fr_BE.IBM-1148@euro
```

Related Information

- “locale.h” on page 30
- “getenv() — Get Value of Environment Variables” on page 326
- “localeconv() — Query Numeric Conventions” on page 454

setnetent() — Open Network Information Data Set

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

Opens and rewinds the HOSTS ADDRINFO file, which contains information about known networks. If the *stayopen* flag is nonzero, the HOSTS ADDRINFO file remains open after each call to setnetent().

You can use the X_ADDR environment variable to specify a file other than HOSTS ADDRINFO.

Note: HOSTS ADDRINFO is described in *z/VM: TCP/IP Planning and Customization*.

Special Behavior for C++

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

Returned Value

The setnetent() function has no returned value.

Related Information

- "endhostent() — Work with a Host Entry" on page 183
- "endnetent() — Close Network Information Data Set" on page 184
- "getnetbyaddr() — Get a Network Entry by Address" on page 347
- "getnetbyname() — Get a Network Entry by Name" on page 348
- "getnetent() — Get Next Network Entry" on page 350

setpgid() — Set Process Group ID for Job Control

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

Format

```
#define _OPEN_SYS
#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.

The setpgid() function cannot be used to change the process group of a child that has issued one of the EXEC functions or was created with one of the spawn() functions. On VM/ESA, if a parent wants to change

the process group of one of its children, it should set the SPAWN_SETGROUP flag in `inherit.flags` and specify the process group ID in `inherit.pgroup` on the call to spawn().

The setpgid() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, setpgid() returns a value of `0`.

If unsuccessful, setpgid() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EACCES
The value of *pid* matches the PID of a child of the calling process, but the child has successfully run one of the EXEC or spawn() functions.

EINVAL
pgid 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:

- The specified process is a session leader.
- *pid* matches the PID of a child of the calling process, but the child is not in the same session as the caller.
- *pgid* does not match the PID of the process specified by *pid*, and it does not match the PGID of any other process in the caller's session.

ESRCH
pid does not match the PID of the calling process or any of its children.

Example

CBC3BS09

```
/* CBC3BS09
   This example sets the PGID.
 */
#define _OPEN_SYS
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

main() {
  printf("original process group id is %d\n", (int) getpgrp());
  printf("parent is performing setpgid() on itself %d\n";
  if (setpgid(0, 0) != 0)
    perror("setpgid() error");
  printf("process group id is now %d\n", (int) getpgrp());
}
```

Output

```
original process group id is 5767174
parent is performing setpgid() on itself
process group id is now 131084
```

Related Information

- "unistd.h" on page 51
- "exec Functions" on page 192
- "getpgrp() — Get the Process Group ID" on page 354
- "setsid() — Create Session, Set Process Group ID" on page 733

- “tcsetpgrp() — Set the Foreground Process Group ID” on page 880

setpgrp() — Set Process Group ID

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

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 a value of -1 and sets errno to the following value:

EPERM
The calling process is a session leader.

Related Information

- “unistd.h” on page 51
- “exec Functions” on page 192
- “getpgrp() — Get the Process Group ID” on page 354
- “setpgid() — Set Process Group ID for Job Control” on page 729
- “setsid() — Create Session, Set Process Group ID” on page 733
- “tcsetpgrp() — Set the Foreground Process Group ID” on page 880

setprotoent() — Open Protocol Information File

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

Opens and rewinds the ETC PROTO file. If the *stayopen* flag is nonzero, the ETC PROTO file 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 setprotoent() function has no returned value.

Related Information

- "endprotoent() — Work with a Protocol Entry" on page 185
- "getprotobyname() — Get a Protocol Entry by Name" on page 358
- "getprotobynumber() — Get a Protocol Entry by Number" on page 359
- "getprotoent() — Get Next Protocol Entry" on page 360

setservent() — Open Network Services Information File

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

Opens and rewinds the ETC SERVICES file. If the *stayopen* flag is nonzero, the ETC SERVICES file remains open after each call. For more information on the ETC SERVICES file, see *z/VM: TCP/IP Planning and Customization*.

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 has no returned value.

Related Information

- "endservent() — Close Network Services Information Data Sets" on page 185
- "getservbyname() — Get a Server Entry by Name" on page 365
- "getservent() — Get Next Service Entry" on page 367

setsid() — Create Session, Set Process Group ID

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

Format

```
#define _OPEN_SYS
#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.

The setsid() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

If successful, setsid() returns the value of the caller's new PGID.

If unsuccessful, setsid() returns a value of -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 _OPEN_SYS
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

main() {
  pid_t pid;

  printf("original process group is %d\n", (int) getpgrp());
  if (setsid() == -1)
    perror("setsid() error");
  else
    printf("new process group id is %d\n", (int) getpgrp());
  exit(0);
}
```

Output

```
original process group id is 262152
new process group id is 262150
```

Related Information

- "sys/types.h" on page 46
- "exec Functions" on page 192
- "_exit() — End a Process and Bypass the Cleanup" on page 195
- "getpid() — Get the Process ID" on page 355
- "kill() — Send a Signal to a Process" on page 439
- "setpgid() — Set Process Group ID for Job Control" on page 729
- "sigaction() — Examine or Change a Signal Action" on page 749

setsockopt() — Set Options Associated with a Socket

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

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>
```

icmp6_filter Structure

To include the icmp6_filter structure in your program, add the following code:

```
#define _OPEN_SYS_SOCK_IPV6
#include <netinet/icmp6.h>
```

General Description

Sets options associated with an AF_IUCV, AF_INET, or AF_INET6 socket. Options can exist at multiple protocol levels.

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 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, you must set the *level* parameter 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. You can use the getprotobyname() call to return the protocol number for a named protocol.

Use the *option_value* and *option_length* parameters to pass data used by the particular set command, specified in *option_name*. The *option_value* parameter points to a buffer containing the data needed by the set command. The *option_value* parameter is optional. You can set it to the NULL pointer if data is not needed by the function. You must set the *option_length* parameter to the size of the data pointed to by *option_value*.

All 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:

IP_MULTICAST_TTL
Sets the IP time-to-live of outgoing multicast datagrams. The default value is 1 (that is, multicast only to the local subnet). The TTL value is passed back as `u_char`.

IP_MULTICAST_LOOP
Enables/disables loopback of outgoing multicast datagrams. The 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 back as `u_char`. Specifying `0` disables loopback. Specifying 1 enables 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 a GATEWAY statement, use that interface.
- If DEFAULTNET was specified and is multicast-capable, use that interface.

IP_ADD_MEMBERSHIP
Joins 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 a 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 IPv6 level:

Option
Description

IPV6_CHECKSUM
For a RAW (non-ICMPv6) socket, this option instructs the kernel to compute and store a checksum for output and verifies the received checksum on input. This prevents applications from having to perform source address selection on the packets sent. This option specifies an integer value into the user data where the checksum is located. This option can be disabled by specifying an option value of -1.

IPV6_JOIN_GROUP
Controls the receipt of multicast packets by joining the multicast group specified by the ipv6_mreq structure that is passed. The ipv6_mreq structure is defined in **netinet/in.h**.

IPV6_LEAVE_GROUP
Controls the receipt of multicast packets by leaving the multicast group specified by the ipv6_mreq structure that is passed. The ipv6_mreq structure is defined in **netinet/in.h**.

IPV6_MULTICAST_HOPS
Sets the hop limit for outgoing multicast packets. The hop limit value is passed in as an int.

IPV6_MULTICAST_IF
Sets the interface for outgoing multicast packets. An interface index is used to specify the interface. It is passed in as a u_int.

IPV6_MULTICAST_LOOP
If a multicast datagram is sent to a group to which the sending host itself belongs (on the outgoing interface), a copy of the datagram is looped back by the IP layer for local delivery if this option is set to one. If this option is set to zero, a copy is not looped back. Other option values return an errno of EINVAL. The default is one (loopback). The option value is passed in as an int.

IPV6_PKTINFO
Directs packets to be sent out over the specified interface with the specified IP address as the packet's source. The option value is passed in as an ipv6_mreq structure as defined in **netinet/in.h**.

IPV6_RECVHOPLIMIT
When this option is enabled, the received hop limit from an incoming packet will be returned to the application as ancillary data on recvmsg(). The option value is specified as an int. A non-zero value enables the option, zero disables the option.

IPV6_RECVPKTINFO
When this option is enabled, the destination address from an incoming packet and the interface over which the packet was received will be returned to the application as ancillary data on recvmsg(). The option value is specified as an int. A non-zero value enables the option, zero disables the option.

IPV6_UNICAST_HOPS
Used to control hop limit in outgoing unicast IPv6 packets. The hop limit value is passed in as an int.

IPV6_USE_MIN_MTU
Controls whether the IP layer will use the minimum MTU size (1280) for sending packets, bypassing path MTU discovery. The option value is passed in as an int. A non-zero value enables the option, zero disables the option.

IPV6_V6ONLY
Used to determine whether a socket is restricted to IPv6 communications only. The default setting is off. The option value is passed in as an int. A non-zero value means the option is enabled (socket can only be used for IPv6 communications). 0 means the option is disabled.

Note: To use these options, you must use the Feature Test Macro #define _OPEN_SYS_SOCK_IPV6.

The following options are recognized at the ICMPv6 level:

Option
Description

ICMP6_FILTER
Used to filter ICMPv6 messages. The option value is passed in as an icmp6_filter structure. The icmp6_filter structure is defined in **netinet/icmp6.h**.

The following options are recognized at the socket level:

SO_BROADCAST
Toggles the ability to broadcast messages. If this option is enabled, it allows the application program to send broadcast messages over *socket* 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 unsent data is present when close() is called, the calling application program 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 virtual machine waits to try to send the data. Although the data transfer is usually successful, it cannot be guaranteed because the TCP/IP virtual machine 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.

For AF_IUCV sockets, the default behavior of close() is to block the closing application until all pending data is written. When this option is enabled and l_linger is zero, a close() call will immediately close the connection and unsent data is discarded. When this option is enabled and l_linger is non-zero, a close() call will cause the calling application to block until all unsent data is transmitted or the connection has timed out.

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, making it available to recv(), recvfrom(), and recvmsg(), without having 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, 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_REUSEADDR
Toggles local address reuse. When this option is enabled, it allows local addresses that are already in use to be bound, which 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. If the association already exists, the error EADDRINUSE is returned.

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, setsockopt() returns a value of `0`.

If unsuccessful, setsockopt() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EAFNOSUPPORT
The address family is not supported.

EBADF
The *socket* parameter is not a valid socket descriptor.

EFAULT
Using the *option_value* and *option_length* parameters would result in an attempt to access storage outside the caller's virtual machine.

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. This error is also returned if setsockopt() is attempted on an AF_UNIX 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. For examples of how the getsockopt() options set are queried, see “getsockopt() — Get Options Associated with a Socket” on page 370.

```
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 207
- “getprotobyname() — Get a Protocol Entry by Name” on page 358
- “getsockopt() — Get Options Associated with a Socket” on page 370
- “ioctl() — Control Device” on page 417
- “socket() — Create a Socket” on page 791

setstate() — Change Generator for random()

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

Allows switching between state arrays used by the random() function once a state has been initialized. The calling thread uses the array defined by the *state* argument for further random-number generation until you call initstate() or setstate() again.

To restart a state array at a different point after initialization, you should call setstate() with the desired state, followed by srandom() with the desired seed.

Returned Value

If successful, setstate() returns a pointer to the previous state array.

If unsuccessful, setstate() returns a null pointer.

The setstate() 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 42
- “drand48() — Generate Pseudo-Random Number” on page 178
- “initstate() — Initialize Generator for random()” on page 415
- “rand() — Generate Random Number” on page 626
- “random() — Generate a Better Random Number” on page 627
- “srandom() — Use Seed to Initialize Generator for random()” on page 801

setuid() — Set the Effective User ID

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

Format

```
#define _OPEN_SYS
#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 or the saved UID of the process, setuid() always succeeds and sets the effective UID to be the same as the real 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 (GID), effective GID, and saved set UID to *uid*.

The setuid() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, setuid() returns a value of 0.

If unsuccessful, setuid() returns a value of -1 and sets errno to one of the following values under the conditions described:

EAGAIN
The process is currently not able to change UIDs.

ECMSERR
CMS detected a system error, most likely insufficient storage.

ECPERR
CP detected a system error.

EINVAL
The value of *uid* is incorrect.

EPERM
The process does not have appropriate privileges to set the UID to *uid*.

Example

CBC3BS11

```
/* CBC3BS11
   This example changes the effective UID.
 */
#define _OPEN_SYS
#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 46
- "exec Functions" on page 192
- "geteuid() — Get the Effective User ID" on page 327
- "getuid() — Get the Real User ID" on page 378
- "setgid() — Set the Group ID" on page 711

setutxent() — Reset to Start of utmpx Database

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

Format

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

void setutxent(void);
```

General Description

Resets the input to the beginning of the `utmpx` database, which was previously opened by calls to getutxid(), getutxent(), getutxline(), or pututxline() from the current thread. You should use setutxent() before each getutxid() and getutxline() search for a new entry if you want the entire database to be examined.

Because setutxent() returns thread-specific data, you can use it safely from a multithreaded 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 calls endutxent().

Returned Value

The setutxent() function has no returned value.

Related Information

- "endutxent() — Close the utmpx Database" on page 186
- "getutxent() — Read Next Entry in utmpx Database" on page 379
- "getutxid() — Search by ID utmpx Database" on page 380
- "getutxline() — Search by Line utmpx Database" on page 381
- "pututxline() — Write Entry to utmpx Database" on page 616
- "__utmpxname() — Change the utmpx Database Name" on page 915

setvbuf() — Control Buffering

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.

The location pointed to by *buf* designates an area that you provide that the runtime 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 runtime 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 byte file system files.

Value
Meaning

`_IONBF`
No buffer is used.

`_IOFBF`
Full buffering is used for input and output. You should use *buf* as the buffer and *size* as the size of the buffer.

`_IOLBF`
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(). You can do this by defining the array with file scope or by dynamically allocating the storage for the array using malloc().

For example, if you declare the buffer 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

If successful or even if it chooses not to use your buffer, setvbuf() returns a value of `0`. If an invalid value was specified in the parameter list or if the request cannot be performed, setvbuf() returns a nonzero value.

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)
{
   stream1 = fopen("myfile1.dat", "r");
   stream2 = fopen("myfile2.dat", "r");
   FILE *stream1, *stream2;

   /* 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

- For information about I/O operations, see *z/OS: XL C/C++ Programming Guide*.
- "stdio.h" on page 40
- "fclose() — Close File" on page 206
- "fflush() — Write Buffer to File" on page 233
- "fopen() — Open a File" on page 254
- "setbuf() — Control Buffering" on page 703

shmat() — Attach Shared Memory Segment

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

Format

```
#define _XOPEN_SOURCE
#include <sys/shm.h>

void  *shmat(int shmid, const void *shmaddr, int shmflg);
```

General Description

Attaches the shared memory segment associated with the shared memory identifier, *shmid*, to the address space of the calling process using one of the following criteria:

- If *shmaddr* is a null pointer, shmat() attaches the segment at the first available address as selected by the system.
- If *shmaddr* is not a null pointer, and the flag, SHM_RND was specified VM will ignore *shmaddr* and treat it as though it were null. Therefore, SHM_RND is also ignored.
- If *shmaddr* is not a null pointer, and the flag, SHM_RND, was not specified, VM will ignore *shmaddr* and treat it as though it were null. Therefore, SHM_RND is also ignored.
- The shmat() function attaches the segment 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, shmat() attaches the segment for reading and writing.

Note: These permissions are only enforced on the other shmat(), shmctl(), shmdt(), and shmget() calls. It is the application's responsibility to enforce them on other calls that access this memory.

Returned Value

If successful, shmat() 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 a value of -1 and sets errno to one of the following values under the conditions described:

EACCES
Operation permission is denied to the calling process.

EINVAL
The value of *shmid* is not a valid shared memory identifier.

EMFILE
The number of shared memory segments attached to the calling process would exceed the system-imposed limit.

ENOMEM
The available system storage is not large enough to accommodate the shared memory segment.

Related Information

- "sys/shm.h" on page 46
- "_exit() — End a Process and Bypass the Cleanup" on page 195
- "exit() — End Program" on page 197
- "fork() — Create a New Process" on page 259
- "shmctl() — Shared Memory Control Operations" on page 744
- "shmdt() — Detach Shared Memory Segment" on page 745
- "shmget() — Get a Shared Memory Segment" on page 746

shmctl() — Shared Memory Control Operations

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

Format

```
#define _XOPEN_SOURCE
#include <sys/shm.h>

int  shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

General Description

Provides a variety of shared memory control operations on the shared memory segment, *shmid*.

The *cmd* argument specifies the shared memory control operation and may be any of the following values:

IPC_STAT
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
Sets 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)

IPC_SET can only be executed by a process with the appropriate privileges or that has an effective user ID equal to the value of `shm_perm.cuid` or `shm_perm.uid` in the `shmid_ds` data structure associated with *shmid*.

IPC_RMIDehp0.
Removes the shared memory identifier, *shmid*, from the system and destroys the shared memory segment and `shmid_ds` data structure associated with *shmid*.

IPC_RMID can only be executed by a process with the appropriate privileges or that has an effective user ID equal 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 shmctl() when the last attachment is detached. Processing IPC_RMID ends further attaches.

Returned Value

If successful, shmctl() returns a value of `0`.

If unsuccessful, shmctl() returns a value of `-1`, and sets errno to one of the following values under the conditions described:

EACCES
The *cmd* argument is IPC_STAT, but the calling process does not have read permission.

EINVAL
The value of *shmid* is not a valid shared memory identifier, or the value of *cmd* is not a valid command.

EPERM
The value of *cmd*, is either IPC_RMID or IPC_SET and the effective user ID of the calling process does not have appropriate privileges and is not equal to the value of `shm_perm.cuid` or `shm_perm.uid` in the data structure associated with *shmid*.

Related Information

- “sys/ipc.h” on page 45
- “sys/shm.h” on page 46
- “shmat() — Attach Shared Memory Segment” on page 743
- “shmdt() — Detach Shared Memory Segment” on page 745
- “shmget() — Get a Shared Memory Segment” on page 746

shmdt() — Detach Shared Memory Segment

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

Format

```
#define _XOPEN_SOURCE
#include <sys/shm.h>

int  shmdt(const void *shmaddr);
```

General Description

Detaches from the calling process's address space the shared memory segment located at the address specified by *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 a value of `0`.

If unsuccessful, shmdt() returns a value of -1 and sets errno to the following value:

EINVAL
The value of *shmaddr* is not the data segment start address of a shared memory segment.

Related Information

- "sys/shm.h" on page 46
- "_exit() — End a Process and Bypass the Cleanup" on page 195
- "exit() — End Program" on page 197
- "fork() — Create a New Process" on page 259
- "shmat() — Attach Shared Memory Segment" on page 743
- "shmctl() — Shared Memory Control Operations" on page 744
- "shmget() — Get a Shared Memory Segment" on page 746

shmget() — Get a Shared Memory Segment

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

Format

```
#define _XOPEN_SOURCE
#include <sys/shm.h>

int  shmget(key_t key, size_t size, int shmflg);
```

General Description

Returns the shared memory identifier associated with *key*.

The `shmget` function creates a shared memory identifier, associated data structure, and shared memory segment of at least *size* bytes (see <sys/shm.h>) for *key* if one of the following is true:

1. The *key* argument has a value of IPC_PRIVATE.
2. The *key* argument does not already have a shared memory identifier associated with it, and the flag IPC_CREAT was specified.

Valid values for *shmflg* include any combination of the following constants, defined in <sys/ipc.h> and <modes.h>:

IPC_CREAT
Creates a shared memory segment if the *key* specified does not already have an associated ID. IPC_CREAT is ignored when IPC_PRIVATE is specified.

IPC_EXCL
Causes the shmget() 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.

S_IRUSR
Permits read access when the effective user ID of the caller matches either `shm_perm.cuid` or `shm_perm.uid`.

S_IWUSR
Permits write access when the effective user ID of the caller matches either `shm_perm.cuid` or `shm_perm.uid`.

S_IRGRP
Permits read access when the effective group ID of the caller matches either `shm_perm.cgid` or `shm_perm.gid`.

S_IWGRP
Permits write access when the effective group ID of the caller matches either `shm_perm.cgid` or `shm_perm.gid`.

S_IROTH
Permits other read access.

S_IWOTH
Permits other write access.

When a shared memory segment associated with *key* already exists, setting IPC_EXCL and IPC_CREAT in *shmflg* will force shmget() to fail.

The following fields are initialized when a `shmid_ds` data structure is created:

- The `shm_perm.cuid` and `shm_perm.uid` fields are set to the effective user ID of the calling process.
- The `shm_perm.cgid` and `sem_perm.gid` fields are set 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 `shm_segsz` field is set to the value of *size*.
- The `shm_lpid`, `shm_nattach`, `shm_atime`, and `shm_dtime` fields are set to zero.
- The value of `shm_ctime` is set to the current time.

Returned Value

If successful, shmget() returns a non-negative integer, namely a shared memory identifier.

If unsuccessful, shmget() returns a value of -1 and sets errno to one of the following values under the conditions described:

EACCES
A shared memory identifier exists for *key*, but operation permission as specified by the low order 9 bits of *shmflg* could not be granted.

EEXIST
A shared memory identifier exists for *key*, and both IPC_CREAT and IPC_EXCL are specified in *shmflg*.

EINVAL
A shared memory identifier does not exist for *key*, and the value of *size*, is less than the system-imposed minimum or greater than the system-imposed maximum. A shared memory identifier exists for the *key*, but the size of the segment associated with it is less that specified by *size*.

ENOENT
A shared memory identifier does not exist for *key*, and IPC_CREAT is not specified.

ENOMEM
The amount of available system storage was insufficient to create a shared memory identifier and associated shared memory segment.

ENOSPC
The system-imposed limit on the maximum number of allocated shared memory identifiers, system-wide, would be exceeded if a shared memory identifier were created.

When *shmflg* is 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, shmget() returns the associated shared memory identifier.
- If a shared memory 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, shmget() returns a value of -1 and sets errno to EACCES.
- If a shared memory identifier has not been created with *key* earlier, shmget() returns a value of -1 and sets errno to ENOENT.

Related Information

- "sys/ipc.h" on page 45
- "sys/shm.h" on page 46
- "sys/types.h" on page 46
- "ftok() — Generate an Interprocess Communication (IPC) Key" on page 302
- "shmat() — Attach Shared Memory Segment" on page 743
- "shmctl() — Shared Memory Control Operations" on page 744
- "shmdt() — Detach Shared Memory Segment" on page 745

shutdown() — Shut Down All or Part of a Duplex Connection

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

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

Shuts down all or part of a duplex connection.

socket
The socket descriptor.

how
The parameter that sets the condition for shutting down the connection to socket *socket*.

The *how* parameter 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

If successful, shutdown() returns a value of `0`.

If unsuccessful, shutdown() returns a value of `-1` and sets errno to one of the following values under the conditions described:

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. This error is not recognized for AF_INET or AF_INET6. A 0 is returned.

ENOTSOCK
The descriptor is for a file, not for a socket.

Related Information

- "accept() — Accept a New Connection on a Socket" on page 69
- "close() — Close a File" on page 127
- "connect() — Connect a Socket" on page 138
- "socket() — Create a Socket" on page 791

sigaction() — Examine or Change a Signal Action

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	POSIX(ON)

Format

```
#define _OPEN_SYS
#include <signal.h>

int sigaction(int sig,const struct sigaction *new, struct sigaction *old);
```

General Description

Examines and changes the signal mask of the calling thread.

The `int` *sig* is the number of a recognized signal. The sigaction() function examines and sets the action to be associated with this signal. For the values of *sig*, as well as the signals supported by OpenExtensions, see Table 24 on page 750. 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. The sigaction() function 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.

The sigaction() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

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"`.

Table 24. Control Signals

Value	Default Action	Meaning
SIGABND	1	Abend.
SIGABRT	1	Abnormal termination (sent by abort()).
SIGALRM	1	A timeout signal (sent by alarm()).
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.
SIGQUIT	1	A quit signal for a terminal.
SIGSEGV	1	Incorrect access to memory.
SIGTERM	1	Termination request sent to the program.
SIGTRAP	1	Internal for use by dbx or ptrace.
SIGUSR1	1	Intended for use by user applications.
SIGUSR2	1	Intended for use by user applications.
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 VM/ESA 2.1.0 or higher).
SIGIO	2	Completion of input or output.

Table 24. Control Signals (continued)

Value	Default Action	Meaning
SIGIOERR	2	A serious I/O error was detected.
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.

Default Actions

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.

Here is an example of a sigaction() structure that uses structures of the sigaction type.

```
struct sigaction {
      void       (*sa_handler)(int);
      sigset_t   sa_mask;
      int        sa_flags;
};
```

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).

`sigset_t sa_mask`
A signal set identifies a set of signals that will be added to the signal mask of the calling process before the signal handling function sa_handler is invoked. For more on signal sets, see “sigemptyset() — Initialize a Signal Mask to Exclude All Signals” on page 758. 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_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 767.) For a description of ANSI and POSIX.1 signal delivery rules, see *z/OS: XL C/C++ Programming Guide*.

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. This new mask stays in effect until the signal handler returns, or sigprocmask(), sigsuspend(), or siglonjump() 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(), or one of the exec functions.

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().

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.

You need to take this restriction 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 application, you should only assume the following functions to be reentrant with respect to asynchronous signals. You should only use these functions in your signal-catching functions:

access()	alarm()	cfgetispeed()
cfgetospeed()	cfsetispeed()	cfsetospeed()
chdir()	chmod()	chown()
closef()	creat()	dup()
dup2()	execle()	execve()
_exit()	fcntl()	fstat()
getegid()	geteuid()	getgid()
getgroups()	getpgrp()	getpid()
getgroups()	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()	spawn()	stat()
sysconf()	tcdrain()	tcflow()
tcflush()	tcgetattr()	tcgetpgrp()
tcsendbreak()	tcsetattr()	tcsetpgrp()
time()	times()	umask()
uname()	unlink()	utime()
wait()	waitpid()	write()

The macro versions of getc() and putc() are not reentrant, even though the library versions of these functions are.

For nonportable POSIX applications, you can use most of the library functions in a signal-catching function. However, you should not use the following functions:

- getgrgid()
- getgrnam()
- getpwnam()
- getpwuid()
- ttyname()

Special Behavior for XPLINK-compiled C++

Restrictions concerning setjmp.h and ucontext.h:

Note:

1. All XPLINK programs compiled with the V2.10 or later C compilers that are to run with Language Environment V2.10 or later libraries and use the **jmp_buf**, **sigjmp_buf** or **ucontext_t** types must not be compiled with C headers from Language Environment V2.9 or earlier.
2. Non-XPLINK functions compiled with any level of Language Environment headers must not define **jmp_buf**, **sigjmp_buf** or **ucontext_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), _longjmp(), setjmp(), _setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.
3. When __XPLINK__ is defined, the Language Environment 2.1 and later headers define a larger **jmp_buf**, **sigjmp_buf** or **ucontext_t** area that is required by setjmp(), getcontext(), and related functions when they are called from an XPLINK routine. If __XPLINK__ is not defined, the Language Environment 2.1 and later headers define a shorter **jmp_buf**, **sigjmp_buf** or **ucontext_t** area. The Language Environment headers prior to V2.10 also define the shorter version of these data areas. If an XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp_buf**, **sigjmp_buf** or **ucontext_t** area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.
4. The sigaction() function supersedes the signal() interface, and should be the preferred usage. In particular, sigaction() and signal() must not be used in the same process to control the same signal.

Returned Value

If successful, sigaction() returns a value of `0`.

If unsuccessful, sigaction() returns a value of -1. No new signal handler is installed. The sigaction() function 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 SIGCHILD and SIGIO is for the signal to be ignored. A sigaction() to set the action to SIG_IGN for either of these two signals will result in an error, with errno equal to EINVAL.

Examples

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 38
- "alarm() — Set an Alarm" on page 76
- "exec Functions" on page 192
- "kill() — Send a Signal to a Process" on page 439
- "raise() — Raise Signal" on page 623
- "sigaddset() — Add a Signal to the Signal Mask" on page 754
- "sigdelset() — Delete a Signal from the Signal Mask" on page 757
- "sigemptyset() — Initialize a Signal Mask to Exclude All Signals" on page 758
- "sigfillset() — Initialize a Signal Mask to Include All Signals" on page 760
- "siglongjmp() — Restore the Stack Environment and Signal Mask" on page 764
- "signal() — Handle Interrupts" on page 767
- "sigprocmask() — Examine or Change a Thread" on page 773
- "sigsuspend() — Change Mask and Suspend the Thread" on page 781

sigaddset() — Add a Signal to the Signal Mask

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

Format

```
#define _OPEN_SYS
#include <signal.h>

int sigaddset(sigset_t *set, int signal);
```

General Description

Adds a signal to the set of signals already recorded in *set*.

The sigaddset() 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()).

The sigaddset() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful sigaddset() returns a value of 0.

If *signal* is not supported, sigaddset() returns a value of -1 and sets errno to EINVAL.

Example

CBC3BS15

```
/* CBC3BS15
   This example adds a set of signals.
 */
#define _OPEN_SYS
#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 38
- "sigaction() — Examine or Change a Signal Action" on page 749
- "sigdelset() — Delete a Signal from the Signal Mask" on page 757
- "sigemptyset() — Initialize a Signal Mask to Exclude All Signals" on page 758
- "sigfillset() — Initialize a Signal Mask to Include All Signals" on page 760
- "sigismember() — Test If a Signal Is in a Signal Mask" on page 763
- "signal() — Handle Interrupts" on page 767
- "sigpending() — Examine Pending Signals" on page 772
- "sigprocmask() — Examine or Change a Thread" on page 773
- "sigsuspend() — Change Mask and Suspend the Thread" on page 781

sigaltstack() — Set and/or Get Signal Alternate Stack Context

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

Allows a thread to define and examine the state of an alternate stack for signal handlers. If you explicitly declare that a signal execute on the alternate stack, it will be delivered on the alternate stack.

Note: To explicitly declare that a signal catcher run on the alternate signal stack, you must set the SA_ONSTACK flag in the sa_flags when you set the signal action with sigaction().

If *ss* is not a null pointer, it points to an alternate signal stack, `stack_t`, that will take effect upon return from sigaltstack(). The `ss_flags` member specifies the new stack state. If you set it 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 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, upon successful completion it will point to an alternate signal stack, `stack_t`, 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 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, no alternate signal stacks are in the new process image.

Returned Value

If successful, sigaltstack() returns a value of `0`.

If unsuccessful, sigaltstack() returns a value of `-1`, and sets errno to one of the following values under the conditions described:

EINVAL
The *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 38
- "sigaction() — Examine or Change a Signal Action" on page 749
- "sigsetjmp() — Save Stack Environment and Signal Mask" on page 778

sigdelset() — Delete a Signal from the Signal Mask

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

Format

```
#define _OPEN_SYS
#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()).

The sigdelset() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, sigdelset() returns a value of `0`.

If *signal* is not supported, sigdelset() returns a value of `-1` and sets errno to EINVAL.

Example

CBC3BS16

```
/* CBC3BS16
   This example deletes specific signals.
 */
#define _OPEN_SYS
#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 38
- “sigaction() — Examine or Change a Signal Action” on page 749
- “sigaddset() — Add a Signal to the Signal Mask” on page 754
- “sigemptyset() — Initialize a Signal Mask to Exclude All Signals” on page 758
- “sigfillset() — Initialize a Signal Mask to Include All Signals” on page 760
- “sigismember() — Test If a Signal Is in a Signal Mask” on page 763
- “signal() — Handle Interrupts” on page 767
- “sigpending() — Examine Pending Signals” on page 772
- “sigprocmask() — Examine or Change a Thread” on page 773

sigemptyset() — Initialize a Signal Mask to Exclude All Signals

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

Format

```
#define _OPEN_SYS
#include <signal.h>

int sigemptyset(sigset_t *set);
```

General Description

Initializes a signal set *set* to the empty set. All recognized signals are excluded.

The sigemptyset() 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. 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()).

The sigemptyset() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, sigemptyset() returns a value of 0.

No errnos are defined for sigemptyset().

Example

CBC3BS17

```
/* CBC3BS17
   This example initializes a set of signals to an empty set.
 */
#define _OPEN_SYS
#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 38
- "sigaction() — Examine or Change a Signal Action" on page 749
- "sigaddset() — Add a Signal to the Signal Mask" on page 754
- "sigdelset() — Delete a Signal from the Signal Mask" on page 757
- "sigfillset() — Initialize a Signal Mask to Include All Signals" on page 760

- “sigismember() — Test If a Signal Is in a Signal Mask” on page 763
- “signal() — Handle Interrupts” on page 767
- “sigpending() — Examine Pending Signals” on page 772
- “sigprocmask() — Examine or Change a Thread” on page 773
- “sigsuspend() — Change Mask and Suspend the Thread” on page 781

sigfillset() — Initialize a Signal Mask to Include All Signals

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

Format

```
#define _OPEN_SYS
#include <signal.h>

int sigfillset(sigset_t *set);
```

General Description

Initializes a signal set *set* to the complete set of supported signals.

The sigfillset() 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. 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()).

The sigfillset() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

If successful, sigfillset() returns a value of `0`.

No errnos are defined for sigfillset().

Example

CBC3BS18

```
/* CBC3BS18
   This example initializes a set of signals to a complete set.
 */
#define _OPEN_SYS
#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 38
- “sigaction() — Examine or Change a Signal Action” on page 749
- “sigaddset() — Add a Signal to the Signal Mask” on page 754
- “sigdelset() — Delete a Signal from the Signal Mask” on page 757
- “sigemptyset() — Initialize a Signal Mask to Exclude All Signals” on page 758
- “sigismember() — Test If a Signal Is in a Signal Mask” on page 763
- “signal() — Handle Interrupts” on page 767
- “sigpending() — Examine Pending Signals” on page 772
- “sigprocmask() — Examine or Change a Thread” on page 773
- “sigsuspend() — Change Mask and Suspend the Thread” on page 781

sighold() — Add a Signal to a Thread

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

Format

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

int sighold(int sig);
```

General Description

Provides a simplified method for adding the signal, *sig*, to the calling thread's signal mask.

Returned Value

If successful, sighold() returns a value of 0.

If unsuccessful, sighold() returns a value of -1 and sets errno to the following value:

EINVAL
The value of *sig* is not a valid signal type, or it is SIGKILL or SIGSTOP.

Related Information

- “signal.h” on page 38
- “sigaddset() — Add a Signal to the Signal Mask” on page 754
- “sigpause() — Unblock a Signal and Wait for a Signal” on page 771
- “sigprocmask() — Examine or Change a Thread” on page 773
- “sigrelse() — Remove a Signal from a Thread” on page 776

sigignore() — Set Disposition to Ignore a Signal

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON)

Format

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

int sigignore(int sig);
```

General Description

Provides a simplified method for setting the signal action of the signal, *sig*, to SIG_IGN.

Returned Value

If successful, sigignore() returns a value of `0`.

If unsuccessful, sigignore() returns a value of -1 and sets errno to the following value:

EINVAL
The value of *sig* is not a valid signal type, or it is SIGKILL or SIGSTOP.

Related Information

- "signal.h" on page 38
- "bsd_signal() — Establish BSD Version of signal()" on page 99
- "sigaction() — Examine or Change a Signal Action" on page 749
- "signal() — Handle Interrupts" on page 767
- "sigset() — Change a Signal Action and/or a Thread" on page 776

siginterrupt() — Allow Signals to Interrupt Functions

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON)

Format

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

int siginterrupt(int sig, int flag);
```

General Description

Provides a simplified method for changing the restart behavior when a function is interrupted by the signal, *sig*.

The *flag* argument serves as a binary switch to enable or disable restart behavior. When *flag* is nonzero, restart behavior is disabled; otherwise, it is enabled.

Returned Value

If successful, siginterrupt() returns a value of `0`.

If unsuccessful, siginterrupt() returns a value of -1 and sets errno to the following value:

EINVAL
The value of *sig* is not a valid signal type.

Related Information

- "signal.h" on page 38
- "sigaction() — Examine or Change a Signal Action" on page 749

sigismember() — Test If a Signal Is in a Signal Mask

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

Format

```
#define _OPEN_SYS
#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*.

The sigismember() 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. 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()).

The sigismember() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If *signal* is in *set*, sigismember() returns a value of 1; if not, sigismember() returns a value of 0. If *signal* is not a valid signal, sigismember() returns a value of -1.

If unsuccessful, sigismember() sets errno to EINVAL, which indicates that the value of *signal* is not one of the supported signals.

Example

CBC3BS19

```
/* CBC3BS19  */
#define _OPEN_SYS
#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 38
- “sigaction() — Examine or Change a Signal Action” on page 749
- “sigaddset() — Add a Signal to the Signal Mask” on page 754
- “sigdelset() — Delete a Signal from the Signal Mask” on page 757
- “sigemptyset() — Initialize a Signal Mask to Exclude All Signals” on page 758
- “sigfillset() — Initialize a Signal Mask to Include All Signals” on page 760
- “signal() — Handle Interrupts” on page 767
- “sigpending() — Examine Pending Signals” on page 772
- “sigprocmask() — Examine or Change a Thread” on page 773
- “sigsuspend() — Change Mask and Suspend the Thread” on page 781

siglongjmp() — Restore the Stack Environment and Signal Mask

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

Format

```
#define _OPEN_SYS
#include <setjmp.h>

void siglongjmp(sigjmp_buf env, int val);
```

General Description

Restores a stack environment previously saved in *env* by sigsetjmp() 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`.

The *env* argument is an address for a `sigjmp_buf` structure. The *value* argument is the return value from siglongjmp().

The siglongjmp() function is similar to the longjmp() function, except for the optional capability of restoring the signal mask. The setjmp()—siglongjmp() pair and the setjmp()—longjmp() pair cannot be intermixed. A stack environment and signal mask saved by setjmp() can be restored only by siglongjmp().

A call to setjmp() saves the current stack environment including, optionally, the signal mask, in *env*. A subsequent call to siglongjmp() restores the saved environment and signal mask (if saved by setjmp()) 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*. 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, as are nonvolatile auto variables that change between calls to setjmp() and siglongjmp().

Note: You should ensure that the function that calls setjmp() does not return before you call the corresponding siglongjmp() function. Calling siglongjmp() after the function calling setjmp() returns, causes unpredictable program behavior.

The *value* argument passed to siglongjmp() must be nonzero. If you give a 0 argument for *value*, siglongjmp() substitutes a value of 1 in its place.

The siglongjmp() function does not use the normal function call and return mechanisms. The siglongjmp() function restores the saved signal mask only if the *env* parameter was initialized by a call to setjmp() with a nonzero *savemask* argument.

The siglongjmp() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Special Behavior for C++

If sigsetjmp() and siglongjmp() are used to transfer control in a z/VM C++ program, the behavior is undefined in terms of the destruction of automatic objects. Additionally, if any automatic objects would be destroyed by a thrown exception transferring control to another (destination) point in the program, then a call to siglongjmp() at the throw point that transfers control to the same (destination) point has undefined behavior. This applies to both z/VM C++ and C/VM ILC modules. The use of sigsetjmp() and siglongjmp() in conjunction with try(), catch(), and throw() is also undefined.

Special Behavior for XPLINK-compiled C++

Restrictions concerning setjmp.h and ucontext.h:

Note:

1. All XPLINK programs compiled with the V2.10 or later C compilers that are to run with Language Environment V2.10 or later libraries and use the **jmp_buf**, **sigjmp_buf** or **ucontext_t** types must not be compiled with C headers from Language Environment V2.9 or earlier.
2. Non-XPLINK functions compiled with any level of Language Environment headers must not define **jmp_buf**, **sigjmp_buf** or **ucontext_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), _longjmp(), setjmp(), _setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.
3. When __XPLINK__ is defined, the Language Environment V2.10 and later headers define a larger **jmp_buf**, **sigjmp_buf** or **ucontext_t** area that is required by setjmp(), getcontext(), and related functions when they are called from an XPLINK routine. If __XPLINK__ is not defined, the Language Environment V2.10 and later headers define a shorter **jmp_buf**, **sigjmp_buf** or **ucontext_t** area. The Language Environment headers prior to V2.10 also define the shorter version of these data areas. If an XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp_buf**, **sigjmp_buf** or **ucontext_t** area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.

Returned Value

The siglongjmp() function has no returned value.

No errnos are defined for siglongjmp().

Example 2

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 setjmp() returns a value of `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 setjmp() 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.

Example 1

```
#define _OPEN_SYS
#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 38
- “longjmp() — Restore Stack Environment” on page 465
- “setjmp() — Preserve Stack Environment” on page 718
- “sigaction() — Examine or Change a Signal Action” on page 749
- “sigprocmask() — Examine or Change a Thread” on page 773
- “sigsetjmp() — Save Stack Environment and Signal Mask” on page 778
- “sigsuspend() — Change Mask and Suspend the Thread” on page 781

signal() — Handle Interrupts

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 26 on page 769).

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"`.

Table 25. Control Signals Supported Under C only — POSIX(ON)

Value	Default Action	Meaning
SIGALRM	1	A timeout signal (sent by alarm())
SIGHUP	1	A controlling terminal is hung up, or the controlling process ended
SIGKILL	1	A termination signal that cannot be caught or ignored
SIGPIPE	1	A write to a pipe that is not being read
SIGQUIT	1	A quit signal for a terminal
SIGTRAP	1	Internal for use by dbx or ptrace
SIGCHLD	2	An ended or stopped child process. (SIGCLD is an alias name for this signal.)
SIGDCE	2	Signal is used by DCE.
SIGIO	2	Completion of input or output
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

Table 25. Control Signals Supported Under C only — POSIX(ON) (continued)

Value	Default Action	Meaning
SIGTTOU	3	A background process attempted to write to a controlling terminal
SIGCONT	4	If stopped, continue

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 will 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);
```

You should make these calls either in the source or from the HLL user exit CEEBINT, which will require a re-link.

For an OpenExtensions 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().

For more information about using POSIX support, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

`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 749.

The *sig* argument must be one of the macros defined in the signal.h header file.

The signals supported are listed below.

Table 26. Control Signals Supported Under C — POSIX(OFF)

Value	Default Action	Meaning
SIGABND	1	Abend
SIGABRT	1	Abnormal termination (sent by abort())
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.

Default Actions

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.

Returned Value

If successful, signal() returns the most recent value of *func*.

If unsuccessful, signal() returns a value of SIG_ERR and a positive value in errno.

No errnos are defined for signal(). If an error occurs, you should 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();
  }
}

void StrCln(int);

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

- For information about signal handling, see *z/OS: XL C/C++ Programming Guide*.
- “signal.h” on page 38
- “abort() — Stop a Program” on page 66
- “atexit() — Register Program Termination Function” on page 85
- “exit() — End Program” on page 197
- “kill() — Send a Signal to a Process” on page 439
- “pthread_kill() — Send a Signal to a Thread” on page 586
- “raise() — Raise Signal” on page 623

- “sigaction() — Examine or Change a Signal Action” on page 749
- “sigprocmask() — Examine or Change a Thread” on page 773

__signgam() — Return __signgam Reference

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	

Format

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

int *__signgam(void);
#define signgam (*__signgam())
```

General Description

Returns the address of the calling thread's storage for the *signgam* external variable used by gamma() and lgamma(). Multithreaded processes that use either of these two functions need this extended mechanism 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 you eliminate this definition, either by not including the header or by using #undef, references to *signgam* will refer to the actual *signgam* external variable, which contains the *signgam* value for the IPT only. Also, the omission of this definition to a call to __signgam() will make *signgam* values in threads other than the IPT inaccessible.

Returned Value

The __signgam() function is always successful.

Related Information

- “math.h” on page 32
- “gamma() — Calculate Gamma Function” on page 310
- “lgamma() — Log Gamma Function” on page 448

sigpause() — Unblock a Signal and Wait for a Signal

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON)

Format

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

int sigpause(int sig);
```

General Description

Provides a simplified method for removing a signal, *sig*, from the calling thread's 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 a value of -1 and sets errno to EINTR.

If unsuccessful, sigpause() returns a value of -1, and sets errno to the following value:

EINVAL
The value of *sig* is not a valid signal type, or it is SIGKILL.

Related Information

- "signal.h" on page 38
- "pause() — Suspend a Thread Pending a Signal" on page 535
- "sigprocmask() — Examine or Change a Thread" on page 773
- "sigsuspend() — Change Mask and Suspend the Thread" on page 781

sigpending() — Examine Pending Signals

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

Format

```
#define _OPEN_SYS
#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 763.

The sigpending() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, sigpending() returns a value of 0.

If unsuccessful, sigpending() returns a value of -1.

No errnos are defined for sigpending().

Example

CBC3BS22

```
/* CBC3BS22
   This example returns blocked or pending signals.
 */
#define _OPEN_SYS
#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 38
- “sigismember() — Test If a Signal Is in a Signal Mask” on page 763
- “sigprocmask() — Examine or Change a Thread” on page 773

sigprocmask() — Examine or Change a Thread

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

Format

```
#define _OPEN_SYS
#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.

The sigprocmask() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

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, ..., ...)`.

`int` *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.

`sigset_t *` *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 758. If *new_set* is a NULL pointer, the set of blocked signals is not changed. The sigprocmask() function 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*.

`sigset_t *`*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 any pending unblocked signals exist, 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. The sigprocmask() function does not return an error status.

SIGFPE, SIGILL, and SIGSEGV signals that are not artificially generated by kill(), 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 sigprocmask() fails, the signal mask of the thread is not changed.

Returned Value

If successful, sigprocmask() returns a value of `0`.

If unsuccessful, sigprocmask() returns a value of -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 _OPEN_SYS
#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 38
- “kill() — Send a Signal to a Process” on page 439
- “pthread_kill() — Send a Signal to a Thread” on page 586
- “raise() — Raise Signal” on page 623
- “sigaction() — Examine or Change a Signal Action” on page 749
- “sigaddset() — Add a Signal to the Signal Mask” on page 754
- “sigdelset() — Delete a Signal from the Signal Mask” on page 757
- “sigemptyset() — Initialize a Signal Mask to Exclude All Signals” on page 758
- “sigfillset() — Initialize a Signal Mask to Include All Signals” on page 760
- “sigismember() — Test If a Signal Is in a Signal Mask” on page 763
- “signal() — Handle Interrupts” on page 767

- “sigpending() — Examine Pending Signals” on page 772
- “sigsuspend() — Change Mask and Suspend the Thread” on page 781

sigrelse() — Remove a Signal from a Thread

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

Format

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

int sigrelse(int sig);
```

General Description

Provides a simplified method for removing the signal, *sig*, from the calling thread's signal mask.

Returned Value

If successful, sigrelse() returns a value of 0.

If unsuccessful, sigrelse() returns a value of -1 and sets errno to the following value:

EINVAL
The value of *sig* is not a valid signal type, or it is SIGKILL or SIGSTOP.

Related Information

- “signal.h” on page 38
- “sighold() — Add a Signal to a Thread” on page 761
- “sigpause() — Unblock a Signal and Wait for a Signal” on page 771
- “sigprocmask() — Examine or Change a Thread” on page 773

sigset() — Change a Signal Action and/or a Thread

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON)

Format

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

void  (*sigset(int sig, void (*disp)(int)))(int);
```

General Description

Provides a simplified method for changing the action associated with a specific signal and unblocking or blocking this signal.

int *sig*
The number of a recognized signal. The sigset() function sets the action associated with this signal and unblocks this signal or adds it to the calling thread's signal mask (thus blocking this signal). For a list of the supported values of *sig*, see Table 24 on page 750.

The value of *sig* can be any valid signal type except SIGKILL or SIGSTOP.

void (*)(int) *disp*
The *disp* argument can have four possible values, three of which are actions that can be associated with the signal, *sig*: SIG_DFL, SIG_IGN, or a pointer to a function and the fourth not being a signal action, but a flag to sigset() that affects whether the signal action is changed.

The permissible values for *disp* are as follows:

SIG_DFL
Sets the signal action to the signal-specific default.

- The default actions for each signal is shown in Table 24 on page 750.
- 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, its execution 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 pending signal whose default action is to ignore the signal (for example SIGCHLD), will cause the pending signal to be discarded.

SIG_IGN
Sets 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 pending signal will cause the pending signal to be discarded. This setting provides a means of discarding signals that sigpending() finds blocked and pending.
- 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 no children were waiting to be 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
Sets the calling thread's signal mask to block *sig*.

- The signal action associated with *sig* is not changed.

Pointer to function
Sets the signal action to catch the signal.

- The sigset() function will change the signal action associated with *sig* and remove this signal from the calling thread's signal mask (thus unblocking this signal).
- Upon delivery of the signal, the receiving process will 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 you use sigset() to specify an action for a particular signal, the action remains installed until you explicitly change it by calling 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 sets errno to the following value:

EINVAL
The value of *sig* was not a valid signal type, or it was SIGKILL.

Related Information

- “signal.h” on page 38
- “bsd_signal() — Establish BSD Version of signal()” on page 99
- “sigaction() — Examine or Change a Signal Action” on page 749
- “sighold() — Add a Signal to a Thread” on page 761
- “signal() — Handle Interrupts” on page 767
- “sigprocmask() — Examine or Change a Thread” on page 773

sigsetjmp() — Save Stack Environment and Signal Mask

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

Format

```
#define _OPEN_SYS
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);
```

General Description

Saves the current stack environment including, optionally, the current signal mask, both of which can subsequently be restored by siglongjmp().

The *env* argument is an address for a `sigjmp_buf` structure; *savemask* is a flag used to determine if the signal mask should be saved. If it has a value of `0`, the current signal mask should not be saved or restored as part of the environment. Any other value means the current signal mask is saved and restored.

The sigsetjmp() function is similar to 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 and the setjmp()—longjmp() pair cannot be intermixed. A stack environment and signal mask saved by setjmp() can be restored only by siglongjmp().

A call to sigsetjmp() saves 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. as are nonvolatile auto variables that are changed between calls to sigsetjmp() and siglongjmp().

The sigsetjmp() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Special Behavior for C++

If sigsetjmp() and siglongjmp() are used to transfer control in a z/VM C++ program, the behavior in terms of the destruction of automatic objects is undefined. This applies to both z/VM C++ and C/VM ILC modules. The use of sigsetjmp() and siglongjmp() in conjunction with try(), catch(), and throw() is also undefined.

Special Behavior for XPLINK-compiled C++

Restrictions concerning setjmp.h and ucontext.h:

Note:

1. All XPLINK programs compiled with the V2.10 or later C compilers that are to run with Language Environment V2.10 or later libraries and use the **jmp_buf**, **sigjmp_buf** or **ucontext_t** types must not be compiled with C headers from Language Environment V2.9 or earlier.
2. Non-XPLINK functions compiled with any level of Language Environment headers must not define **jmp_buf**, **sigjmp_buf** or **ucontext_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), _longjmp(), setjmp(), _setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.
3. When __XPLINK__ is defined, the Language Environment V2.10 and later headers define a larger **jmp_buf**, **sigjmp_buf** or **ucontext_t** area that is required by setjmp(), getcontext(), and related functions when they are called from an XPLINK routine. If __XPLINK__ is not defined, the Language Environment V2.10 and later headers define a shorter **jmp_buf**, **sigjmp_buf** or **ucontext_t** area. The Language Environment headers prior to V2.10 also define the shorter version of these data areas. If an XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp_buf**, **sigjmp_buf** or **ucontext_t** area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.

Returned Value

Returns a value of `0` when it is invoked to save the stack environment and signal mask. The sigsetjmp() function returns the value *val*, specified on siglongjmp() (or 1 if *val* is `0`), when siglongjmp() causes control to be transferred to the place in the user's program where sigsetjmp() was issued.

No errnos are defined for sigsetjmp().

Example

The following 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 a value of `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 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 _OPEN_SYS
#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 38
- “longjmp() — Restore Stack Environment” on page 465
- “setjmp() — Preserve Stack Environment” on page 718
- “sigaction() — Examine or Change a Signal Action” on page 749
- “siglongjmp() — Restore the Stack Environment and Signal Mask” on page 764
- “sigprocmask() — Examine or Change a Thread” on page 773
- “sigsuspend() — Change Mask and Suspend the Thread” on page 781

sigstack() — Set and/or Get Signal Stack Context

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

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 run on the alternate signal stack, you must set the SA_ONSTACK flag in the sa_flags when you set the signal action with sigaction().

If *ss* 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 *ss* is a null pointer, the current alternate signal stack context is not changed.

If *oss* 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 *oss* 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, no alternate signal stacks are in the new process image.

Returned Value

If successful, sigstack() returns a value of `0`.

If unsuccessful, sigstack() returns a value of -1, and sets errno to the following value:

EPERM
An attempt was made to modify an active stack.

Related Information

- “signal.h” on page 38
- “sigaction() — Examine or Change a Signal Action” on page 749
- “sigaltstack() — Set and/or Get Signal Alternate Stack Context” on page 756
- “sigsetjmp() — Save Stack Environment and Signal Mask” on page 778

sigsuspend() — Change Mask and Suspend the Thread

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	POSIX(ON)

Format

```
#define _OPEN_SYS
#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 758.)

The signal mask indicates a set of signals that should be blocked. Such signals do not "wake up" the suspended function. The SIGSTOP and SIGKILL signals cannot be blocked or ignored; they are delivered to the thread no matter what *mask* 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.

The sigsuspend() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If sigsuspend() returns, it always returns a value of -1.

If 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 _OPEN_SYS
#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",
                         signum);
  }
}

main() {
  sigset_t sigset;
  struct sigaction sact;
  time_t   t;

  if ((pid = spawn("CBC3BS2A", 0, 0, 0, 0, 0)) == -1)
    perror("spawn() error");
  else {
    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));
    }
  }
}


/* CBC3BS2A
   This is the child.
 */
#define _OPEN_SYS
#include <stdio.h>
#include <signal.h>
#include <time.h>
#include <unistd.h>

main() }

  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);

}
```

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 38
- “kill() — Send a Signal to a Process” on page 439
- “pause() — Suspend a Thread Pending a Signal” on page 535
- “pthread_kill() — Send a Signal to a Thread” on page 586
- “raise() — Raise Signal” on page 623
- “sigaction() — Examine or Change a Signal Action” on page 749
- “sigaddset() — Add a Signal to the Signal Mask” on page 754
- “sigdelset() — Delete a Signal from the Signal Mask” on page 757
- “sigemptyset() — Initialize a Signal Mask to Exclude All Signals” on page 758
- “sigfillset() — Initialize a Signal Mask to Include All Signals” on page 760
- “signal() — Handle Interrupts” on page 767
- “sigpending() — Examine Pending Signals” on page 772
- “sigprocmask() — Examine or Change a Thread” on page 773

sigwait() — Wait for an Asynchronous Signal

Standards / Extensions	C or C++	Dependencies
POSIX.1	both	

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*, atomically 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.

The sigwait() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, sigwait() returns the signal number.

If unsuccessful, sigwait() returns a value of -1 and sets errno to EINVAL, which indicates that *set* 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 38
- "pause() — Suspend a Thread Pending a Signal" on page 535
- "sigaction() — Examine or Change a Signal Action" on page 749
- "sigpending() — Examine Pending Signals" on page 772
- "sigprocmask() — Examine or Change a Thread" on page 773
- "sigsuspend() — Change Mask and Suspend the Thread" on page 781

sin() — Calculate Sine

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

Format

```
#include <math.h>

double sin(double x);
```

General Description

Calculates the sine of *x*, with *x* expressed in radians.

Note: These functions work in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

Returned Value

If successful, sin() returns the calculated value, expressed as a double float.

If the result underflows, it returns a value of `0` and sets 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 32
- “acos(), acosf(), acosl() — Calculate Arccosine” on page 73
- “asin(), asinf(), asinl() — Calculate Arcsine” on page 79
- “atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate Arctangent” on page 83
- “cos(), cosf(), cosl()— Calculate Cosine” on page 143
- “cosh(), coshf(), coshl()— Calculate Hyperbolic Cosine” on page 145
- “sinh() — Calculate Hyperbolic Sine” on page 786
- “tan() — Calculate Tangent” on page 861
- “tanh() — Calculate Hyperbolic Tangent” on page 862

sinh() — Calculate Hyperbolic Sine

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

Format

```
#include <math.h>

double sinh(double x);
```

General Description

Calculates the hyperbolic sine of *x*, with *x* expressed in radians.

Returned Value

If successful, sinh() returns the calculated value.

If the result is too large, sinh() returns HUGE_VAL (plus or minus, depending on the value of *x*) and sets errno to ERANGE. If the value underflows, sinh() returns a value of `0` and sets errno to ERANGE.

Note: These functions work in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

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 32
- "acos(), acosf(), acosl() — Calculate Arccosine" on page 73
- "asin(), asinf(), asinl() — Calculate Arcsine" on page 79
- "atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate Arctangent" on page 83
- "cos(), cosf(), cosl()— Calculate Cosine" on page 143
- "cosh(), coshf(), coshl()— Calculate Hyperbolic Cosine" on page 145
- "sin() — Calculate Sine" on page 785
- "tan() — Calculate Tangent" on page 861
- "tanh() — Calculate Hyperbolic Tangent" on page 862

sleep() — Suspend Execution of a Process

Standards / Extensions	C or C++	Dependencies
POSIX.1 XPG4 XPG4.2	both	POSIX(ON)

Format

```
#define _OPEN_SYS
#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 sleep time remains.

The sleep() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

Returns a value of `0` 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).

The sleep() function is always successful; it has no failure return. An abend is generated when any failures are encountered that prevent sleep() from completing successfully.

No errnos are defined for sleep().

Example

CBC3BS29

```
/* CBC3BS29
   This example suspends execution for a specified time.
 */
#define _OPEN_SYS
#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 38
- “unistd.h” on page 51
- “alarm() — Set an Alarm” on page 76
- “kill() — Send a Signal to a Process” on page 439

- "longjmp() — Restore Stack Environment" on page 465
- "pause() — Suspend a Thread Pending a Signal" on page 535
- "pthread_kill() — Send a Signal to a Thread" on page 586
- "sigaction() — Examine or Change a Signal Action" on page 749
- "siglongjmp() — Restore the Stack Environment and Signal Mask" on page 764
- "signal() — Handle Interrupts" on page 767
- "sigprocmask() — Examine or Change a Thread" on page 773
- "sigsuspend() — Change Mask and Suspend the Thread" on page 781

sock_debug() — Provide OE Syscall Tracing Facility

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>
void sock_debug(int onoff);
```

General Description

Provides the UNIT tracing facility. The *onoff* parameter can have a value of `0` or nonzero. If *onoff* is `0` (the default), no UNIT tracing is done. If *onoff* is nonzero, 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 SOCKDEBUG statement in the TCPIP DATA file. When the process is started, all the programs will begin the UNIT trace and report the progress to the `stderr` file.

onoff
A parameter that can be set to zero or nonzero.

Returned Value

The sock_debug() function has no returned value.

Related Information

- "sys/socket.h" on page 46

sock_debug_bulk_perf0() — Produce a Report When a Socket Configured

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>
void sock_debug_bulk_perf0(int onoff);
```

General Description

Collects and uses data to produce a report when a socket is configured in bulk mode. The report is written to `stderr` when the bulk mode socket is closed.

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 14601460 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 SOCKDEBUGBULKPERF0 statement in the TCPIP DATA file. When the process is started, all the programs using bulk mode sockets for this process will produce a report.

Returned Value

The sock_debug_bulk_perf0() function has no returned value.

Related Information

- “sys/socket.h” on page 46

sock_do_bulkmode() — Use Bulk Mode for Messages Read by Socket

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>
void sock_do_bulkmode(int onoff);
```

General Description

Uses bulk mode for messages read by the socket program.

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.

Using bulk mode can improve program performance, which 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 datagram sockets are processed, an even greater reduction in CPU usage should occur.

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 `0` 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 SOCKBULKMODE statement in the TCPIP DATA file. When the process is started, all the programs using datagram sockets will begin running in bulk mode for reads on the socket.

Returned Value

The sock_do_bulkmode() function has no returned value.

Related Information

- "sys/socket.h" on page 46

sock_do_teststor — Check for Attempt to Access Storage

Standards / Extensions	C or C++	Dependencies
z/VM OpenExtensions	both	

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>
void sock_do_teststor(int onoff);
```

General Description

Checks for calls that attempt to access storage outside the caller's virtual machine.

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, the sock_do_teststor() function checks both the address of the message buffer and the message buffer for addressability for each bulk mode socket call. If an addressing problem exists, sock_do_teststor() sets errno to EFAULT.

If *onoff* is set to `0`, the socket library program does not check addresses. If an error occurs when *onoff* is `0`, normal runtime 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 SOCKTESTSTOR statement in the TCPIP DATA file. 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

The sock_do_teststor() function has no returned value.

Related Information

- "sys/socket.h" on page 46

socket() — Create a Socket

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

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

Creates an endpoint for communication and returns a socket descriptor representing the endpoint. Different types of sockets provide different communication services.

domain
The address domain requested, either AF_INET, AF_INET6, AF_IUCV, or AF_UNIX.

type
The type of socket created, either SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW.

protocol
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 and AF_INET6, which is the internet domain; AF_IUCV, which is the VM IUCV domain; and AF_UNIX, which is the local socket domain. These constants are defined in the sys/socket.h header file.

The *type* parameter specifies the type of socket created. The type is compatible with the semantics of the communication requested. These socket type constants are defined in the sys/socket.h header file. The types supported are as follows:

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 the AF_INET, AF_INET6, 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 the AF_INET, AF_INET6, and AF_UNIX domains.

SOCK_RAW
Provides the interface to internal protocols (such as IP and ICMP). This type is supported in the AF_INET and AF_INET6 domains.

For more information on building C applications which use any sockets on CMS, see *z/VM: TCP/IP Programmer's Reference*.

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 ETC PROTO file. Alternatively, you can use the getprotobyname() call to get the protocol number for a protocol with a known name.

Note:

1. The *protocol* field *must* be set to `0` if the *domain* parameter is set to AF_UNIX.
2. Protocols 0, 41, 43, 44, 50, 51 59, and 60 are not valid for AF_INET6 raw sockets.

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 and AF_INET6 domains, applying the bind() call 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. For AF_INET sockets, this is done by setting the *Internet address* field in the sockaddr_in structure to the constant INADDR_ANY, as defined in **<netinet/in.h>**. For AF_INET6 sockets, this is done by setting the *Internet address* field in the sockaddr_in6 structure to IN6ADDR_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 writev()). Usually, the read()-write() or send()-recv() pairs are used for sending data on stream sockets. To exchange out-of-band data, 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 only supported in the AF_INET and AF_INET6 domains.

Datagram sockets have no active or passive analogy to stream sockets. Servers must still call bind() to name a socket and to specify from which network interfaces they wish 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 application program can also use the other data transfer calls read(), write(), readv(), writev(), send(), and recv(). For more information on placing a socket into the connected state, see "connect() — Connect a Socket" on page 138.

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). You can use the INADDR_BROADCAST constant, defined in netinet/in.h, to broadcast to the primary network if the primary network configured supports broadcast.

Outgoing packets have an IP header prefixed to them. You can set and inspect IP options using the setsockopt() and getsockopt() calls, respectively. Incoming packets are received with the IP header and options intact.

SOCK_RAW sockets give the application an interface to lower layer protocols, such as IP and ICMP. This interface is often used to bypass the transport layer when direct access to lower layer protocols is needed. Raw sockets are also used to test new protocols. Raw sockets are not supported in the AF_IUCV and AF_UNIX domains.

Raw sockets are connectionless and data transfer semantics are the same as those described previously for datagram sockets. The connect() call can be used similarly to specify the peer. 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

If successful, socket() returns a nonnegative socket descriptor.

If unsuccessful, socket() returns a value of -1 and sets errno to one of the following values under the conditions described:

EACCES
Permission to create a socket of the specified type or protocol is denied.

EAFNOSUPPORT
The address family is not supported (it is not AF_UNIX, AF_INET, or AF_INET6).

EAGAIN
The resource is temporarily unavailable.

EINVAL
The request is invalid or not supported.

EIO
A network or transport failure has occurred.

ENOBUFS
Insufficient system resources are available to complete the call.

EPROTOTYPE
The socket type is not supported by the protocol.

Example

The following are examples of the socket() call.

```
int s;
⋮
/* 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 69
- “bind() — Bind a Name to a Socket” on page 92
- “close() — Close a File” on page 127
- “connect() — Connect a Socket” on page 138
- “fcntl() — Control Open File Descriptors” on page 207
- “getprotobyname() — Get a Protocol Entry by Name” on page 358
- “getsockname() — Get Name of a Socket” on page 368
- “getsockopt() — Get Options Associated with a Socket” on page 370
- “ioctl() — Control Device” on page 417
- “read() — Read From a File ” on page 628
- “readv() — Read Data on a Socket and Store in a Set of Buffers” on page 635

- "recv() — Receive Data on a Socket" on page 640
- "recvfrom() — Receive Messages on a Socket" on page 642
- "recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers" on page 645
- "select() — Monitor Activity on Files/Sockets and Message Queues" on page 680
- "selectex() — Monitor Activity on Files/Sockets and Message Queues" on page 685
- "send() — Send Data on a Socket" on page 694
- "sendmsg() — Send Messages on a Socket" on page 697
- "shutdown() — Shut Down All or Part of a Duplex Connection" on page 748
- "write() — Write Data on a File or Socket" on page 975
- "writev() — Write Data on a Socket from an Array" on page 979

spawn() - spawnp() — Spawn a New Process

Standards / Extensions	C or C++	Dependencies
POSIX.4b z/VM OpenExtensions	both	

Standard
This function(s) is defined in the draft standard POSIX.4b.

Format

```
#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

Creates a new process from the specified regular executable file called the new process image file.

When a C program is executed as a result of this call, it is entered as a C language 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*.

The *path* argument to spawn() is a path name that identifies the new process image file to execute.

The *file* parameter to spawnp() is used to construct a path name that identifies the new process image file. If the file parameter contains a slash character, the file parameter is used as a path name for the new process image file. Otherwise, the path prefix for this file is obtained by searching the directories passed as the environment variable PATH.

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 image are closed in the new process image, with the effect of the closedir() operation.

The *fd_count* argument 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 inherit descriptor *fd_map*(*x*) from the parent process.

If an element of *fd_map* refers to an invalid file descriptor, then the (EBADF) error status is posted by spawn() or spawnp().

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.

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.

If the SPAWN_SETGROUP flag is set in `inherit.flags`, then the child's process group is the one 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 created 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.

If the SPAWN_SETSIGMASK flag is set in inherit.flags, the child process initially has the signal mask specified in `inherit.sigmask`.

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`.

If the SPAWN_SETTCPGRP flag is set in `inherit.flags`, the file descriptor specified in `inherit.ctlttyfd` is used to set the controlling terminal file descriptor (tcsetgrp()) 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` should be used to determine the child's process group.

The value in the first element of *argv* should point to a file name 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}.

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() and setgid() functions.

The new process image inherits the following attributes from the calling process image:

1. Process group ID (unless the SPAWN_SETGROUP flag is set in `inherit.flags`)
2. Session membership
3. Real user ID
4. Real group ID
5. Supplementary group IDs
6. Priority
7. Current working directory
8. Root directory
9. File creation mask
10. Signal mask (unless the SPAWN_SETSIGMASK flag is set in `inherit.flags`)
11. Signal actions specified as default (SIG_DFL)
12. 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:

1. Signals set to be caught by the calling process are set to the default action (SIG_DFL).
2. The process and system utilization times for the child are set to zero.
3. Any file locks previously set by the parent are not inherited by the child.
4. The child process has no alarms set.
5. The child process has no interval timers set.
6. The child has no pending signals.
7. 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 marks the `st_atime` field of the new process image file for update.

If the spawn() or spawnp() function is successful, the new child process image file is opened, with all the effects of the open() function.

Returned Value

If successful, spawn() and spawnp() return the value of the process ID of the child process to the parent process.

If unsuccessful, no child process is created, and spawn() and spawnp() return a value of -1 and set errno to one of the following values under the conditions described:

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.

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.

EBADF
One of the following conditions applies:

- An entry in the "fd_map" array refers to:
  - An invalid file descriptor

  - A sockets file descriptor
  - An external link file descriptor
- The controlling terminal file descriptor specified in `inherit.ctlttyfd` is not valid.

ECMSERR
An unexpected system error occurred, often the result of being out of storage.

EFAULT
The system detected an invalid address in attempting to use a parameter of the call.

EINVAL
The process group ID specified in `inherit.pgroup` is less than zero or has some other unsupported value.

ELOOP
A loop exists in symbolic links encountered during resolution of the *file* argument. This error is issued if more than 8 symbolic links are detected in the resolution of file name.

EMFILE
The process has reached the maximum number of file descriptors that it can have open.

ENAMETOOLONG
The length of *path* or *file* or an element of the environment variable PATH prefixed to *file*, exceeds {PATH_MAX}, or a path name component is longer than {NAME_MAX} and {_POSIX_NO_TRUNC} is in effect for that file.

ENFILE
The system has reached the maximum number of file descriptors that it can have open.

ENOENT
One or more components of the path name of the new process image file do not exist, or the *path* or *file* argument is empty.

ENOEXEC
The new process image file has the appropriate access permissions, but is not in the proper format.

ENOMEM
The new process requires more memory than is permitted by the hardware or the operating system.

ENOTDIR
A component of the path prefix of the new process image file is not a directory.

ENOTTY
The tcsetgrp() function 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
The tcsetpgrp() function failed for the specified controlling terminal file descriptor in `inherit.ctlttyfd`. The failure occurred because the spawned process is not a process group leader.

ESRCH
The process group ID specified in `inherit.pgroup` is not that of a process group in the calling process's session.

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.

Related Information

- “spawn.h” on page 39

sprintf() — Format and Write Data to Buffer

The information for this function is included in "fprintf() - printf() - sprintf() — Format and Write Data" on page 264.

sqrt() — Calculate Square Root

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

Format

```
#include <math.h>

double sqrt(double x);
```

General Description

Calculates the square root of the positive value, *x*.

Note: These functions work in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

Returned Value

Returns the square root result. If *x* is negative, sqrt() returns a value of `0` and sets errno to EDOM. If the correct value would cause underflow, sqrt() returns a value of `0` and stores the value of the macro ERANGE 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 32
- "exp(), expf(), expl()— Calculate Exponential Function" on page 199
- "hypot() — Calculate Hypotenuse" on page 397
- "log(), logf(), logl() — Calculate Natural Logarithm" on page 460
- "log10(), log10f(), log10l() — Calculate Base 10 Logarithm" on page 462
- "pow() — Raise to Power" on page 543

srand() — Set Seed for rand() Function

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

Format

```
#include <stdlib.h>

void srand(unsigned int seed);
```

General Description

Uses *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.

You may find it convenient to use the return value of the time() function as the argument to srand(), to ensure random sequences of random numbers.

Returned Value

The srand() function has no returned 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 42
- “rand() — Generate Random Number” on page 626

srandom() — Use Seed to Initialize Generator for random()

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

Format

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

void srandom(unsigned seed);
```

General Description

Initializes the calling thread's current state array for random(), using the value of *seed*.

Returned Value

The srandom() function has no returned value.

Related Information

- “stdlib.h” on page 42
- “drand48() — Generate Pseudo-Random Number” on page 178
- “initstate() — Initialize Generator for random()” on page 415
- “rand() — Generate Random Number” on page 626
- “random() — Generate a Better Random Number” on page 627
- “setstate() — Change Generator for random()” on page 739

srand48() — Initialize Pseudorandom Number

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 lcong48(), seed48(), and srand48() functions are initialization functions. You should invoke one of these before calling either the drand48(), lrand48, or mrand48 function.

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 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)mod(2**48)     n>=0
```

The initial values of X, a, and c are:

```
X(0)= 1
a   = 5deece66d (base 16)
c   = b         (base 16)
```

IBM XL C/C++ for z/VM provides storage to save the most recent 48-bit integer value of the sequence, `X(i)`. The drand48(), lrand48(), and mrand48() functions share this storage. You may reinitialize the most recent 48-bit value in this storage by calling srand48(). The srand48() function replaces the high order (leftmost) 32 bits of this storage with the *seedval* argument value and the low order 16 bits of this storage with the value `330E` (base 16).

You may change the values of a and `c`, by calling lcong48(). The srand48() function restores the initial values of a and `c`.

Special Behavior for OpenExtensions Services

You can make 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, IBM XL C/C++ for z/VM serializes access to the storage for `X(n)`, a, and c by functions in the `drand48` family when a multithreaded application calls them.

If you request thread-specific behavior, making calls to drand48(), lrand48(), and mrand48() from thread `t` generates 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))mod(2**48)     n>=0
```

IBM XL C/C++ for z/VM provides thread-specific storage to save the most recent 48-bit integer value of the sequence, `X(t,i)`. When you call srand48() from thread `t`, srand48() reinitializes the most recent 48-bit value in this storage. The srand48() function replaces the high order (leftmost) 32 bits of this storage with the *seedval* argument value and the low order 16 bits of this storage with the value `330E` (base 16).

You may change the values of `a(t)` and `c(t)`, for the thread `t` by calling lcong48(). When you call srand48() 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

After srand48() has used the value of *seedval* to reinitialize 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 OpenExtensions Services

If you request thread-specific behavior for the `drand48` family and call srand48() on thread `t`, srand48() uses the value of the *seedval* argument 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 42
- "drand48() — Generate Pseudo-Random Number" on page 178
- "erand48() — Generate Pseudo-Random Number" on page 187
- "jrand48() — Generate Pseudorandom Number" on page 436
- "lcong48() — Initialize Pseudorandom Number" on page 444
- "lrand48() — Generate Pseudorandom Number" on page 467
- "mrand48() — Generate Pseudorandom Number" on page 510
- "nrand48() — Pseudo-Random Number Generator" on page 523
- "seed48() — Initialize Pseudorandom Number" on page 679

sscanf() — Read and Format Data from Buffer

The information for this function is included in "fscanf() - scanf() - sscanf() — Read and Format Data" on page 283.

stat() — Get File Information

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

Format

```
#define _OPEN_SYS
#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 *info*. 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.

The information is returned as shown in the following `stat` structure table, as defined in the sys/stat.h header file.

Table 27. 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 117.
`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.

The stat() function 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 mode.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_ISVMEXTL(*mode*)
Is nonzero for external links.

S_ISVMEXTL_EXEC(*mode*)
Is nonzero for CMS EXEC external links.

S_ISVMEXTL_DATA(*mode*)
Is nonzero for CMS DATA external links.

S_ISVMEXTL_MEL(*mode*)
Is nonzero for MOUNT external links.

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.

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.

For external links to CMS files, stat() functions as usual with the exception of how it determines the file size. For fixed-format files, the file size returned is the product of the number of records and the logical record length. For variable-format files, the size returned is the same as above if the file does not exceed one data block in length. If it is longer, the size is calculated as the product of the number of data blocks and the block size.

Returned Value

If successful, stat() returns a value of `0`.

If unsuccessful, stat() returns a value of -1 and sets errno to one of the following values under the conditions described:

EACCES
The process does not have search permission on some component of the *pathname* prefix.

EINVAL
info is a null pointer.

Or the path name starts with "//", which is not allowed for BFS files.

ELOOP
A loop exists in symbolic links encountered during resolution of *pathname*. The stat() function returns this error if it detects more than POSIX_SYMLOOP (defined in the limits.h header file) symbolic links during 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 with pathconf().

ENOENT
No file named *pathname* exists, or *pathname* is an empty string.

ENOTDIR
A component of the *pathname* prefix is not a directory.

Example

CBC3BS33

```
/* CBC3BS33
   This example gets status information about a file.
 */
#define _OPEN_SYS
#include <sys/types.h>
#include <sys/stat.h>
#undef _OPEN_SYS
#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:   11
   mode:   010001ed
  links:   11
    uid:   0
    gid:   500
created:   Fri Jun 16 10:07:55 1995
```

Related Information

- “sys/stat.h” on page 46
- “sys/types.h” on page 46
- “chmod() — Change the Mode of a File or Directory” on page 117
- “chown() — Change the Owner or Group of a File or Directory” on page 120
- “creat() — Create a New File or Rewrite an Existing One” on page 146
- “dup() — Duplicate an Open File Descriptor” on page 180
- “exec Functions” on page 192
- “fcntl() — Control Open File Descriptors” on page 207
- “fstat() — Get Status Information about a File” on page 296
- “link() — Create a Link to a File” on page 450
- “lstat() — Get Status of File, External Link, or Symbolic Link” on page 471
- “mkdir() — Make a Directory” on page 498
- “mkfifo() — Make a FIFO Special File” on page 499
- “open() — Open a File” on page 527
- “pipe() — Create an Unnamed Pipe” on page 539
- “read() — Read From a File ” on page 628
- “readlink() — Read the Value of a Symbolic Link” on page 633
- “remove() — Delete File” on page 659
- “symlink() — Create a Symbolic Link to a Path Name” on page 854
- “unlink() — Remove a Directory Entry” on page 910
- “utime() — Set File Access and Modification Times” on page 912
- “write() — Write Data on a File or Socket” on page 975

statvfs() — Get File System Information

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

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#include <sys/statvfs.h>
int statvfs(const char *pathname, struct statvfs *fsinfo);
```

General Description

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*.

The information is returned in the following `statvfs` structure, as defined in the sys/statvfs.h header file.

Table 28. 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 file name length.
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 the information 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 a value of `0`.

If unsuccessful, statvfs() returns a value of `-1`, and set errno to one of the following values under the conditions described:

EACCES
The process does not have search permission on a component of the path prefix.

EIO
An I/O error occurred while reading from the file system.

EINTR
A signal was caught during the execution of the function.

ELOOP
Too many symbolic links (more than the system-defined limit of 8) were encountered during resolution of the path name.

ENAMETOOLONG
The length of the path name exceeded PATH_MAX, or a component of the path name is longer than NAME_MAX.

ENOENT
No file named *pathname* exists, or *pathname* is an empty string.

ENOTDIR
A component of the path prefix of the path name 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 46
- “chmod() — Change the Mode of a File or Directory” on page 117
- “chown() — Change the Owner or Group of a File or Directory” on page 120
- “creat() — Create a New File or Rewrite an Existing One” on page 146
- “dup() — Duplicate an Open File Descriptor” on page 180
- “exec Functions” on page 192
- “fcntl() — Control Open File Descriptors” on page 207
- “link() — Create a Link to a File” on page 450
- “mknod() — Make a FIFO or Charcter Special File” on page 501
- “open() — Open a File” on page 527
- “pipe() — Create an Unnamed Pipe” on page 539

- “read() — Read From a File ” on page 628
- “time() — Determine Current Time” on page 885
- “unlink() — Remove a Directory Entry” on page 910
- “utime() — Set File Access and Modification Times” on page 912
- “write() — Write Data on a File or Socket” on page 975

strcasecmp() — Compare Case-Insensitive Strings

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

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.

Returned Value

Returns a value indicating the relationship between the strings, while ignoring case, as follows:

Value
Meaning

< 0
String pointed to by *string1* is less than string pointed to by *string2*.

= 0
String pointed to by *string1* is equal to string pointed to by *string2*.

> 0
String pointed to by *string1* is greater than string pointed to by *string2*.

No errnos are defined for strcasecmp().

Related Information

- “strings.h” on page 44
- “strncasecmp() — Compare Case-Insensitive Strings” on page 826
- “strcspn() — Compare Strings” on page 816

strcat() — Concatenate Strings

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

A built-in function that concatenates *string2* with *string1* and ends the resulting string with the null character. In other words, strcat() appends a copy of *string2*—including the terminating null byte—to the end of *string1*, with its last byte (that is, the terminating null byte of *string1*) overwritten by the first byte of the appended string.

Note: You should 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 43
- “strchr() — Search for Character” on page 811
- “strcmp() — Compare Strings” on page 812
- “strcpy() — Copy String” on page 815
- “strcspn() — Compare Strings” on page 816
- “strncat() — Concatenate Strings” on page 827

strchr() — Search for Character

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

A built-in function that finds the first occurrence of *c* (converted to a character) in **string*. The *c* character can be the null character (`\0`); the ending null character of *string* is included in the search.

Rule: 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

If successful, strchr() returns a pointer to the first occurrence of *c* (converted to a character) in *string*.

If the character is not found, strchr() returns a null pointer.

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 43
- “memchr() — Search Buffer” on page 492
- “strcat() — Concatenate Strings” on page 809
- “strcmp() — Compare Strings” on page 812

- “strcpy() — Copy String” on page 815
- “strcspn() — Compare Strings” on page 816
- “strncmp() — Compare Strings” on page 828
- “strpbrk() — Find Characters in String” on page 831
- “strrchr() — Find Last Occurrence of Character in String” on page 836
- “strspn() — Search String” on page 837

strcmp() — Compare Strings

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

A built-in function that compares the string pointed to by *string1* to the string pointed to by *string2*. The string arguments to strcmp() must contain a null character (`\0`) marking the end of the string.

The relation between the strings is determined by subtracting: *string[i] — string2[i]*, as *i* increases from `0` to *strlen* of the smaller string. The result is the sign of the difference between the values of the leftmost first pair of characters that differ. The strcmp() function is *not* locale-sensitive.

Returned Value

Returns a value indicating the relationship between the strings, as listed below.

Value
Meaning

< 0
String pointed to by *string1* is less than string pointed to by *string2*.

= 0
String pointed to by *string1* is equal to string pointed to by *string2*.

> 0
String pointed to by *string1* is greater than string pointed to by *string2*.

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\"\n", argv[1], argv[2] );
  }
}
```

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 43
- "memcmp() — Compare Bytes" on page 493
- "strcspn() — Compare Strings" on page 816
- "strncmp() — Compare Strings" on page 828
- "strpbrk() — Find Characters in String" on page 831
- "strrchr() — Find Last Occurrence of Character in String" on page 836
- "strspn() — Search String" on page 837

strcoll() — Compare Strings

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

< 0
String pointed to by *string1* is less than string pointed to by *string2*.

= 0
String pointed to by *string1* is equal to string pointed to by *string2*.

> 0
String pointed to by *string1* is greater than string pointed to by *string2*.

Notes:

- The strcoll() function may need to allocate additional memory to perform the comparison algorithm specified in the LC_COLLATE category. 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 43
- “setlocale() — Set Locale” on page 721
- “strcmp() — Compare Strings” on page 812
- “strncmp() — Compare Strings” on page 828

strcpy() — Copy String

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

A built-in function that copies *string2*, including the ending null character, to the location specified by *string1*. The string arguments 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 43
- “memcpy() — Copy Buffer” on page 495
- “strcat() — Concatenate Strings” on page 809
- “strchr() — Search for Character” on page 811
- “strcmp() — Compare Strings” on page 812

- “strcspn() — Compare Strings” on page 816
- “strncpy() — Copy String” on page 830
- “strpbrk() — Find Characters in String” on page 831
- “strrchr() — Find Last Occurrence of Character in String” on page 836
- “strspn() — Search String” on page 837

strcspn() — Compare Strings

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 43

- “strcat() — Concatenate Strings” on page 809
- “strchr() — Search for Character” on page 811
- “strcmp() — Compare Strings” on page 812
- “strcpy() — Copy String” on page 815
- “strncmp() — Compare Strings” on page 828
- “strpbrk() — Find Characters in String” on page 831
- “strrchr() — Find Last Occurrence of Character in String” on page 836
- “strspn() — Search String” on page 837

strdup() — Duplicate a String

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

Creates a duplicate of the string pointed to by *string*.

Returned Value

If successful, strdup() returns a pointer to a new string that is a duplicate of *string*.

If unsuccessful, strdup() returns a null pointer and sets errno to the following value:

ENOMEM
Insufficient storage space is available.

The caller of strdup() should free the storage obtained for the string.

Related Information

- “string.h” on page 43
- “free() — Free a Block of Storage” on page 278
- “malloc() — Reserve Storage Block” on page 477

strerror() — Get Pointer to Runtime Error Message

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.

Returned Value

Returns a pointer to the string, which may be overwritten by a subsequent call to strerror(). You should 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 43
- "clearerr() — Reset Error and End-of-File" on page 124
- "ferror() — Test for Read/Write Errors" on page 220
- "perror() — Print Error Message" on page 538

strfmon() — Convert Monetary Value to String

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. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

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 strfmon() copies to the output array, and directives, each of which results in fetching zero or more arguments that are converted and formatted. The results are undefined if insufficient arguments are provided for the *format*. If the

format is exhausted while arguments remain, strfmon() simply ignores the excess arguments. 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.

Optional conversion specifications may immediately follow the initial % of a directive in the following order:

=f
A flag used in conjunction with the maximum digits specification *#n* (see below), specifies that the character *f* 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.

^
A flag. You should not format the currency amount with thousands grouping characters. The default is to insert the grouping characters, if defined for the current locale.

Note: The code point for the ^ character will be determined according to the current LC_SYNTAX category.

+ | C | (
A flag that specifies the style of representing positive and negative currency amounts. Only one of +, C, or (may be specified. If + is specified, the locale's equivalent of + and - are used (for example, in USA: the empty (null) string if positive and - if negative). If C is specified, the locale's equivalent of DB for negative and CR for positive are used. If (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.

[-]w
The field width. The decimal digit string 32770dh specifies a minimum field width in which the result of the conversion is right-justified (or left-justified if the optional flag "-" is specified).

#n
The left precision. The decimal digit string *n* 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 strfmon() aligned in the same columns. It can also be used to fill unused positions with a special character as in $***123.45. This option causes an amount to be formatted as if it has the number of digits specified by *n*. If more digit positions are required than the number specified, this conversion specification is ignored. Digit positions in excess of those actually required are filled with the numeric fill character. (See the =f 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*, strfmon() returns the number of bytes placed into the array pointed to by *s, not including the terminating null character. Otherwise, strfmon() returns a value of 0, and the contents of the array are indeterminate.

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 <localdef.h>
#include <monetary.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 33

strftime() — Convert to Formatted Time

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

Format

```
#include <time.h>

size_t strftime(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 29. Conversion Specifiers Used by strftime()

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 is a 2-character, right-justified, blank-filled field.
%E[cCxyY]	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.
%h	Replace with locale's abbreviated month name. This is the same as %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.

Table 29. Conversion Specifiers Used by strftime() (continued)

Specifier	Meaning
%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 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 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 %I:%M:%S %p; or use `t_fmt_ampm` from LC_TIME, 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.

Table 29. Conversion Specifiers Used by strftime() (continued)

Specifier	Meaning
%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.
%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, strftime() copies the characters following the % 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 strftime() 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.

Special Behavior for C

This function is sensitive to time-zone information, which is provided by:

- The TZ environment variable if this program is running with POSIX(ON) 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 if this program is running with POSIX(OFF) or TZ is not defined.

For more information about using POSIX support, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Note: The strftime() function requires time-zone name information to convert the %Z conversion specifier. It is obtained as follows.

If strftime() is called by an 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 strftime() was produced by calling localtime(), strftime() 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 strftime() 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, strftime() uses the characters STD for Standard or DST for Daylight Savings time name.

If the `tm` structure input to strftime() was produced by the gmtime() function, strftime() 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, strftime() returns a value of `0`; 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 49
- “asctime() — Convert Time to Character String” on page 78
- “ctime() — Convert Time to a Character String” on page 153
- “gmtime() — Convert Time to Broken-Down UTC Time” on page 391
- “localtime() — Convert Time and Correct for Local Time” on page 456
- “setlocale() — Set Locale” on page 721
- “time() — Determine Current Time” on page 885

strlen() — Determine String Length

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

A built-in function that determines the length of *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 43
- "mblen() — Calculate Length of Multibyte Character" on page 480
- "strncat() — Concatenate Strings" on page 827
- "strncmp() — Compare Strings" on page 828
- "strncpy() — Copy String" on page 830
- "wcslen() — Calculate Length of Wide-Character String" on page 944

strncasecmp() — Compare Case-Insensitive Strings

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

Compares, while ignoring differences in case, the string pointed to by *string1* to the string pointed to by *string2*. The strncasecmp() function will compare *n* characters, at most.

The string arguments to the function should contain a null character (\0) marking the end of the string.

The strncasecmp() function is locale-sensitive.

Returned Value

Returns a value indicating the relationship between the strings, while ignoring case, as follows:

Value
Meaning

< 0
String pointed to by *string1* is less than string pointed to by *string2*.

= 0
String pointed to by *string1* is equal to string pointed to by *string2*.

> 0
String pointed to by *string1* is greater than string pointed to by *string2*.

No errnos are defined for strncasecmp().

Related Information

- "strings.h" on page 44
- "strcasecmp() — Compare Case-Insensitive Strings" on page 809
- "strcspn() — Compare Strings" on page 816
- "strncmp() — Compare Strings" on page 828

strncat() — Concatenate Strings

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

A built-in function that 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 *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 43
- “strcat() — Concatenate Strings” on page 809
- “strncmp() — Compare Strings” on page 828
- “strncpy() — Copy String” on page 830
- “strpbrk() — Find Characters in String” on page 831
- “strrchr() — Find Last Occurrence of Character in String” on page 836
- “strspn() — Search String” on page 837

strncmp() — Compare Strings

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

A built-in function that compares at most the first *count* characters of *string1* to *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. The strncmp() function is *not* locale sensitive.

Returned Value

Returns a value indicating the relationship between the substrings, as follows:

Value
Meaning

< 0
String pointed to by *substring1* is less than string pointed to by *substring2*

0
String pointed to by *substring1* is equivalent to string pointed to by *substring2*

> 0
String pointed to by *substring1* is greater than string pointed to by *substring2*

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\"\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 493
- “strcmp() — Compare Strings” on page 812
- “strcspn() — Compare Strings” on page 816
- “strncat() — Concatenate Strings” on page 827
- “strncpy() — Copy String” on page 830
- “strpbrk() — Find Characters in String” on page 831
- “strrchr() — Find Last Occurrence of Character in String” on page 836
- “strspn() — Search String” on page 837

strncpy() — Copy String

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

A built-in function that copies at most *count* characters of *string2* to *string1*. If *count* is less than or equal to the length of *string2*, strncpy() does *not* append a null character (`\0`) to the copied string. If *count* is greater than the length of *string2*, strncpy() pads the *string1* result 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", destination );


  /* This is how strncpy works */
  printf( "destination1 is originally = '%s'\n", destination1 );
  return_string = strncpy( destination1, source1, index );
  printf( "After strncpy, destination1 becomes '%s'\n", destination1 );
}
```

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 43
- “memcpy() — Copy Buffer” on page 495
- “strcpy() — Copy String” on page 815
- “strncat() — Concatenate Strings” on page 827
- “strncmp() — Compare Strings” on page 828
- “strpbrk() — Find Characters in String” on page 831
- “strrchr() — Find Last Occurrence of Character in String” on page 836
- “strspn() — Search String” on page 837

strpbrk() — Find Characters in String

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, strpbrk() returns a NULL pointer.

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 "The Blue Danube"
is "anube"
```

Related Information

- "string.h" on page 43
- "strchr() — Search for Character" on page 811
- "strcspn() — Compare Strings" on page 816
- "strncmp() — Compare Strings" on page 828
- "strrchr() — Find Last Occurrence of Character in String" on page 836
- "strspn() — Search String" on page 837

strptime() — Date and Time Conversion

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*.

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. White-space or other non-alphanumeric characters must come between any two conversion specifications.

Table 30. 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).
%	Date, using locale's date format.
%X	Time, using locale's time format.
%y	Year within century (0-99; 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.

Table 31. Modified Directives Used by strptime()

Specifier	Meaning
%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.
%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 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, using the locale's alternative numeric symbols.

The strptime() function executes a directive composed of white-space characters by scanning input up to the first character that is not white space (which remains unscanned) or until it can scan no more characters.

The strptime() function executes a directive that is an ordinary character 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.

The strptime() function executes a series of directives composed of %n, %t, white-space characters or any combination by scanning up to the first character that is not white space (which remains unscanned), or until it can scan no more characters.

The strptime() function executes any other conversion specification by scanning characters until it scans a character matching the next directive or until it can scan no more characters. It then compares these characters, except for the one matching the next directive, to the locale values associated with the conversion specifier. If it finds a match, it sets the values for the appropriate tm structure members to values corresponding to the locale information. It ignores case when matching items in *buf* such as month or weekday names. If it finds no match, strptime() fails and no more characters are scanned.

Returned Value

If successful, strptime() returns a pointer to the character following the last character parsed. Otherwise, strptime() returns a null pointer.

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 49

strrchr() — Find Last Occurrence of Character in String

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

Format

```
#include <string.h>

char *strrchr(const char *string, int c);
```

General Description

A built-in function that 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, strrchr() returns a NULL pointer.

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 43
- “memchr() — Search Buffer” on page 492
- “strchr() — Search for Character” on page 811
- “strcspn() — Compare Strings” on page 816

- “strncmp() — Compare Strings” on page 828
- “strpbrk() — Find Characters in String” on page 831
- “strspn() — Search String” on page 837

strspn() — Search String

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 43
- “strcat() — Concatenate Strings” on page 809

- “strchr() — Search for Character” on page 811
- “strcmp() — Compare Strings” on page 812
- “strcpy() — Copy String” on page 815
- “strcspn() — Compare Strings” on page 816
- “strpbrk() — Find Characters in String” on page 831
- “strrchr() — Find Last Occurrence of Character in String” on page 836

strstr() — Locate Substring

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, strstr() 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 43
- “strchr() — Search for Character” on page 811
- “strcmp() — Compare Strings” on page 812
- “strcspn() — Compare Strings” on page 816
- “strncmp() — Compare Strings” on page 828
- “strpbrk() — Find Characters in String” on page 831
- “strrchr() — Find Last Occurrence of Character in String” on page 836
- “strspn() — Search String” on page 837

strtocoll() — Return Collating Element for String

Standards / Extensions	C or C++	Dependencies
ISO C	both	

Format

```
#include <collate.h>

collel_t strtocoll(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 `collel_t` represents the collating elements. If many-to-one mapping is not defined in the LC_COLLATE of the current locale, then strtocoll() returns `(collel_t)-1`. Also, if the string is not a valid collating element or is of zero length, strtocoll() returns `(collel_t)-1`.

Example

CBC3BS52

```
/* CBC3BS52
   This example uses the strtocoll() function to get the collel_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[]) {
   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 (ismccollel(*rp))
         printf("'%s' ", colltostr(*rp));
      else if (iswprint(*rp))
         printf("'%lc' ", *rp);
      else
         printf("'%x' ", *rp);
   }
}
```

Related Information

- “collate.h” on page 20
- “cclass() — Return Characters in a Character Class” on page 104
- “collequiv() — Return List of Equivalent Collating Elements” on page 132
- “collorder() — Return List of Collating Elements” on page 133
- “collrange() — Calculate the Range List of Collating Elements” on page 134
- “colltostr() — Return a String for a Collating Element” on page 136
- “getmccoll() — Get Next Collating Element from String” on page 345
- “getwmccoll() — Get Next Collating Element from Wide String” on page 386
- “ismccollel() — Identify a Multicharacter Collating Element” on page 429
- “maxcoll() — Return Maximum Collating Element” on page 479

strtod() — Convert Character String to Double

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 fscanf Family of Formatted Input Functions for a description of special infinity and NaN sequences recognized by z/VM formatted input functions, including atof() and strtod() in IEEE floating-point mode.

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 strtod() 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, strtod() returns -HUGE_VAL or +HUGE_VAL. In an underflow, strtod() returns a value of `0`. If no conversion is performed, strtod() returns a value of `0`. In both cases, strtod() sets errno 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 42
- "atof() — Convert Character String to Double" on page 88
- "atoi() — Convert Character String to Integer" on page 89
- "atol() — Convert Character String to Long" on page 89
- "fscanf() - scanf() - sscanf() — Read and Format Data" on page 283
- "strtol() — Convert Character String to Long" on page 843
- "strtoul() — Convert String to Unsigned Integer" on page 845

strtok() — Tokenize String

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, no tokens are contained in the string, and 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, 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. When there are no more tokens, strtok() returns a NULL pointer. 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 43
- “strcat() — Concatenate Strings” on page 809
- “strchr() — Search for Character” on page 811
- “strcmp() — Compare Strings” on page 812
- “strcpy() — Copy String” on page 815
- “strcspn() — Compare Strings” on page 816
- “strspn() — Search String” on page 837

strtol() — Convert Character String to Long

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 *nptr*, a character string, to a `long int` value.

The strtol() 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 `0`. If base is `0`, 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 `0`, 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, strtol() returns LONG_MAX or LONG_MIN, according to the sign of the value and sets errno to ERANGE.

If *base* is not a valid number, or has a value of `1`, strtol() returns a value of `0` and sets errno to `EDOM`.

If the string pointed to by *nptr* does not have the expected form, no conversion is performed, strtol() returns a value of `0` and stores the value of *nptr* 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 42
- “atof() — Convert Character String to Double” on page 88
- “atoi() — Convert Character String to Integer” on page 89
- “atol() — Convert Character String to Long” on page 89
- “fscanf() - scanf() - sscanf() — Read and Format Data” on page 283
- “strtod() — Convert Character String to Double” on page 840
- “strtoul() — Convert String to Unsigned Integer” on page 845

strtoul() — Convert String to Unsigned Integer

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.

The strtoul() 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 0. If base is 0, 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 0, 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 *endptr*, as long as *endptr* is not a null pointer.

Returned Value

Returns the converted unsigned `long int` value, represented in the string, or returns a value of 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 a value of 1, the function returns a value of 0 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 42
- "atof() — Convert Character String to Double" on page 88
- "atoi() — Convert Character String to Integer" on page 89
- "atol() — Convert Character String to Long" on page 89
- "fscanf() - scanf() - sscanf() — Read and Format Data" on page 283
- "strtod() — Convert Character String to Double" on page 840
- "strtol() — Convert Character String to Long" on page 843

strxfrm() — Transform String

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 it 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 strcoll() 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 strxfrm() 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), strxfrm() validates the multibyte characters, whereas previously strxfrm() 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 43
- “localeconv() — Query Numeric Conventions” on page 454
- “setlocale() — Set Locale” on page 721
- “strcmp() — Compare Strings” on page 812
- “strcoll() — Compare Strings” on page 813
- “strncmp() — Compare Strings” on page 828

swab() — Copy and Swap Bytes

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

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 leaves the disposition of the last byte unchanged in the target area. If *nbytes* is `0` or negative, swab() performs no copying.

Returned Value

The swab() function has no returned value.

Related Information

None.

swapcontext() — Save and Restore User Context

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

Saves the current user context in the context structure pointed to by *oucp* and restores the user context structure pointed to by *ucp*. The swapcontext() function 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 `makecontext` has not modified the saved context, and you subsequently call setcontext() or swapcontext() using the saved context, swapcontext() returns a value of `0`.

Note: If makecontext() has not modified the *ucontext* input to swapcontext(), you should 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.

The swapcontext() 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         A pointer to the context that will
                                 resume 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 swapcontext() are used to transfer control in a z/VM C++ program, the behavior in terms of the destruction of automatic objects is undefined. This applies to both z/VM C++ and z/VM 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.

Special Behavior for XPLINK-compiled C++

Restrictions concerning setjmp.h and ucontext.h:

Note:

1. All XPLINK programs compiled with the V2.10 or later C compilers that are to run with Language Environment V2.10 or later libraries and use the **jmp_buf**, **sigjmp_buf** or **ucontext_t** types must not be compiled with C headers from Language Environment V2.9 or earlier.
2. Non-XPLINK functions compiled with any level of Language Environment headers must not define **jmp_buf**, **sigjmp_buf** or **ucontext_t** data items and pass them to XPLINK functions that call getcontext(), longjmp(), _longjmp(), setjmp(), _setjmp(), setcontext(), sigsetjmp(), or swapcontext() with these passed-in data items.
3. When __XPLINK__ is defined, the Language Environment V2.10 and later headers define a larger **jmp_buf**, **sigjmp_buf** or **ucontext_t** area that is required by setjmp(), getcontext(), and related functions when they are called from an XPLINK routine. If __XPLINK__ is not defined, the Language Environment V2.10 and later headers define a shorter **jmp_buf**, **sigjmp_buf** or **ucontext_t** area. The Language Environment headers prior to V2.10 also define the shorter version of these data areas. If an XPLINK function calls setjmp(), getcontext() or similar functions with a short **jmp_buf**, **sigjmp_buf** or **ucontext_t** area, a storage overlay or program check may occur when the C library tries to store past the end of the passed-in (too short) data area.

Returned Value

If successful, swapcontext() does not return from the initial invocation. If the unmodified saved context is later restored, swapcontext() returns a value of `0`.

If unsuccessful, swapcontext() returns a value of -1.

No errnos are defined for swapcontext().

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 51
- “getcontext() — Get User Context” on page 318
- “_longjmp() — Nonlocal Goto” on page 463
- “longjmp() — Restore Stack Environment” on page 465
- “makecontext() — Modify User Context” on page 475
- “setcontext() — Restore User Context” on page 704
- “_setjmp() — Set Jump Point for a Nonlocal Goto” on page 717
- “setjmp() — Preserve Stack Environment” on page 718
- “siglongjmp() — Restore the Stack Environment and Signal Mask” on page 764
- “sigsetjmp() — Save Stack Environment and Signal Mask” on page 778

swprintf() — Write Wide Characters to a Wide-Character Array

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

Format

Non-XPG4

```
#include <wchar.h>

int swprintf(wchar_t *wcs, size_t n, const wchar_t *format, …);
```

XPG4

```
#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 *wcs* argument specifies an array of type `wchar_t` into which swprintf() will write the generated output, rather than an array of type `char`.
- The *format* argument specifies an array of type `wchar_t` that describes how swprintf() will convert subsequent arguments for output, rather than an array of type `char`.
- %c without an `l` prefix means an `int` arg will be converted to `wchar_t`, as if mbtowc() were called, and then written.
- %c with an `l` prefix means a `wint_t` will be converted to `wchar_t` and then written.
- %s without an `l` prefix means a character array containing a multibyte character sequence will be converted to an array of `wchar_ts` and then written. The conversion will take place as if mbrtowc() were called repeatedly.
- %s with an `l` prefix means an array of `wchar_ts` will be written. The array will be written up to but not including the terminating null character, unless the precision specifies a shorter output.

The swprintf() function writes a null wide character at the end of the wide characters written and does not count the null wide character as part of the returned sum. If copying takes place between objects that overlap, the behavior is undefined.

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, you must also define the _MSE_PROTOS feature test macro to make the declaration of swprintf() in the wchar header available when you compile your program.

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 53
- sprintf(), in “fprintf() - printf() - sprintf() — Format and Write Data” on page 264

swscanf() — Read a Wide-Character String

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

Format

Non-XPG4

```
#include <wchar.h>

int swscanf(const wchar_t *wcs, const wchar_t *format, …);
```

XPG4

```
#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 *wcs* argument specifies an array of type `wchar_t` from which swscanf() will obtain the input, rather than an array of type `char`.
- The *format* argument specifies an array of type `wchar_t` that describes the admissible input sequences and how they will be converted for assignment, rather than an array of type `char`.
- %c with no `l` prefix means one or more (depending on precision) `wchar_ts` will 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_ts` will be copied to the array of `wchar_ts` pointed to by the corresponding argument.
- %s with no `l` prefix means a sequence of non-white-space `wchar_ts` 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_ts will be copied, including the terminating null wide character, to the array of `wchar_ts` pointed to by the corresponding argument.

Reaching the end of the wide-character string is equivalent to reaching the end of the character string for sscanf(). If copying takes place between objects that overlap, the behavior is undefined.

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, you must also define the _MSE_PROTOS feature test macro to make the declaration of swscanf() in the wchar header available when you compile your program.

Returned Value

Returns EOF if an input failure occurs before any conversion. Otherwise, swscanf() 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 53
- “fscanf() - scanf() - sscanf() — Read and Format Data” on page 283

symlink() — Create a Symbolic Link to a Path Name

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

Format

```
#define _OPEN_SYS
#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*. The symlink() function does not perform file access checking 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 a value of `0`.

If unsuccessful, symlink() returns a value of `-1`, and it does not affect any file it names. The symlink() function sets errno to one of the following values under the conditions described:

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 error may be returned for either of these reasons:

- *pathname* contains a null character.
- *slink* has a slash as its last component, which indicates that the preceding component will be a directory. A symbolic link cannot be a directory.
- *slink* begins with “//”, which is not allowed.

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().

ENOTDIR
A component of the path prefix of *slink* is not a directory.

ELOOP
A loop exists in symbolic links. The symlink() function issues this error if it detects more than POSIX_SYMLOOP symbolic links during resolution of the *slink* argument.

EEXIST
The file named by *slink* already exists.

ENOSPC
The new symbolic link cannot be created because no space is left on the file system that will contain the symbolic link.

EROFS
The file *slink* cannot reside on a read-only system.

Example

```
#define _OPEN_SYS
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _OPEN_SYS
#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 vmuser1  SYS1     0 Apr 20 13:57 test.file

after symlink()
 4030 --w-------   1 vmuser1  SYS1     0 Apr 20 13:57 test.file
 4031 l---------   1 vmuser1  SYS1     9 Apr 20 13:57 test.symlink -> test.file
after first unlink()
 4031 l---------   1 vmuser1  SYS1     9 Apr 20 13:57 test.symlink -> test.file
```

Related Information

- "unistd.h" on page 51
- "link() — Create a Link to a File" on page 450
- "readlink() — Read the Value of a Symbolic Link" on page 633
- "unlink() — Remove a Directory Entry" on page 910

sysconf() — Determine System Configuration Options

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

Format

```
#define _OPEN_SYS
#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_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_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_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 a value of -1 but does not change the value of errno. If sysconf() fails in some other way, it returns a value of -1.

If unsuccessful, sysconf() sets errno to EINVAL, indicating that the value specified for *name* is incorrect.

Example

CBC3BS61

```
/* CBC3BS61
   This example determines the value of ARG_MAX.
 */
#define _OPEN_SYS
#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 51
- “clock() — Determine Processor Time” on page 125
- “exec Functions” on page 192

system() — Execute a Command

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 system(), you can call commands, EXECs, or executable modules from a system call under CMS.

The *string* argument can take one of two formats:

Command Line

A string with CMS 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 IBM XL C/C++ for z/VM 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 command-line string, the system function generally passes the given string to the command processor for execution. 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 CMS.

If system() receives a string in the form of a command, system() passes the given *string* to the command processor for execution.

If the string is of the form "PGM…", system() calls the program *program_name* to be executed with the parameters following PARM, if any.

In either case, if the specified module is an IBM XL C/C++ for z/VM module, full initialization and termination is performed: automatic closing of files and releasing of fetched modules. To pass information across a system() call, you should use memory files. Memory files are not removed until the root program terminates or *clrmemf* is used.

Special Behavior for POSIX C

The system() function passes the *string* to the sh shell command for execution. The environment is established by the runtime library by issuing a spawn() of the shell.

The system() function ignores the SIGINT and SIGQUIT signals, and blocks the SIGCHILD signal while it waits for the command specified by the *string* to end.

Under OpenExtensions, system() is issued for CMS or OpenExtensions 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 CMS. The system() function has these restrictions:

If __POSIX_SYSTEM is set to no, No, or NO, the program being called cannot reside in the byte file system.

For traditional C application programs, system() is issued only for CMS commands. It cannot pass commands to the shell.

For more information about using POSIX support, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Mixed Environments across a system() Call

You cannot mix Language Environment and system programming environments across a system() call. 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 a nonzero value if a command processor exists and a value of 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. The system() function accepts either command-line or named-program strings. You can use either form to execute a CMS command or a user-written command. Only command MODULES are supported: EXEC and CP commands are not supported directly.

In the case of either a command-line or named-program string, the system will search the usual VM sources for command modules, as documented in the standard CMS search order *for modules* (that is, nucleus extension module, transient area module, nucleus-resident module, file with file type MODULE). If the search sequence is exhausted without finding the name specified in the string, the search is repeated for the translation or synonym of the name (if any exists). This search sequence excludes EXECs and CP commands. Therefore, to execute one of these, the name of an EXEC must be prefixed with "EXEC" and the name of a CP command must be prefixed with "CP" in the argument string.

The system() function restores the *emsg* setting at program invocation, issues the given command in the system() call, and sets *emsg* off again.

Returned Value for POSIX C

This information applies to z/VM programs that have issued a POSIX(ON) system() call. If *string* is a NULL pointer, system() returns a nonzero value. 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 a value of -1 and sets errno to one of the following values under the conditions described:

EAGAIN
There are insufficient resources to create another process, or the process has already reached the maximum number of processes you can run.

ENOMEM
The process requires more space than is available.

For more information about using POSIX support, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Example

```
/* This example illustrates how to use system() to execute a command which
   returns the time.
 */
#include <stdlib.h>

int main(void)
{
    int rc;
    rc = system("CP QUERY 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 42
- "clrmemf() — Clear Memory Files" on page 131
- "signal() — Handle Interrupts" on page 767

takesocket() — Acquire a Socket from Another Program

Standards / Extensions	C or C++	Dependencies
OpenExtensions	both	

Format

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

int takesocket(struct clientid *clientid,int sdesc);
```

General Description

Acquires an AF_INET or AF_INET6 socket from another program. Typically, the other program passes its client ID and socket descriptor through some agreed upon mechanism.

Note: You can give an AF_INET socket only to an AF_INET socket, and an AF_INET6 socket only to an AF_INET6 socket; that is you cannot give an AF_INET6 socket to an AF_INET socket or vise-versa.

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.

Returned Value

If unsuccessful, takesocket() returns a value of -1 and sets errno to one of the following values under the conditions described. Otherwise, the return value is the new socket descriptor.

EACCES
The other application did not give the socket to your application.

EAFNOSUPPORT
The address family is not supported.

EBADF
The *sdesc* parameter does not specify a valid socket descriptor owned by the other application, or the socket has already been taken.

EFAULT
Using the *clientid* parameter, as specified, would result in an attempt to access storage outside the caller's virtual machine.

EINVAL
The *clientid* 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.

EPFNOSUPPORT
The domain field of the *clientid* parameter is not AF_INET or AF_INET6.

Related Information

- "sys/socket.h" on page 46
- "sys/types.h" on page 46
- "getclientid() — Get Identifier for Calling Application" on page 316
- "givesocket() — Make Specified Socket Available" on page 387

tan() — Calculate Tangent

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++	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: These functions work in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

Returned Value

Returns the calculated tangent of *x*. If the correct value would cause underflow, tan() returns a value of 0. If the result overflows, tan() returns HUGE_VAL or -HUGE_VAL. For both underflow and overflow, tan() stores the macro ERANGE 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 32
- “acos(), acosf(), acosl() — Calculate Arccosine” on page 73
- “asin(), asinf(), asinl() — Calculate Arcsine” on page 79
- “atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate Arctangent” on page 83
- “cos(), cosf(), cosl()— Calculate Cosine” on page 143
- “cosh(), coshf(), coshl()— Calculate Hyperbolic Cosine” on page 145
- “sin() — Calculate Sine” on page 785
- “sinh() — Calculate Hyperbolic Sine” on page 786
- “tanh() — Calculate Hyperbolic Tangent” on page 862

tanh() — Calculate Hyperbolic Tangent

Standards / Extensions	C or C++	Dependencies
ISO C POSIX.1 XPG4 XPG4.2 ISO/ANSI C++	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 tanh() cannot have a range error.

Note: These functions work in both IEEE floating-point and hexadecimal floating-point formats. See “IEEE floating-point” on page 63 for more information about IEEE floating-point.

Returned Value

Returns the calculated value of the arctangent of *x*. If the result underflows, atanh() returns a value of `0` and sets 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 32
- “acos(), acosf(), acosl() — Calculate Arccosine” on page 73
- “asin(), asinf(), asinl() — Calculate Arcsine” on page 79
- “atan(), atanf(), atanl(), atan2(), atan2f(), atan2l() — Calculate Arctangent” on page 83
- “cos(), cosf(), cosl()— Calculate Cosine” on page 143
- “cosh(), coshf(), coshl()— Calculate Hyperbolic Cosine” on page 145
- “sin() — Calculate Sine” on page 785
- “sinh() — Calculate Hyperbolic Sine” on page 786
- “tan() — Calculate Tangent” on page 861

tcdrain() — Wait Until Output Has Been Transmitted

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

Format

```
#define _OPEN_SYS
#include <termios.h>

int tcdrain(int fildes);
```

General Description

Waits until all output sent to *fildes* has actually been sent to the terminal device. For subsidiary pseudoterminals, tcdrain() considers data sent when the manager side has read it.

The tcdrain() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

If tcdrain() is called from a background process group against the caller's controlling terminal, a SIGTTOU signal may be generated depending on 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; tcdrain() returns a value of -1 and sets errno to EINTR.
Ignored or blocked	The SIGTTOU signal is not sent, and the function continues normally.

Returned Value

Returns a value of -1 and sets errno to the following value:

EINVAL
Not supported.

Related Information

- "tcflow() — Suspend or Resume Data Flow on a Terminal" on page 864
- "tcflush() — Flush Input or Output on a Terminal" on page 867
- "tcgetattr() — Get the Attributes for a Terminal" on page 869
- "tcgetpgrp() — Get the Foreground Process Group ID" on page 870
- "tcsendbreak() — Send a Break Condition to a Terminal" on page 873
- "tcsetattr() — Set the Attributes for a Terminal" on page 874
- "tcsetpgrp() — Set the Foreground Process Group ID" on page 880

tcflow() — Suspend or Resume Data Flow on a Terminal

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

Format

```
#define _OPEN_SYS
#include <termios.h>

int tcflow(int fildes, int action);
```

General Description

Suspends or resumes transmission or reception of data on a terminal device.

The tcflow() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

int ***fildes***
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 on 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; tcflow() returns a value of -1 and sets errno to EINTR.
Ignored or blocked	The SIGTTOU signal is not sent, and the function continues normally.

Returned Value

If successful, tcflow() returns a value of 0.

If unsuccessful, tcflow() returns a value of -1 and sets errno to one of the following values under the conditions described:

EBADF
fildes is not a valid open file descriptor.

EINTR
A signal interrupted tcflow().

EINVAL
action had an incorrect value.

EIO
TCIOFF or TCION was requested, but the other side of the pseudoterminal connection is closed.

ENOTTY
fildes is not associated with a terminal.

Example

CBC3BT04

```
/* CBC3BT04
   This example resumes transmission.
 */
#define _OPEN_SYS
#include <termios.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>
```

```
main() {
  char Manager[]="manager.tty";
  char Subsidiary[]="subsidiary.tty";
  char text[]="text to be written to tty";
  char data[80];
  int  manager, subsidiary;

  if (mknod(Manager, S_IFCHR¦S_IRUSR¦S_IWUSR, 0x00010000 + 10) != 0)
    perror("mknod() error for manager tty");
  else {
    if (mknod(Subsidiary, S_IFCHR¦S_IRUSR¦S_IWUSR, 0x00020000 + 10) != 0)
      perror("mknod() error for subsidiary tty");
    else {
      if ((manager = open(Manager, O_RDWR¦O_NONBLOCK)) < 0)
        perror("open() error for manager tty");
      else {
        if ((subsidiary = open(Subsidiary, O_RDWR¦O_NONBLOCK)) < 0)
          perror("open() error for subsidiary tty");
        else {
          if (write(subsidiary, text, strlen(text)+1) == -1)
            perror("write() error");
          else if (tcflow(subsidiary, TCOOFF) != 0)
            perror("tcflow() error");
          else {
            puts("output is suspended to tty");
            if (read(manager, data, sizeof(data)) == -1)
              perror("read() error");
            else printf("read '%s' from the tty\n", data);
            if (tcflow(subsidiary, TCOON) != 0)
              perror("tcflow() error");
            else {
              puts("output is resumed to tty");
              if (read(manager, data, sizeof(data)) == --1)
                perror("read() error");
              else printf("read '%s' from the tty\n", data);
            }
          }
          close(subsidiary);
        }
        close(manager);
      }
      unlink(Subsidiary);
    }
    unlink(Manager);
  }
}
```

Output

```
read() error: Resource temporarily unavailable
output is resumed to tty
read 'text to be written to tty' from the tty
```

Note: The example must be run from superuser because mknod() is being used.

Related Information

- “tcdrain() — Wait Until Output Has Been Transmitted” on page 863
- “tcflush() — Flush Input or Output on a Terminal” on page 867
- “tcgetattr() — Get the Attributes for a Terminal” on page 869
- “tcgetpgrp() — Get the Foreground Process Group ID” on page 870
- “tcsendbreak() — Send a Break Condition to a Terminal” on page 873
- “tcsetattr() — Set the Attributes for a Terminal” on page 874
- “tcsetpgrp() — Set the Foreground Process Group ID” on page 880

tcflush() — Flush Input or Output on a Terminal

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

Format

```
#define _OPEN_SYS
#include <termios.h>

int tcflush(int fildes,int where);
```

General Description

Flushes input or output on a terminal.

int *fildes*;
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.

The tcflush() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

If tcflush() is called from a background process group against the caller's controlling terminal, a SIGTTOU signal may be generated depending on 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; tcflush() returns a value of -1 and sets errno to EINTR.
Ignored or blocked	The SIGTTOU signal is not sent, and the function continues normally.

Returned Value

If successful, tcflush() returns a value of 0.

If unsuccessful, tcflush() returns a value of -1 and sets errno to one of the following values under the conditions described:

EBADF
fildes is not a valid open file descriptor.

EINTR
A signal interrupted tcflush().

EINVAL
where has an incorrect value.

ENOTTY
fildes is not associated with a terminal.

Example

CBC3BT05

```
/* CBC3BT05
   This example flushes a string.
 */
#define _OPEN_SYS
#include <termios.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h>

main() {
  char Manager[]="manager.tty";
  char Subsidiary[]="subsidiary.tty";
  char text1[]="string that will be flushed from buffer";
  char text2[]="string that will not be flushed from buffer";
  char data[80];
  int  manager, subsidiary;

  if (mknod(Manager, S_IFCHR¦S_IRUSR¦S_IWUSR, 0x00010000 + 10) != 0)
    perror("mknod() error for manager tty");
  else {
    if (mknod(Subsidiary, S_IFCHR¦S_IRUSR¦S_IWUSR, 0x00020000 + 10) != 0)
      perror("mknod() error for subsidiary tty");
    else {
      if ((manager = open(Manager, O_RDWR¦O_NONBLOCK)) < 0)
        perror("open() error for manager tty");
      else {
        if ((subsidiary = open(Subsidiary, O_RDWR¦O_NONBLOCK)) < 0)
          perror("open() error for subsidiary tty");
        else {
          if (write(subsidiary, text1, strlen(text1)+1) == -1)
            perror("write() error");
          else if (tcflush(subsidiary, TCOFLUSH) != 0)
            perror("tcflush() error");
          else {
            puts("first string is written and tty flushed");
            puts("now writing string that will not be flushed");
            if (write(subsidiary, text2, strlen(text2)+1) == -1)
              perror("write() error");
            else if (read(manager, data, sizeof(data)) == -1)
              perror("read() error");
            else printf("read '%s' from the tty\n", data);
          }
          close(subsidiary);
        }
        close(manager);
      }
      unlink(Subsidiary);
    }
    unlink(Manager);
  }
}
```

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
```

Note: The example must be run from superuser because mknod() is being used.

Related Information

- "tcdrain() — Wait Until Output Has Been Transmitted" on page 863
- "tcflow() — Suspend or Resume Data Flow on a Terminal" on page 864
- "tcgetattr() — Get the Attributes for a Terminal" on page 869
- "tcgetpgrp() — Get the Foreground Process Group ID" on page 870
- "tcsendbreak() — Send a Break Condition to a Terminal" on page 873
- "tcsetattr() — Set the Attributes for a Terminal" on page 874
- "tcsetpgrp() — Set the Foreground Process Group ID" on page 880

tcgetattr() — Get the Attributes for a Terminal

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

Format

```
#define _OPEN_SYS
#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 *fildes*, and stores that information in a memory location specified by *termptr*. The contents of a `termios` structure are described in "tcsetattr() — Set the Attributes for a Terminal" on page 874.

The tcgetattr() function 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.

The tcgetattr() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, tcgetattr() returns a value of `0`.

If unsuccessful, tcgetattr() returns a value of -1 and sets errno to one of the following values under the conditions described:

EBADF
fildes is not a valid open file descriptor.

ENOTTY
The file associated with *fildes* is not a terminal.

Example

CBC3BT06

```
/* CBC3BT06
   This example provides information about the attributes.
 */
#define _OPEN_SYS
#include <termios.h>
#include <stdio.h>
```

```
#include <unistd.h>

main() {
  struct termios term;

  if (tcgetattr(STDIN_FILENO, &term) != 0)
    perror("tcgetatt() error");
  else {
    if (term.c_iflag & BRKINT)
      puts("BRKINT is set");
    else
      puts("BRKINT is not set");
    if (term.c_cflag & PARODD)
      puts("Odd parity is used");
    else
      puts("Even parity is used");
    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

```
BRKINT is set
Odd parity is used
ECHO is set
The end-of-file character is x'37'
```

Related Information

- “cfgetispeed() — Determine Input Baud Rate” on page 108
- “cfgetospeed() — Determine Output Baud Rate” on page 110
- “cfsetispeed() — Set Input Baud Rate in the Termios” on page 112
- “cfsetospeed() — Set Output Baud Rate in the Termios” on page 113
- “tcdrain() — Wait Until Output Has Been Transmitted” on page 863
- “tcflow() — Suspend or Resume Data Flow on a Terminal” on page 864
- “tcflush() — Flush Input or Output on a Terminal” on page 867
- “tcgetpgrp() — Get the Foreground Process Group ID” on page 870
- “tcsendbreak() — Send a Break Condition to a Terminal” on page 873
- “tcsetattr() — Set the Attributes for a Terminal” on page 874
- “tcsetpgrp() — Set the Foreground Process Group ID” on page 880

tcgetpgrp() — Get the Foreground Process Group ID

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

Format

```
#define _OPEN_SYS
#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 specified by *fildes*. The tcgetpgrp() function can run from a background process, but the information may subsequently be changed by a process in the foreground process group.

The tcgetpgrp() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, tcgetpgrp() returns the value of the foreground process group's PGID.

If unsuccessful, tcgetpgrp() returns a value of -1 and sets errno to one of the following values under the conditions described:

EBADF
fildes is not a valid open file descriptor.

ENOTTY
fildes is not associated with a terminal.

Example

CBC3BT07

```
/* CBC3BT07
   This example gets the foreground PGID.
 */
#define _OPEN_SYS
#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 51
- "setpgid() — Set Process Group ID for Job Control" on page 729
- "setsid() — Create Session, Set Process Group ID" on page 733
- "tcdrain() — Wait Until Output Has Been Transmitted" on page 863
- "tcflow() — Suspend or Resume Data Flow on a Terminal" on page 864
- "tcflush() — Flush Input or Output on a Terminal" on page 867
- "tcgetattr() — Get the Attributes for a Terminal" on page 869
- "tcsendbreak() — Send a Break Condition to a Terminal" on page 873
- "tcsetattr() — Set the Attributes for a Terminal" on page 874
- "tcsetpgrp() — Set the Foreground Process Group ID" on page 880

tcperror() — Print Error Messages of a Socket Function

Standards / Extensions	C or C++	Dependencies
OpenExtensions	both	

Format

```
#define _OPEN_SYS_SOCK_EXT
#include <sys/socket.h>
#include <stdio.h>
#include <errno.h>
void tcperror(const char *socket);
```

General Description

When a socket call produces an error, the call returns a negative value and the errno variable 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 *socket* is non-NULL, tcperror() prints the *socket* string followed by a colon, followed by a space, followed by the error message, and terminated with a new-line character. If *socket* 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 OpenExtensions.

socket
A NULL or NULL-terminated character string.

Returned Value

The tcperror() function has no returned 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 sets errno to ENOMEM. When tcperror() is called, it prints the string:

```
   socket():  not enough storage (ENOMEM)
```

Example 2:

```
if ((s=socket(AF_INET6, SOCK_DGRAM, 0)) < 0)
   tcperror(NULL);
```

If the socket() call produces the error ENOMEM, socket() returns a negative value and sets errno to ENOMEM. When tcperror() is called, it prints the string:

```
    Not enough storage (ENOMEM)
```

Related Information

- “sys/socket.h” on page 46
- “perror() — Print Error Message” on page 538

tcsendbreak() — Send a Break Condition to a Terminal

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

Format

```
#define _OPEN_SYS
#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. The tcsendbreak() function sends a continuous stream of zero bits for the specified *duration*. Calling tcsendbreak() is the usual method of sending a BREAK on a line.

Under OpenExtensions, tcsendbreak() does not send a BREAK signal; it has no effect because only pseudoterminals are supported.

The tcsendbreak() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

If tcsendbreak() is called from a background process group against the caller's controlling terminal, a SIGTTOU signal may be generated depending on 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; tcsendbreak() returns a value of -1 and sets errno to EINTR.
Ignored or blocked	The SIGTTOU signal is not sent, and the function continues normally.

Returned Value

If successful, tcsendbreak() returns a value of 0.

If unsuccessful, tcsendbreak() returns a value of -1 and sets errno to one of the following values under the conditions described:

EBADF
fildes is not a valid open file descriptor.

EINTR
A signal interrupted tcsendbreak().

ENOTTY
fildes is not associated with a terminal.

Example

CBC3BT08

```
/* CBC3BT08
   This example breaks terminal transmission.
 */
#define _OPEN_SYS
```

```
#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 863
- "tcflow() — Suspend or Resume Data Flow on a Terminal" on page 864
- "tcflush() — Flush Input or Output on a Terminal" on page 867
- "tcgetattr() — Get the Attributes for a Terminal" on page 869
- "tcgetpgrp() — Get the Foreground Process Group ID" on page 870
- "tcsetattr() — Set the Attributes for a Terminal" on page 874
- "tcsetpgrp() — Set the Foreground Process Group ID" on page 880

tcsetattr() — Set the Attributes for a Terminal

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

Format

```
#define _OPEN_SYS
#include <termios.h>

int tcsetattr(int fd, int when, const struct termios *termptr);
```

General Description

Changes the attributes associated with a terminal. A `termios` control structure specifies the new attributes. Programs should always issue a tcgetattr() first, modify the desired fields, and then issue a tcsetattr(). Programs should only issue a tcsetattr() using a `termios` structure that was obtained using tcgetattr(); and tcsetattr() should only use a `termios` structure that was obtained using tcgetattr().

`int` *fd*
: Indicates an open file descriptor associated with a terminal.

`int` *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 the manager pseudoterminal has read all output written to *fd*. You should 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.

struct termios *termptr;
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
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.

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.

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.

IGNCR
Ignores input carriage returns. If this bit is set to 1, the setting of ICRNL has no effect.

If IGNCR is `0` and ICRNL is 1, input carriage returns are converted to new-line (line-feed) characters.

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, the terminal special file does not check whether this bit is set correctly.

ISTRIP
Strips valid input bytes to 7 bits. If this bit is set to `0`, the complete byte is processed.

IXOFF
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 `c_cc` member of the `termios` structure. dictates the character used as the STOP character. 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 the input queues contain space for more input. Again, the `c_cc` member dictates the character used as the START character. It is intended to tell the terminal that it can resume transmission of input.

IXON
Enables start/stop output control. If the system receives a STOP character as input, it will suspend output on the associated terminal until it receives a START character. 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.

If IXON is `0`, any STOP or START characters read are passed on as input to an application reading from the terminal.

PARMRK
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.

Under OpenExtensions, if this bit is set to `1` in a request, it is ignored. It is always set to `0`.

If both PARMRK and IGNPAR are `0`, a character with a framing or parity error is sent to the application as null.

`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.

`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 openf() returns immediately without waiting for a modem connection to complete. If this bit is set to `0`, modem status lines are monitored and openf() waits for the modem connection.

CREAD
Enables reception. If this bit is set to `0`, no input characters are received from the terminal.

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
```

Under OpenExtensions, all values are accepted but CSIZE is changed to CS8.

CSTOPB
Sends two stop bits when necessary. If CSTOPB is `0`, only one stop bit is used.

HUPCL
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.

For pseudo-TTYs, HUPCL controls what happens when the manager TTY is closed.

PARENB
Enables parity generation and detection. A parity bit is added to each character on output, and expected from each character on input.

PARODD
Indicates odd parity (when parity is enabled). If PARODD is `0`, even parity is used (when parity is enabled).

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 `c_cc` member of the `termios` 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 `\n` character. ECHOK has an effect only if the ICANON bit is set to `1`.

ECHONL
Echoes the new-line (line-feed) character `\n`, even if the ECHO bit is off. ECHONL has an effect only if the ICANON bit is set to `1`.

ICANON
Enables canonical input processing, also called *line mode*. 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 `c_cc` member of the `termios` 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 MAX_CANON value specifies the maximum number of bytes in an input line in canonical mode.

Under OpenExtensions, ICANON is always set to `1`.

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 *noncanonical mode*. 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 `c_ccumember`.

IEXTEN
Enables extended implementation-defined functions. These are not defined, and IEXTEN is always set to `0`.

ISIG
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 `c_cc` member.

If ISIG is `0`, the system does not generate signals when these special control characters are entered.

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.

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: 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 \n).

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 c_lflag. 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
Gives the MIN value for noncanonical mode processing.

OpenExtensions does not support noncanonical mode processing.

This is the minimum number of bytes that a call to read() should return in noncanonical mode; it is not used in line mode.

If both MIN and TIME are greater than 0, read() returns when MIN characters have been read 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 a character is already in the queue when read() is called, the timer starts running immediately.

If MIN is greater than 0 and TIME is 0, read() waits for MIN characters to be entered, no matter how long that takes.

If MIN is 0 and TIME is greater than 0, read() returns when the timer runs out or when a single character is received (whichever comes first). The read() function returns either one character (if one is received) or 0 (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 `0`, and the timer starts only when a character is received.)

If both MIN and TIME are `0`, 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().

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.

OpenExtensions does not support noncanonical mode processing.

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 is processed in this way:

Processing for SIGTTOU	Expected Behavior
Default or signal handler	The SIGTTOU signal is generated, and the function is not performed; tcsetattr() returns a value of -1 and sets errno to EINTR.
Ignored or blocked	The SIGTTOU signal is not sent, and the function continues normally.

The tcsetattr() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, tcsetattr() returns a value of `0`.

If unsuccessful, tcsetattr() returns a value of -1 and sets errno to one of the following values under the conditions described:

EBADF
fildes 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.

ENOTTY
fildes is not associated with a terminal.

Example

CBC3BT09

```
/* CBC3BT09
   The following attributes changes the terminal attributes.
 */
#define _OPEN_SYS
#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 Input Baud Rate” on page 108
- “cfgetospeed() — Determine Output Baud Rate” on page 110
- “cfsetispeed() — Set Input Baud Rate in the Termios” on page 112
- “cfsetospeed() — Set Output Baud Rate in the Termios” on page 113
- “open() — Open a File” on page 527
- “read() — Read From a File ” on page 628
- “tcdrain() — Wait Until Output Has Been Transmitted” on page 863
- “tcflow() — Suspend or Resume Data Flow on a Terminal” on page 864
- “tcflush() — Flush Input or Output on a Terminal” on page 867
- “tcgetattr() — Get the Attributes for a Terminal” on page 869
- “tcgetpgrp() — Get the Foreground Process Group ID” on page 870
- “tcsendbreak() — Send a Break Condition to a Terminal” on page 873
- “tcsetpgrp() — Set the Foreground Process Group ID” on page 880

tcsetpgrp() — Set the Foreground Process Group ID

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

Format

```
#define _OPEN_SYS
#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 specified 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 tcsetpgrp() affects future access from any file descriptor in use for the terminal. You should consider using redirection when specifying the file descriptor.

The tcsetpgrp() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, tcsetpgrp() returns a value of 0.

If unsuccessful, tcsetpgrp() returns a value of -1 and sets errno to one of the following values under the conditions described:

EBADF
fildes is not a valid open file descriptor.

EINVAL
The *newid* value is not supported by this implementation.

ENOTTY
The process calling tcsetpgrp() does not have a controlling terminal, or *fildes* 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 _OPEN_SYS
#include <termios.h>
#include <unistd.h>
#include <wait.h>
#include <stdio.h>

main() {
  pid_t pid;
  int status;

  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);
    }

}
```

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 51
- "tcdrain() — Wait Until Output Has Been Transmitted" on page 863
- "tcflow() — Suspend or Resume Data Flow on a Terminal" on page 864
- "tcflush() — Flush Input or Output on a Terminal" on page 867
- "tcgetattr() — Get the Attributes for a Terminal" on page 869
- "tcgetpgrp() — Get the Foreground Process Group ID" on page 870
- "tcsendbreak() — Send a Break Condition to a Terminal" on page 873
- "tcsetattr() — Set the Attributes for a Terminal" on page 874

tdelete() — Delete Binary Tree Node

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

Deletes a node from a binary search tree. The arguments for tdelete() are the same as for tsearch(). If the deleted node was the root of the tree, tdelete() changes the variable pointed to by *rootp* and 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.

The tdelete() function makes comparisons with a user-supplied routine, whose address you pass as the *compar* argument. It calls this routine with two arguments, which point to the elements being compared. The 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.

For information on threading behavior, see “tsearch() — Search Binary Tree” on page 896.

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

Returns a pointer to the parent of the deleted node.

If the node is not found or if *rootp* is a null pointer on entry, tdelete() returns a null pointer.

No errnos are defined for tdelete().

Related Information

- “search.h” on page 38
- “bsearch() — Search Arrays” on page 100
- “hsearch() — Search Hash Tables” on page 394
- “lsearch() — Linear Search and Update” on page 468
- “tfind() — Find Binary Tree Node” on page 884
- “tsearch() — Search Binary Tree” on page 896
- “twalk() — Walk Binary Tree” on page 899

tempnam() — Generate a Temporary File Name

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	POSIX(ON)

Format

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

char *tempnam(const char *dir, const char *pfx);
```

General Description

Generates a path name that may be used for a temporary file. If you set the environment variable TMPDIR, the directory it specifies will become the directory part of the generated path name, if it is accessible. If *dir* is non-null and accessible, tempnam() will use it in the generated path name. Otherwise, it will use the value of {P_tmpdir}, defined in the <stdio.h> header, as the directory component of the name. If that is inaccessible, then tempnam() will use /tmp.

You can use *pfx* to specify the beginning of the file name part of the path name. It may be a null pointer or point to a string of up to five bytes.

The generated names 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.

The tempnam() function is supported only in a POSIX program.

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 sets errno to one of the following values under the conditions described:

ENAMETOOLONG
The generated name exceeded the maximum allowable path name length.

ENOMEM
Insufficient storage is available.

Related Information

- "stdio.h" on page 40
- "fopen() — Open a File" on page 254
- "free() — Free a Block of Storage" on page 278
- "open() — Open a File" on page 527
- "tmpfile() — Create Temporary File" on page 888
- "tmpnam() — Produce Temporary File Name" on page 889
- "unlink() — Remove a Directory Entry" on page 910

tfind() — Find Binary Tree Node

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

Like tsearch(), tfind() searches for a node in the tree, returning a pointer to it if it is found. The arguments for tfind() are the same as for tsearch().

The tfind() function makes comparisons with a user-supplied routine, whose address you pass as the *compar* argument. It calls this routine with two arguments, which point to the elements being compared. The 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.

For information on threading behavior, see "tsearch() — Search Binary Tree" on page 896.

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, tfind() returns a pointer to it.

If the node is not found or if *rootp* is a null pointer on entry, tfind() returns a null pointer.

No errors are defined for tfind().

Related Information

- "search.h" on page 38
- "bsearch() — Search Arrays" on page 100
- "hsearch() — Search Hash Tables" on page 394
- "lsearch() — Linear Search and Update" on page 468
- "tdelete() — Delete Binary Tree Node" on page 882
- "tsearch() — Search Binary Tree" on page 896
- "twalk() — Walk Binary Tree" on page 899

time() — Determine Current Time

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.

Returned Value

Returns the current calendar time and stores the returned value in the location specified by *timeptr*. If *timeptr* is NULL, time() does not store the returned value. If the calendar time is not available, time() returns `(time_t)(-1)`.

The time() function is accurate to the nearest second. If you need a greater degree of accuracy, you can call the S/370* STCK instruction from an assembler routine. For more information about this instruction, see *z/Architecture Principles of Operation (https://publibfp.dhe.ibm.com/epubs/pdf/a227832d.pdf)*.

A returned value of `0` indicates the epoch, which was at the Coordinated Universal Time of 00:00:00 on January 1, 1970.

Returned Value for POSIX C

Under POSIX only, time() returns the value of time in seconds since the epoch. For more information about using POSIX support, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

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 49
- "asctime() — Convert Time to Character String" on page 78
- "clock() — Determine Processor Time" on page 125
- "ctime() — Convert Time to a Character String" on page 153
- "gmtime() — Convert Time to Broken-Down UTC Time" on page 391
- "localtime() — Convert Time and Correct for Local Time" on page 456
- "mktime() — Convert Local Time" on page 505

times() — Get Process and Child Process Times

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

Format

```
#define _OPEN_SYS
#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.

The times() function 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 OpenExtensions, 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 OpenExtensions process.

clock_t tms_stime
Amount of processor time used by the system.

Under OpenExtensions, this value represents kernel busy time running on behalf of the calling process. It does not include processor time performing other VM 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 that 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.

The times() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

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 that an overflow 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 _OPEN_SYS
#include <sys/times.h>
#include <time.h>
#include <sys/types.h>
#include <spawn.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 (spawn("childprog", 0, 0, 0, 0, 0) == -1)
    perror("spawn() error");
  else 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 46
- “time.h” on page 49
- “exec Functions” on page 192
- “time() — Determine Current Time” on page 885
- “wait() — Wait for a Child Process to End” on page 927
- “waitpid() — Wait for a Specific Child Process to End” on page 930

tmpfile() — Create Temporary File

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

If successful, tmpfile() returns a pointer to the stream associated with the file created. If tmpfile() cannot open the file, it returns NULL. On normal termination (exit()), these temporary files are removed; on abnormal termination, an effort is made to remove these files.

When the calling application is an OpenExtensions C 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:

EINTR
A signal was caught during tmpfile().

EMFILE
{OPEN_MAX} file descriptors are currently open in the calling process.

{FOPEN_MAX} streams are currently open in the calling process.

ENFILE
The maximum allowable number of files is currently open in the system.

ENOMEM
Insufficient storage space is available.

ENOSPC
The directory or file system which would contain the new file cannot be expanded.

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 40
- “fopen() — Open a File” on page 254

tmpnam() — Produce Temporary File Name

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 and 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.

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.

If *string* is not a NULL pointer, tmpnam() 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 C

When the calling application is an OpenExtensions C program, tmpnam() returns a unique file name in the byte file system. 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 40
- “fopen() — Open a File” on page 254

toascii() — Translate Integer to a 7-Bit ASCII Character

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

Special Behavior for _XOPEN_SOURCE

Converts its argument to a 7-bit US-ASCII character code.

Note: You should not use toascii() to convert EBCDIC characters to ASCII. Attempts to use it in this manner will not function as expected.

Special Behavior for _ALL_SOURCE

The toascii() function assumes *c* module 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 (for example, 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() module 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

Returns the value (c & 0x7f).

Special Behavior for _ALL_SOURCE

If the current locale is not a single byte locale (that is, mb_cur_max > 1), toascii() sets errno to ENOSYS and returns a value of -1. Otherwise, toascii() assumes *c* module 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.

The 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.                                    */
/*                                                                   */
/*                           IBM 1047               ISO 8859-1       */
/*    Character              Encoding               Encoding         */
/* (Symbolic Name)           (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      OB              0B      0B          */
/*  <FF>       <FF>        0C      0C              0C      0C          */
/*  <CR>       <CR>        0D      0D              0D      OD          */
/*  <SO>       <SO>        0E      0E              0E      OE          */
/*  <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>   <hypehn>    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          */
```

```
/*  <EO>       <IUS/IS1>   FF      1F             9F      1F         */
/*                                                                   */
/*********************************************************************/
```

Related Information

- "isascii() — Test for 7-bit US-ASCII Character" on page 421

tolower(), toupper() — Convert Character Case

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

The tolower() function converts *c* to a lowercase letter, if possible. Conversely, toupper() converts *c* to an uppercase letter, if possible.

The DBCS is not supported. Using characters from the DBCS results in unspecified behavior.

Returned Value

If successful, tolower() and toupper() return the corresponding character, as defined in the LC_CTYPE category of the current locale, if such a character exists.

If unsuccessful, tolower() and toupper() 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 22
- "isalnum() to isxdigit() — Test Integer Value" on page 419

• “towlower(), towupper() — Convert Wide-Character Case” on page 896

towlower(), towupper() — Convert Wide-Character Case

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

Format

```
#include <wctype.h>

wint_t towlower(wint_t wc);
wint_t towupper(wint_t wc);
```

General Description

The towlower() function converts *wc* to the corresponding lowercase letter. Conversely, towupper() converts *wc* to the corresponding uppercase letter.

Returned Value

If *wc* is a wide character for which iswupper() (or iswlower()) is true and a corresponding wide character exists for which iswlower() (or iswupper()) is true, towlower() (or towupper()) returns the corresponding wide character; otherwise, *wc* is returned unchanged.

Related Information

• “wctype.h” on page 54
• “iswalnum() to iswxdigit() — Test Wide Integer Value” on page 432
• “tolower(), toupper() — Convert Character Case” on page 895

tsearch() — Search Binary Tree

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

Builds and accesses a binary search tree. The *key* argument points to an element to be accessed or stored. If a node's element in the tree has the same value as key, tsearch() returns a pointer to this found node. Otherwise, tsearch() creates a new node and copies the value of *key* to this new node. The tsearch() function only copies pointers, so the calling routine must store the data. The variable specified by *rootp* points to the root node of the tree. If this variable has a null value, the tree is empty and tsearch() will set the variable to point to the node which will be at the root of the new tree.

The tsearch() function makes comparisons with a user-supplied routine, whose address you pass as the *compar* argument. It calls this routine with two arguments, which point to the elements being compared. The 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 can be shared among threads. The user is responsible for serializing access to a shared tree. No variables are related to these functions, which are internal to the library and/or give rise to multithreading 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, tsearch() returns a pointer to it; otherwise, it returns a pointer to the inserted item. If not enough space is available to create a new node, or if *rootp* is a null pointer on entry, tsearch() returns a null pointer.

No errnos are defined for tsearch().

Related Information

- “search.h” on page 38
- “bsearch() — Search Arrays” on page 100
- “hsearch() — Search Hash Tables” on page 394
- “lsearch() — Linear Search and Update” on page 468
- “tdelete() — Delete Binary Tree Node” on page 882
- “tfind() — Find Binary Tree Node” on page 884
- “twalk() — Walk Binary Tree” on page 899

ttyname() — Get the Name of a Terminal

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

Format

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

char *ttyname(int fildes);
```

General Description

Returns a string containing the path name of the terminal associated with the given file descriptor, *fildes*. Subsequent calls to ttyname() may overwrite this string because the returned pointer may point to static data.

The ttyname() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

If successful, ttyname() returns a string containing a path name.

If unsuccessful because *fildes* is not a terminal, or the path name cannot be determined, ttyname() returns NULL.

No errnos are defined for ttyname()

Example

CBC3BT16

```
/* CBC3BT16
   This example provides the path name of the terminal associated with stdin.
 */
#define _OPEN_SYS
#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/ttyp0000
```

Related Information

- “unistd.h” on page 51
- “ctermid() — Generate Path Name for Controlling Terminal” on page 151
- “isatty() — Test if Descriptor Represents a Terminal” on page 426

ttyslot() — Find the Slot in the utmpx File of the Current User

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

Format

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

int ttyslot(void);
```

General Description

Returns the index of the current user's entry in the `utmpx` database. This 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 obtains the terminal device. The /dev/ part returned by ttyname() is not used when searching the `utmpx` database member `ut_line`.

Returned Value

If successful, ttyslot() returns the index of the current user's entry in the `utmpx` database.

If ttyslot() encounters an error while searching the database or if none of the file descriptors `0`, `1`, or `2` is associated with a terminal device, ttyslot() returns a value of `-1`.

No errors are defined for ttyslot().

Related Information

- "endutxent() — Close the utmpx Database" on page 186
- "ttyname() — Get the Name of a Terminal" on page 897

twalk() — Walk Binary Tree

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

Traverses a binary search tree. The *root* argument points to the root node of the tree to be traversed. (You may use any node in a tree as the root for a walk below that node.) The *action* argument is the name of a routine to be invoked at each node. You call this routine with three arguments. The first argument is the address of the node being visited. It points to an unspecified structure, which the application must not modify. A pointer-to-node can always be converted to a 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.

For information on threading behavior, see "tsearch() — Search Binary Tree" on page 896.

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 has no returned value.

No errnos are defined for twalk().

Related Information

- "search.h" on page 38

- “bsearch() — Search Arrays” on page 100
- “hsearch() — Search Hash Tables” on page 394
- “lsearch() — Linear Search and Update” on page 468
- “tdelete() — Delete Binary Tree Node” on page 882
- “tfind() — Find Binary Tree Node” on page 884
- “tsearch() — Search Binary Tree” on page 896

tzset() — Set the Time Zone

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

Format

```
#define _OPEN_SYS
#include <time.h>

void tzset(void);
```

General Description

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 is incorrect,tzset() obtains time-conversion information 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.

The tzset() function 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
: Indicates 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. *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≤*n*≤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≤*n*≤365). Leap days are counted, and it is possible to refer to February 29.

Mm.n.d

The *d*th day (0≤*d*≤6) of week *n* of month *m* of the year (1≤*n*≤5, and 1≤*m*≤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 the 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.

The tzset() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Special Behavior for XPG4

The tzset()function sets the external variable *timezone* to the difference, in seconds, between Coordinated Universal Time (UTC) and local standard time. The tzset() function 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, you cannot use them reliably in a multithreaded application or an application running from a DLL. The runtime library provides two special functions, __tzone() and __dlght() that return the address of thread-specific versions of these external variables.

Special Behavior for OpenExtensions Services

The tzset() function only parses the TZ environment variable if a threaded application calls it from the initial processing thread (IPT).

Note: The tzset() function is sensitive to time zone information which is provided by:

- The TZ environmental variable when POSIX(ON) and TZ are correctly defined, or by the _TZ environmental variable when POSIX(OFF) and _TZ are 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

The tzset() function has no returned value.

No errnos are defined for tzset().

Example

CBC3BT17

```
/* CBC3BT17
   This example set time conversion information for Eastern Standard and
   Eastern Daylight Savings Time in the United States.
 */
#define _OPEN_SYS
#include <env.h>
#include <time.h>

int main(void)
{

   setenv("TZ", "EST5EDT", 1);
   tzset();
}
```

Related Information

- "time.h" on page 49
- "ctime() — Convert Time to a Character String" on page 153
- "localtime() — Convert Time and Correct for Local Time" on page 456
- "mktime() — Convert Local Time" on page 505
- "strftime() — Convert to Formatted Time" on page 821

umask() — Set or Display the File Mode Creation Mask

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

Format

```
#define _OPEN_SYS
#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 controls file-permission bits that should be set whenever the process creates a file. 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.

The umask() function uses only the file-permission bits of the new mask. The meaning of other bits is implementation defined. For more information on these symbols, see "chmod() — Change the Mode of a File or Directory" on page 117.

Returned Value

The umask() function is always successful and returns the previous value of the file creation mask.

No errnos are defined for umask().

Example

CBC3BU01

```
/* CBC3BU01
*/
#define _OPEN_SYS
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _OPEN_SYS
#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 46
- "chmod() — Change the Mode of a File or Directory" on page 117
- "creat() — Create a New File or Rewrite an Existing One" on page 146
- "mkdir() — Make a Directory" on page 498
- "mkfifo() — Make a FIFO Special File" on page 499
- "open() — Open a File" on page 527

umount() — Remove a Virtual File System

Standards / Extensions	C or C++	Dependencies
OpenExtensions	both	

Format

```
#define _OPEN_SYS
#include <sys/stat.h>

int umount(const char *filesystem, mtm_t mtm);
```

General Description

Removes a file system from the file hierarchy. 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.

The umount() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

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, umount() performs the unmount. Otherwise, umount() rejects the request.

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 umount() function unmounts the file system 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 a problem occurs saving the data, the unmount request fails.

MTM_FORCE
A forced unmount request. The umount() function unmounts the file system immediately, forcing any users of any files in the specified file system to fail. All data changes made up to the time of the request are saved. If a problem occurs saving the data, the request continues; and the data may be lost. To prevent lost data, you should issue an immediate umount() request before issuing a forced umount() request.

MTM_RESET
A reset unmount request. This stops a previous unmount drain request.

For the Byte File System, only MTM_UMOUNT is appropriate. All other values for the argument are treated as MTM_UMOUNT.

Returned Value

If successful, umount() returns a value of `0`.

If unsuccessful, umount() returns a value of `-1`, and sets errno to one of the following values under the conditions described:

EBUSY
The file system is unavailable.

EINVAL
One of the following occurred:

- A parameter was incorrectly specified. You should verify the spelling of the file system name and the setting of *mtm*.
- The file system name started with the reserved string "//".
- The file system specified is not mounted.

ENOTDIR
The file system name identifies an existing file, but the file is not a directory.

ENOENT
One of the following occurred:

- The file system name specified does not exist.
- Nothing was mounted prior to issuing umount().

Example

CBC3BU02

```
/* CBC3BU02
   This example removes a file system from a user's logical
   filesystem hierarchy
 */
#define _OPEN_SYS
#include <sys/stat.h>
#include <stdio.h>
#include <unistd.h>

main() {
  char BFS--="/new_fs";
  char filesystype-9-="VMBFS   ";

  puts("Before umount()");
  system("EXEC OPENVM QUERY MOUNT");
  if (umount(BFS, MTM_UMOUNT) != 0)
    perror("umount() error");
  else {
    puts("After umount()");
    system("EXEC OPENVM QUERY MOUNT");
  }
}
```

Output

```
Before umount()
Mount point = '/new_fs'
Type  Stat  Mounted
BFS   R/W  '/../VMBFS:VMSYSU:USERXYZ/'
Mount point = '/'
Type  Stat  Mounted
BFS   R/W  '/../VMBFS:VMSYS:ROOT/'

After umount()
Mount point = '/'
Type  Stat  Mounted
BFS   R/W  '/../VMBFS:VMSYS:ROOT/'
```

Related Information

- “sys/stat.h” on page 46
- “mount() — Make a File System Available” on page 508

uname() — Display Current Operating System Name

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

Format

```
#define _OPEN_SYS
#include <sys/utsname.h>

int uname(struct utsname *name);
```

General Description

Gets information identifying the operating system you are running on. The *name* argument points to a memory area where uname() can store a structure describing the operating system the process is running on.

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 name of the node within an implementation-specified communication network.

`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.

The uname() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, uname() returns a nonnegative value.

If unsuccessful, uname() returns a value of -1 and may set errno to indicate the reason for the failure; however, 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 _OPEN_SYS
#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:  VM/ESA
Nodename: SY1
Release:  CMS_14_0000_36
Version:  CP_2.3.0_0000_32
Machine:  9021
```

Related Information

- "sys/utsname.h" on page 48

ungetc() — Push Character onto Input Stream

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*. A subsequent read on the same stream returns the pushed back characters 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, which 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 ungetc() clears the EOF indicator for the stream.

Characters pushed back by ungetc(), and subsequently not read in, will be erased if you call fseek(), fsetpo(), rewind(), or fflush() 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 an application to ignore ungetc() characters for fflush(), fgetpos(), and fseek() using SEEK_CUR. For more details, see *z/OS: XL C/C++ Programming Guide*.

The ungetc() function is not supported for files opened with `type=record`.

The ungetc() function 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition must also occur, unless an EOF has been reached.

Returned Value

Returns the integer *c* argument 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 40
- “fflush() — Write Buffer to File” on page 233
- “fseek() — Change File Position” on page 292
- “getc() - getchar() — Read a Character” on page 312
- “putc() - putchar() — Write a Character” on page 613

ungetwc() — Push a Wide Character onto a Stream

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, `wc` back onto the input stream, `stream`. Subsequent read on that stream will return the pushed-back wide characters 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() used in SEEK_CUR, and fgetpos(). Using the environment variable, _EDC_COMPAT causes fflush(), fseek() with SEEK_CUR, and fgetpos()to ignore a pushed-back wide character. For details, see *z/OS: XL 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(). You can use the environment variable _EDC_COMPAT to cause fflush(), fgetpos(), and fseek() to ignore the pushed-back wide character.

The ungetwc() function is not supported for files opened with `type=record`.

The ungetwc() function 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition must also occur, unless an EOF has been reached.

Returned Value

If successful, ungetwc() returns the wide character pushed back after conversion.

If unsuccessful, ungetwc() returns WEOF.

Notes:

- Only one wide character can be pushed back.
- The position on the stream after a successful ungetwc() is one wide character prior to the current position. For details on backing up a wide character, see *z/OS: XL C/C++ Programming Guide*.

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 53
- “fflush() — Write Buffer to File” on page 233
- “fgetwc() — Get Next Wide Character” on page 239
- “fputwc() — Output a Wide Character” on page 274
- “fseek() — Change File Position” on page 292
- “fsetpos() — Set File Position” on page 294

unlink() — Remove a Directory Entry

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

Format

```
#define _OPEN_SYS
#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 path name, 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.

You cannot use unlink() to remove a directory; you should 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.

Returned Value

If successful, unlink() returns a value of `0`.

If unsuccessful, unlink() returns a value of `-1` and sets errno to one of the following values under the conditions described:

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. The unlink() function issues this error if it detects more than POSIX_SYMLOOP symbolic links 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 _OPEN_SYS
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#undef _OPEN_SYS
#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 51
- “close() — Close a File” on page 127
- “link() — Create a Link to a File” on page 450
- “open() — Open a File” on page 527
- “remove() — Delete File” on page 659
- “rmdir() — Remove a Directory” on page 674

usleep() — Suspend Execution for an Interval

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	POSIX(ON)

Format

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

int usleep( useconds_t useconds);
```

General Description

Suspends thread execution for the number of microseconds specified by *useconds*. 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, the thread will be woken up when the previously set timer interval expires.

Returned Value

If successful, usleep() returns a value of `0`.

If unsuccessful, usleep() returns a value of -1 and sets errno to the following value:

EINVAL
The *useconds* argument was greater than or equal to 1,000,000.

Related Information

- "unistd.h" on page 51
- "alarm() — Set an Alarm" on page 76
- "sigaction() — Examine or Change a Signal Action" on page 749
- "sleep() — Suspend Execution of a Process" on page 787

utime() — Set File Access and Modification Times

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

Format

```
#define _OPEN_SYS
#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, utime() sets the access and modification times 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 `time_t` type gives the number of seconds since the epoch.)
The new access time

`time_t modtime`
The new modification time

The utime() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

Returned Value

If successful, utime() returns a value of `0` and updates the access and modification times of the file to those specified.

If unsuccessful, utime() returns a value of `-1`, it does not update file times, and it sets errno to one of the following values under the conditions described:

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. The utime() function issues this error if it detects more than POSIX_SYMLOOP symbolic links (defined in the limits.h header file) 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
No file named *pathname* exists, or *pathname* 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 _OPEN_SYS
#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 28
- “utime.h” on page 52

utimes() — Set File Access and Modification Times

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

Format

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

int utimes(const char *path, const struct timeval times);
```

General Description

Sets the access and modification times of the file pointed to by *path* to the value of *times*.

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 *times* is a null pointer, utimes() sets the access and modification times of the file to the current time. The utimes() function requires the same process privilege requirements as utime(). The utimes() function updates the last file status change, field `st_ctime` in a stat(), with the current time.

Returned Value

If successful, utimes() returns a value of 0.

If unsuccessful, utimes() returns a value of -1 and sets errno to one of the following values under the conditions described.

S
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. The utimes() function issues this error if it detects more than POSIX_SYMLOOP symbolic links (defined in the limits.h header file) 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 path name string substituted for a symbolic link exceeds PATH_MAX. The PATH_MAX and NAME_MAX values can be determined using pathconf().

ENOTENT
No file named *path* exists, or *path* 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 46
- "utime() — Set File Access and Modification Times" on page 912

__utmpxname() — Change the utmpx Database Name

Standards / Extensions	C or C++	Dependencies
OpenExtensions	both	

Format

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

int __utmpxname(char *file);
```

General Description

Changes the name of the utmpx database file for the current thread from the 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 utmpxname() returns thread-specific data, you can use it safely from a multithreaded 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 calls endutxent().

Programs should 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` database back to the default value. If you must do additional `utmpx` operations on a nonstandard `utmpx` database after calling endutxent(), you must call __utmpxname() again, to re-establish the nonstandard name.

Returned Value

If successful, __utmpxname() returns a value of `0`.

If unsuccessful, __utmpxname() returns a value of -1.

Related Information

- “endutxent() — Close the utmpx Database” on page 186
- “getutxent() — Read Next Entry in utmpx Database” on page 379
- “getutxid() — Search by ID utmpx Database” on page 380
- “getutxline() — Search by Line utmpx Database” on page 381
- “pututxline() — Write Entry to utmpx Database” on page 616
- “setutxent() — Reset to Start of utmpx Database” on page 741

va_arg(), va_end(), va_start() — Access Function Arguments

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);
```

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 *variable_name* is the identifier of the rightmost named parameter in the parameter list (preceding `,` `...`). You should 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 *var_type* argument must be one of `int`, `long`, `double`, `struct`, `union`, or a pointer, or `typedef` of one of these types.

The macros also provide fixed-point decimal support. 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 not valid.

The va_end() macro is needed by some systems to indicate the end of parameter scanning.

The va_start() and va_arg() functions do not work with parameter lists of functions whose linkages were changed with the `#pragma linkage` directive.

Returned Value

The va_arg() macro returns the current argument. The va_end() and va_start() macros have no returned value.

Example

CBC3BV01: Example 1

```
/* 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 1

```
Day:  Sat
Day:  Sun
Day:  Mon

Day:  Mon
Day:  Tues
Day:  Wed
```

```
Day:  Thurs
Day:  Fri
```

CBC3BV01: Example 2

```
/* This example uses a variable number of arguments for
   fixed-point decimal data types.
 */
#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 2

```
m    = 168
va01 = 12345678.12
va02 = -987.65000
va03 = 12345678901.12345678900987654321
n    = 135
```

Related Information

- “stdarg.h” on page 39
- “vfprintf() — Format and Print Data to Stream” on page 922
- “vprintf() — Format and Print Data to stdout” on page 923
- “vsprint() — Format and Print Data to Buffer” on page 924

valloc() — Allocate Page-Aligned Memory

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

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 you use free() 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 NULL if insufficient storage is available or if *size* is `0`. If valloc() returns NULL because of insufficient storage, it will also set errno to the following value:

ENOMEM
Insufficient memory is available.

Related Information

- “stdlib.h” on page 42
- “malloc() — Reserve Storage Block” on page 477
- “sysconf() — Determine System Configuration Options” on page 856

vfork() — Create a New Process

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

Format

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

pid_t vfork(void);
```

Recommendation: Although POSIX.1 does not require the inclusion of <sys/types.h>, XPG4 has it as an optional header; therefore, you should include it for portability.

General Description

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 886.
- 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 multithread environment.

You can use VM memory files from an OpenExtensions application. However, using 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 OpenExtensions Services

If the specified file name represents an external link, the program is loaded from the caller's CMS 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 loadbfs service. In other cases, the program is loaded from the BFS.

Restrictions:

1. The fork() call in the child process must be followed by an exec().
2. You must issue the CMS command OPENVM SET FORK ON before running an application using fork(). If the CMS FORK flag is not turned on, the application will receive a return value of -1, a return code of ENOSYM, and a reason code of JRFunctNotSupported.
3. You must run the application as a POSIX(ON) application. If the Language Environment POSIX flag is not turned on, the application will receive a return value of -1, a return code of ENOSYS, and a reason code of JRFunctNotSupported.
4. The child process is *not* allowed to issue any function that will cause the child process to be blocked. Doing so will result in an X'AE6' abnormal end code, for example, a pipe read() or a pause(), before the

child issues the exec(). Any attempt to exit the child process before the exec() is issued will result in an X'AE5' abnormal end code.

5. Any local variables in the application that are changed in the child process before the exec() is issued will be changed in the parent process as well. This is due to the fact that the child and parent processes are still using the same program storage. The exec() function causes the child process to begin using its own program storage.
6. Any global and/or environment variables in the application that are changed in the child process before the exec() is issued will be changed in the parent process as well. This is due to the fact that the child and parent processes are still using the same program storage. The exec() function causes the child process to begin using its own program storage.

Returned Value

If successful, vfork() returns a value of 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, returns -1 to the parent,. and sets errno to one of the following values under the conditions described:

EAGAIN
Insufficient resources are available to create another process, or the process has already reached the maximum number of processes you can run.

ELEMSGERR
The Language Environment message file is not available.

ELENOFORK
The application contains a language that does not support fork()

ENOMEM
The process requires more space than is available.

Related Information

- “sys/types.h” on page 46
- “unistd.h” on page 51
- “alarm() — Set an Alarm” on page 76
- “fcntl() — Control Open File Descriptors” on page 207
- “putenv() — Change or Add an Environment Variable” on page 614
- “semop() — Perform Semaphore Operations” on page 692
- “setlocale() — Set Locale” on page 721
- “shmat() — Attach Shared Memory Segment” on page 743
- “sigaction() — Examine or Change a Signal Action” on page 749
- “signal() — Handle Interrupts” on page 767
- “sigpending() — Examine Pending Signals” on page 772
- “sigprocmask() — Examine or Change a Thread” on page 773
- “stat() — Get File Information” on page 803
- “system() — Execute a Command” on page 858
- “times() — Get Process and Child Process Times” on page 886

vfprintf() — Format and Print Data to Stream

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

Format

```
#include <stdio.h>

int vfprintf(FILE *stream, const char *format, va_list arg_ptr);
```

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. You should initialize these argument by using 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 264.

The vfprintf() function is not supported for files opened with `type=record`.

The vfprintf() function 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition must also occur, unless an EOF has been reached.

Returned Value

If successful, vfprintf() returns the number of characters written to *stream*.

If unsuccessful, vfprintf() returns a negative value.

Note: In contrast to some UNIX-based implementations of the C language, the IBM XL C/C++ for z/VM implementation of the vprintf() family increments the pointer to the variable arguments list. To control whether the pointer to the argument is incremented, you should 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 39
- "stdio.h" on page 40
- "va_arg(), va_end(), va_start() — Access Function Arguments" on page 916
- "vprintf() — Format and Print Data to stdout" on page 923
- "vsprint() — Format and Print Data to Buffer" on page 924

vprintf() — Format and Print Data to stdout

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

Format

```
#include <stdio.h>

int vprintf(const char *format, va_list arg_ptr);
```

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. You should initialize these arguments using 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 264.

The vprintf() function is not supported for files opened with `type=record`.

The vprintf() function 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, an intervening flush or reposition must occur. Between a read and a subsequent write, an intervening flush or reposition must also occur, unless an EOF has been reached.

Returned Value

If successful, vprintf() returns the number of characters written to `stdout`.

If unsuccessful, vprintf() returns a negative value.

Note: In contrast to some UNIX-based implementations of the C language, the IBM XL C/C++ for z/VM implementation of the vprintf() family increments the pointer to the variable arguments list. To control whether the pointer to the argument is incremented, you should 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 39
- “stdio.h” on page 40
- “va_arg(), va_end(), va_start() — Access Function Arguments” on page 916
- “vfprintf() — Format and Print Data to Stream” on page 922
- “vsprint() — Format and Print Data to Buffer” on page 924

vsprint() — Format and Print Data to Buffer

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

Format

```
#include <stdio.h>

int vsprintf(char *target-string, const char *format, va_list arg_ptr);
```

Description

The vsprint() 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 264.

Returned Value

If successful, vsprint() returns the number of characters written *target-string*.

If unsuccessful, vsprint() returns a negative value.

Note: In contrast to some UNIX-based implementations of the C language, the IBM XL C/C++ for z/VM implementation of the vprintf() family increments the pointer to the variable arguments list. To control whether the pointer to the argument is incremented, you should call the va_end macro after each call to vsprint().

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 39
- “stdio.h” on page 40
- “va_arg(), va_end(), va_start() — Access Function Arguments” on page 916
- “vfprintf() — Format and Print Data to Stream” on page 922
- “vprintf() — Format and Print Data to stdout” on page 923

vswprintf() — Write Wide Characters

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

Format

```
#include <stdarg.h>
#include <wchar.h>
```

```
int vswprintf(wchar_t *wcs, size_t n, const wchar_t *format, va_list arg);
```

XPG4

```
#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);
```

Description

The vswprintf() function is equivalent to vsprintf(), except for the following:

- *wcs* specifies an array of type `wchar_t`, rather than an array of type `char`, into which the generated output will be written.
- *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 will be converted to `wchar_t`, as if by calling mbtowc(), and then written.
- %c with an `l` prefix means a `wint_t` will be converted to `wchar_t` and then written.
- %s without an `l` prefix means a character array containing a multibyte character sequence will be converted to an array of `wchar_t` and written. The conversion will take place as if mbrtowc() were called repeatedly.
- %s with an `l` prefix means an array of `wchar_ts` will be written. The array is written up to but not including the terminating null character, unless the precision specifies a shorter output.

The vswprintf() function writes a null wide character 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 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 vswprintf() function in the wchar header available when you compile your program.

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 IBM XL C/C++ for z/VM implementation of the vprintf() family increments the pointer to the variable arguments list. To control whether the pointer to the argument is incremented, you should 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 wcstr[100];

   vout(wcstr, 100, format3, L"ONE", 2L, L"THREE");
   printf("%ls\n",wcstr);
   vout(wcstr, 100, format5, L"ONE", 2L, L"THREE", 4L, L"FIVE");
   printf("%ls\n",wcstr);
}

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 39
- “wchar.h” on page 53
- “swprintf() — Write Wide Characters to a Wide-Character Array” on page 851
- “vsprint() — Format and Print Data to Buffer” on page 924

wait() — Wait for a Child Process to End

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

Format

```
#define _OPEN_SYS
#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. The wait() function also ends if the calling process receives a signal whose action is either to execute a signal handler or to end the process.

The *status_ptr* argument 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);
```

For more information, see “waitpid() — Wait for a Specific Child Process to End” on page 930.

The wait() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

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 uexit() 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 uexit() 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. You should only use this 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, wait() returns a value of -1 and sets errno to one of the following values under the conditions described:

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() or waitpid().

EINTR
The wait() function 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 _OPEN_SYS
#include <sys/types.h>
#include <sys/wait.h>
#include <spawn.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>
```

```
main() {
  pid_t pid;
  time_t t;
  int status;

  if ((pid = spawn("childprog', 0, 0, 0, 0, 0)) < 0)
    perror("spawn() error");
  else {
    printf("parent has spawned 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 spawned child with pid of 65546
parent is starting wait at Fri Jun 16 10:53:03 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 38
- “sys/types.h” on page 46
- “sys/wait.h” on page 48
- “_exit() — End a Process and Bypass the Cleanup” on page 195
- “exit() — End Program” on page 197
- “pause() — Suspend a Thread Pending a Signal” on page 535
- “waitpid() — Wait for a Specific Child Process to End” on page 930

waitid() — Wait for Child Process to Change State

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

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

Suspends the calling process until one of its children changes state. The `waitid` function records the 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 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)*pid*.

If *idtype* is P_GID, waitid() will wait for any child with a process group ID equal to (pid_t)*pid*.

If *idtype* is P_ALL, waitid() will wait for any children and *id* is ignored.

The *options* argument specifies which state changes to wait for. You form it 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 *idtype* and *options* arguments, then the system will fill in the structure pointed to by *infop* 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,it returns `0`. Otherwise, it returns `-1` and sets errno to one of the following values under the conditions described:

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 *options*, or *idtype* and *id* specify an invalid set of processes.

Related Information

- "exec Functions" on page 192
- "exit() — End Program" on page 197
- "wait() — Wait for a Child Process to End" on page 927

waitpid() — Wait for a Specific Child Process to End

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

Format

```
#define _OPEN_SYS
#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. The waitpid() function also ends if the calling process receives a signal whose action is either to execute a signal handler or to end the process.

The waitpid() function is supported only in a POSIX program. For more information, see "IBM XL C/C++ for z/VM Applications with OpenExtensions Services" on page 7.

pid_t *pid*
Specifies the child processes the caller wants to wait for:

- If *pid* is greater than `0`, waitpid() waits for termination of the specific child whose process ID is equal to *pid*.
- If *pid* is equal to `0`, 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 `0` if the child process explicitly returns a `0` 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 manifests 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 uexit() 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 uexit() 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. You should only use this macro 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. You should only use this macro 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 the status information is not available for at least one process (usually a child), waitpid() returns a value of `0`.

If unsuccessful, waitpid() returns a value of `-1` and sets errno to one of the following values under the conditions described:

ECHILD
The process specified by *pid* does not exist or is not a child of the calling process, or the process group specified by *pid* does not exist or does not have any member process that is a child of the calling process.

EINTR
The waitpid() function was interrupted by a signal. The value of **status_ptr* is undefined.

EINVAL
The value of *options* is incorrect.

Example

CBC3BW02

```
/* CBC3BW02
   The following function suspends the calling process using waitpid()
   until a child process ends.
 */
#define _OPEN_SYS
#include <sys/types.h>
#include <sys/wait.h>
#include <spawn.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>
```

```
main() {
  pid_t pid;
  time_t t;
  int status;

  if ((pid = spawn("childprog', 0, 0, 0, 0, 0)) < 0)
    perror("spawn() error");
  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 38
- “sys/types.h” on page 46
- “sys/wait.h” on page 48
- “_exit() — End a Process and Bypass the Cleanup” on page 195
- “exit() — End Program” on page 197
- “pause() — Suspend a Thread Pending a Signal” on page 535
- “wait() — Wait for a Child Process to End” on page 927

wcrtomb() — Convert a Wide Character to a Multibyte Character

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

Format

Non-XPG4

```
#include <wchar.h>

int wcrtomb(char *s, wchar_t wchar, mbstate_t *pss);
```

XPG4

```
#define _XOPEN_SOURCE
#define _MSE_PROTOS
#include <wchar.h>

int wcrtomb(char *s, wchar_t wchar, mbstate_t *pss);
```

General Description

If *s* is a null pointer, wcrtomb() 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, wcrtomb() 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.

The wcrtomb() function is a "restartable" version of wctomb(), that is, shift state information is passed as one of the arguments and is updated on return. With wcrtomb(), 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 XPG4

If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, you must also define the _MSE_PROTOS feature test macro to make the declaration of the wcrtomb() function in the wchar header available when you compile your program.

Returned Value

If *s* is a null pointer, wcrtomb() 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, wcrtomb() returns the number of bytes stored in the array object (including any shift sequences) when `wchar` is a valid wide character. When `wchar` is not a valid wide character, an encoding error occurs, wcrtomb() returns a value of -1 and stores the value of the macro EILSEQ in errno. The conversion state remains unchanged.

Example

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
   char    *string;
   wchar_t wc;
   int     length;

   length = wcrtomb(string, wc, NULL);
}
```

Related Information

- “wchar.h” on page 53
- “mblen() — Calculate Length of Multibyte Character” on page 480
- “mbrlen() — Calculate Length of Multibyte Character” on page 482
- “mbrtowc() — Convert a Multibyte Character to a Wide Character” on page 484
- “mbsrtowcs() — Convert a Multibyte String to a Wide-Character String” on page 487
- “wcsrtombs() — Convert Wide-Character String to Multibyte String” on page 951
- “wctomb() — Convert Wide Character to Multibyte Character” on page 967

wcscat() — Append to Wide-Character String

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

Format

```
#include <wchar.h>  /*  or #include <wcstr.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. The wcscat() function does not perform bounds checking.

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.

The wcscat() 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 53

- “strcat() — Concatenate Strings” on page 809
- “wcschr() — Search for Wide-Character Substring” on page 936
- “wcscmp() — Compare Wide-Character Strings” on page 937
- “wcscpy() — Copy Wide-Character String” on page 940
- “wcscspn() — Find Offset of First Wide-Character Match” on page 941
- “wcslen() — Calculate Length of Wide-Character String” on page 944
- “wcsncat() — Append to Wide-Character String” on page 945

wcschr() — Search for Wide-Character Substring

Standards / Extensions	C or C++	Dependencies
ISO C Amendment SAA XPG4 XPG4.2	both	

Format

```
#include <wchar.h>  /*  or #include <wcstr.h>  */

wchar_t  *wcschr(const wchar_t *string1, wchar_t character);
```

Compiler Option

LANGLVL(EXTENDED), LANGLVL(SAA), or LANGLVL(SAA2)

General Description

Operates on null-terminated wide-character strings.

The wcschr() function searches *string* for the occurrence of *character*, which may be a wide null character (`\0`). The wide null character at the end of *string* is included in the search. 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.

The wcschr() 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, wcschr() returns a NULL pointer.

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 53
- "strchr() — Search for Character" on page 811
- "wcscat() — Append to Wide-Character String" on page 935
- "wcscmp() — Compare Wide-Character Strings" on page 937
- "wcscpy() — Copy Wide-Character String" on page 940
- "wcscspn() — Find Offset of First Wide-Character Match" on page 941
- "wcslen() — Calculate Length of Wide-Character String" on page 944
- "wcsncmp() — Compare Wide-Character Strings" on page 946

wcscmp() — Compare Wide-Character Strings

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

Format

```
#include <wchar.h>  /*  or #include <wcstr.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.

The wcscmp() 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

< 0
String pointed to by *string1* is less than string pointed to by *string2*.

= 0
String pointed to by *string1* is identical to string pointed to by *string2*.

> 0
String pointed to by *string1* is greater than string pointed to by *string2*.

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 53
- “strcmp() — Compare Strings” on page 812
- “wcscat() — Append to Wide-Character String” on page 935
- “wcschr() — Search for Wide-Character Substring” on page 936
- “wcscpy() — Copy Wide-Character String” on page 940
- “wcscspn() — Find Offset of First Wide-Character Match” on page 941
- “wcslen() — Calculate Length of Wide-Character String” on page 944
- “wcsncmp() — Compare Wide-Character Strings” on page 946

wcscoll() — Language Collation String Comparison

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

Format

```
#include <wchar.h>

int wcscoll(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 wcscoll() function differs from the wcscmp() function. The wcscoll() 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 wcscmp() functions performs a wide-character code to wide-character code comparison.

The wcscoll() function indicates error conditions by setting errno; however, it has no returned value to indicate an error. To check for errors, you should set errno to `0` and then check it upon return from wcscoll(). If errno is a nonzero value, an error has occurred.

You can set the EILSEQ error 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 wcscoll() 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 = wcscoll(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 53
- “setlocale() — Set Locale” on page 721
- “strcoll() — Compare Strings” on page 813

wcscpy() — Copy Wide-Character String

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

Format

```
#include <wchar.h>
#include <wcstr.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. The wcscpy() function doe not perform bounds checking.

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.

The wcscpy() 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 53

- “strcpy() — Copy String” on page 815
- “wcscat() — Append to Wide-Character String” on page 935
- “wcschr() — Search for Wide-Character Substring” on page 936
- “wcscmp() — Compare Wide-Character Strings” on page 937
- “wcscspn() — Find Offset of First Wide-Character Match” on page 941
- “wcslen() — Calculate Length of Wide-Character String” on page 944
- “wcsncpy() — Copy Wide-Character String” on page 948

wcscspn() — Find Offset of First Wide-Character Match

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

Format

```
#include <wchar.h>  /*  or #include <wcstr.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 *string1* that do not appear in *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.

The wcscspn() 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 53
- "strcspn() — Compare Strings" on page 816
- "wcscat() — Append to Wide-Character String" on page 935
- "wcschr() — Search for Wide-Character Substring" on page 936
- "wcscmp() — Compare Wide-Character Strings" on page 937
- "wcscpy() — Copy Wide-Character String" on page 940
- "wcslen() — Calculate Length of Wide-Character String" on page 944

wcsftime() — Format Date and Time

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

Formats the date and time into a wide-character string. The wcsftime() function is equivalent to the strftime() function, except that:

- The `wcs` argument specifies an array of a wide string into which the generated output is to be placed.
- The `maxsize` argument indicates a number of wide characters.
- The `*format` argument 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.

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`, wcsftime() returns the number of wide characters placed into the array pointed to by `wcs` not including the terminating null wide character. Otherwise, wcsftime() returns a value of `0`, 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 53
- “strftime() — Convert to Formatted Time” on page 821

wcsid() — Character Set ID for Wide Character

Standards / Extensions	C or C++	Dependencies
C Library	both	

Format

```
#include <stdlib.h>

int wcsid(const wchar_t c)
```

General Description

Determines the character set identifier for the specified wide character.

Returned Value

Returns the character set identifier for the wide character, or a value of -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 42

wcslen() — Calculate Length of Wide-Character String

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

Format

```
#include <wchar.h> /*  or #include <wcstr.h>  */

size_t wcslen(const wchar_t *string);
```

General Description

Computes the number of wide 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 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 53
- "mblen() — Calculate Length of Multibyte Character" on page 480
- "strlen() — Determine String Length" on page 825
- "wcsncat() — Append to Wide-Character String" on page 945
- "wcsncmp() — Compare Wide-Character Strings" on page 946
- "wcsncpy() — Copy Wide-Character String" on page 948

wcsncat() — Append to Wide-Character String

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

Format

```
#include <wchar.h>  /*  or #include <wcstr.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.

The wcsncat() 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 53
- “strncat() — Concatenate Strings” on page 827
- “wcsncmp() — Compare Wide-Character Strings” on page 946
- “wcsncpy() — Copy Wide-Character String” on page 948

wcsncmp() — Compare Wide-Character Strings

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

Format

```
#include <wchar.h>  /*  or #include <wcstr.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.

The wcsncmp() 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

< 0
String pointed to by *string1* is less than string pointed to by *string2*

0
String pointed to by *string1* is identical to string pointed to by *string2*

> 0
string pointed to by *string1* is greater than string pointed to by *string2*

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
  wcscmp: "abcdefg" is less than "abcfg"

Comparison of only the first 3 characters
  wcsncmp: "abcdefg" is identical to "abcfg"
```

Related Information

- "wchar.h" on page 53
- "strncmp() — Compare Strings" on page 828
- "wcscmp() — Compare Wide-Character Strings" on page 937
- "wcsncat() — Append to Wide-Character String" on page 945
- "wcsncpy() — Copy Wide-Character String" on page 948

wcsncpy() — Copy Wide-Character String

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

Format

```
#include <wchar.h>  /*  or #include <wcstr.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, wcsncpy() pads *string1* 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.

The wcsncpy() 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 wcscpy() 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 wcscpy works */
  printf( "destination is originally = '%ls'\n", destination );
  return_string = wcscpy( destination, source );
  printf( "After wcscpy, 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 wcscpy, destination becomes '123456789'

destination1 is originally = 'abcdefg'
After wcsncpy, destination1 becomes '12345fg'
```

Related Information

- “wchar.h” on page 53
- “strncpy() — Copy String” on page 830
- “wcscpy() — Copy Wide-Character String” on page 940
- “wcsncat() — Append to Wide-Character String” on page 945
- “wcsncmp() — Compare Wide-Character Strings” on page 946

wcspbrk() — Locate First Wide Character in String

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

Format

```
#include <wchar.h>  /*  or #include <wcstr.h>  */

wchar_t *wcspbrk(const wchar_t *string1, const wchar_t *string2);
```

General Description

Locates the first occurrence in *string1* of any character from *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.

The wcspbrk() function was formerly declared in wcstr.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 wcspbrk().
 */
#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 = wcspbrk( 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 53
- “strpbrk() — Find Characters in String” on page 831
- “wcschr() — Search for Wide-Character Substring” on page 936
- “wcscmp() — Compare Wide-Character Strings” on page 937
- “wcscspn() — Find Offset of First Wide-Character Match” on page 941
- “wcsncmp() — Compare Wide-Character Strings” on page 946
- “wcsrchr() — Locate Last Wide Character in String” on page 950
- “wcswcs() — Locate Wide-Character Substring in Wide Character String” on page 963

wcsrchr() — Locate Last Wide Character in String

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

Format

```
#include <wchar.h>  /*  or #include <wcstr.h>  */

wchar_t *wcsrchr(const wchar_t *string, wchar_t character);
```

General Description

Locates the last occurrence of *character* in *string.* The terminating null wide character is considered 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.

The wcsrchr() 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 wscrchr */
  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 53
- “strrchr() — Find Last Occurrence of Character in String” on page 836
- “wcscmp() — Compare Wide-Character Strings” on page 937
- “wcscspn() — Find Offset of First Wide-Character Match” on page 941
- “wcsncmp() — Compare Wide-Character Strings” on page 946
- “wcspbrk() — Locate First Wide Character in String” on page 949
- “wcswcs() — Locate Wide-Character Substring in Wide Character String” on page 963

wcsrtombs() — Convert Wide-Character String to Multibyte String

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

Format

Non-XPG4

```
#include <wchar.h>

size_t wcsrtombs(char *dst, const wchar_t **src, size_t len, mbstate_t *ps);
```

XPG4

```
#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) is stored.

Conversion stops 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` is 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 converted wide character. If conversion stopped due to reaching a terminating null wide character, the resulting state described is the initial conversion state.

The wcsrtombs() function is a "restartable" version of wcstombs(), that is, shift state information is passed as one 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 XPG4

If you define any feature test macro specifying XPG4 behavior before the statement in your program source file to include the wchar header, 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.

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 is unchanged. Otherwise, wcsrtombs() 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 53
- “mblen() — Calculate Length of Multibyte Character” on page 480
- “mbrlen() — Calculate Length of Multibyte Character” on page 482
- “mbrtowc() — Convert a Multibyte Character to a Wide Character” on page 484
- “mbsrtowcs() — Convert a Multibyte String to a Wide-Character String” on page 487
- “wcrtomb() — Convert a Wide Character to a Multibyte Character” on page 933
- “wcstombs() — Convert a Wide-Character String to Multibyte Character” on page 960

wcsspn() — Search for Wide Characters in a String

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

Format

```
#include <wchar.h>  /*  or #include <wcstr.h>  */

size_t wcsspn(const wchar_t *string1, const wchar_t *string2);
```

General Description

Computes the number of wide characters in the initial segment of *string1*, which consists entirely of wide characters from *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.

The wcsspn() 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

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, wcsspn() 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 = wcsspn( 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 53
- “strspn() — Search String” on page 837
- “wcscat() — Append to Wide-Character String” on page 935
- “wcschr() — Search for Wide-Character Substring” on page 936
- “wcscmp() — Compare Wide-Character Strings” on page 937
- “wcscspn() — Find Offset of First Wide-Character Match” on page 941
- “wcsncmp() — Compare Wide-Character Strings” on page 946

wcsstr() — Locate a Wide-Character Sequence

Standards / Extensions	C or C++	Dependencies
ISO C Amendment	both	

Format

Non-XPG4

```
#include <wchar.h>

wchar_t *wcsstr(const wchar_t *wcs1, const wchar_t *wcs2);
```

XPG4

```
#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 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 wcsstr() function in the wchar header available when you compile your program.

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, wcsstr() 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 53
- “strstr() — Locate Substring” on page 838

wcstod() — Convert Wide-Character String to a Double Floating-Point

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 fscanf Family of Formatted Input Functions for a description of special infinity and NaN sequences recognized by z/VM formatted input functions, including wcstod() in IEEE floating-point mode.

This function converts the initial portion of the wide-character string pointed to by `nptr` to `double` representation. First the wcstod() function 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 9899:1990 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 9899:1990 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, wcstod() returns a value of `0`. If the correct value is outside the range of representable values, wcstod() returns plus or minus HUGE_VAL—according to the sign of the value—and stores the value of the macro ERANGE in errno. If the correct value would cause underflow, wcstod() returns a value of `0` and stores the value of the macro ERANGE 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 53

wcstok() — Break a Wide-Character String into Tokens

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`, points to a wide-character string, while in subsequent calls for the same wide string, `wcs1` is a null pointer. If `wcs1` is a null pointer, the value pointed to by `ptr` matches 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 character 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.

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, IBM XL C/C++ for z/VM provides comparable restart information in runtime library storage.

Note: 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 53
- “strtok() — Tokenize String” on page 842

wcstol() — Convert a Wide-Character String to a Long Integer

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 9899:1990 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 which 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 9899:1990 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, wcstol() returns a value of 0. If the correct value is outside the range of representable values, wcstol() returns LONG_MAX or LONG_MIN (according to the sign of the value) and stores the value of the macro ERANGE 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 53
- "strtol() — Convert Character String to Long" on page 843

wcstombs() — Convert a Wide-Character String to Multibyte Character

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 wcstombs() function returns a value of `(size_t)-1` if it encounters an invalid multibyte character.

If `count` is the returned value, the array is not null terminated.

If `*dest` is a null pointer, wcstombs() returns the number of characters required to convert the wide character string.

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 42
- “mbstowcs() — Convert Multibyte Characters to Wide Characters” on page 489
- “wcslen() — Calculate Length of Wide-Character String” on page 944
- “wcsrtombs() — Convert Wide-Character String to Multibyte String” on page 951

- "wctomb() — Convert Wide Character to Multibyte Character" on page 967

wcstoul() — Convert a Wide-Character String to an Unsigned Long

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 an 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 9899:1990 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, which 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 9899:1990 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 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, wcstoul() returns a value of 0. If the correct value is outside the range of representable values, wcstoul() returns ULONG_MAX and stores the value of the macro ERANGE 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 53
- "strtoul() — Convert String to Unsigned Integer" on page 845

wcswcs() — Locate Wide-Character Substring in Wide Character String

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

Format

```
#include <wcstr.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.

The wcswcs() 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 0 length, wcswcs() 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 <wcstr.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 53
- "strstr() — Locate Substring" on page 838
- "wcschr() — Search for Wide-Character Substring" on page 936
- "wcscmp() — Compare Wide-Character Strings" on page 937
- "wcscspn() — Find Offset of First Wide-Character Match" on page 941
- "wcspbrk() — Locate First Wide Character in String" on page 949
- "wcsrchr() — Locate Last Wide Character in String" on page 950

wcswidth() — Determine the Display Width of a Wide-Character String

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 a value of `0` (if `wcs` points to a null wide character) or the number of printing positions occupied by the wide-character string pointed to by `wcs` The wcswidth() function returns a value of -1 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: The returned width 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 53

wcsxfrm() — Transform a Wide-Character String

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 that 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 wcscmp() 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 wcscoll() 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 will 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%ls\n\ntransformed according", wcs);
   printf(" to this program's locale.\n");
}
```

Related Information

- "wchar.h" on page 53
- "strxfrm() — Transform String" on page 847

wctob() — Convert Wide Character to Byte

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, you must also define the _MSE_PROTOS feature test macro to make the declaration of wctob() in the wchar header available when you compile your program.

Returned Value

Returns EOF if c does not correspond to a multibyte character with length one; otherwise, wctob() 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 53

wctomb() — Convert Wide Character to Multibyte Character

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. If *character* is not a valid multibyte character, wctomb() returns a value of -1. If *string* is a NULL pointer, the wctomb() function returns a nonzero value if shift-dependent encoding is used, or a value of `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 42
- “mbtowc() — Convert Multibyte Character to Wide Character” on page 490
- “wcrtomb() — Convert a Wide Character to a Multibyte Character” on page 933
- “wcslen() — Calculate Length of Wide-Character String” on page 944
- “wcstombs() — Convert a Wide-Character String to Multibyte Character” on page 960

wctype() — Obtain Handle for Character Property Classification

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 which requires code-page-specific type information. The wctype() function returns a value of type `wctype_t`, which can be used as the second argument to a call to 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 a value of `0` if the given property name is not valid for the current locale (category LC_CTYPE); otherwise, wctype() returns a value of type *wctype_t* that can be used in calls to iswctype().

Related Information

- "wctype.h" on page 54

wcwidth() — Determine the Display Width of a Wide Character

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.

The behavior of the wcwidth() function is affected by the LC_CTYPE category.

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.

The prototype for the XPG4 variety of the wcwidth() function is:

```
int wcwidth(const wchar_t wc);
```

The difference between this variety and the 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 a value of 0 if wc is a null wide character or returns the number of printing positions occupied by wc. If wc is not a printing wide character, wcwidth() returns a value of -1.

Note: The returned width will be 0 for a null character, 1 for a single-byte character, and 2 for a double-byte character.

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", wcwidth(wc));
}
```

Output

```
wc has a width of: 1
```

Related Information

- “wchar.h” on page 53

w_getpsent() — Get Process Data

Standards / Extensions	C or C++	Dependencies
OpenExtensions	both	

Format

```
#define _OPEN_SYS
#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, you should pass the token 0; the function

then returns the token that identifies the next process to which the caller has access. You should 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 32 on page 971 for the format of the structure stored in the buffer.

Table 32. 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_pgpid*	Process group ID
pid_t	*ps_fgpid*	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_systime*	System CPU time
int	*ps_conttylen*	Length of ConTTY
char	**ps_conttyptr*	Controlling terminal
int	*ps_pathlen*	Length of *arg0*
char	**ps_pathptr*	File name
int	*ps_cmdlen*	Length of command
char	**ps_cmdptr*	Command and arguments

The w_getpsent() function is supported only in a POSIX program. For more information, see “IBM XL C/C++ for z/VM Applications with OpenExtensions Services” on page 7.

Returned Value

If 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 a value of `0`, indicating that no more processes are to be accessed.

If unsuccessful, w_getpsent returns a value of -1 and sets errno to EINVAL, which indicates an incorrect process token.

Example

CBC3BW33

```
/* CBC3BW33
   This example provides status information, using wgetpsent().
 */
#define _OPEN_SYS
#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 = 1024);
  buf.ps_pathptr   = (char *) malloc(buf.ps_pathlen   = 1024);
  buf.ps_cmdptr    = (char *) malloc(buf.ps_cmdlen    = 1024);
  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 45

wordexp() — Perform Shell Word Expansions

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	POSIX(ON)

Format

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

```
int wordexp(const char *words, wordexp_t *pwordexp, int flags);

void wordfree(wordexp_t *pwordexp);
```

General Description

Performs word expansions as described in *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 *words*.
char **	we_wordv	Pointer to a list of expanded words.
size_t	we_offs	Slots to reserve at the beginning of *pwordexp*->we_wordv.

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 path name expansion (see the *XCU* specification, Section 2.6.6, "Path name 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.

You must 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().

You use the *flags* argument to control the behavior of wordexp(). The values of *flags* is the bitwise inclusive OR or `0` 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 you set this flag, *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.

You can use the WRDE_APPEND flag to append a new set of words to those generated by a previous call to wordexp().

Rules: The following rules apply when you make two or more calls to wordexp() 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 a value of `0`.

If unsuccessful, wordexp() returns a nonzero value, as defined in <wordexp.h> and described above, 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 54

wordfree() — Perform Shell Word Expansions

Standards / Extensions	C or C++	Dependencies
XPG4 XPG4.2	both	POSIX(ON)

Format

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

int wordexp(const char *words, wordexp_t *pwordexp, int flags);

void wordfree(wordexp_t *pwordexp);
```

General Description

Frees any memory associated with *pwordexp* from a previous call to wordexp(). For the rules governing use of wordexp() and wordfree(), see the description of wordexp().

Returned Value

The wordfree() function has no returned value.

Related Information

- "wordexp.h" on page 54

write() — Write Data on a File or Socket

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

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. No file offset is 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 not enough space is available to write the requested number of bytes (for example, because not enough space is available 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 a value of -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 as 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 a value of *N*; or else it writes nothing, returns a value of -1, and sets errno to EAGAIN. 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, returns a value of -1, and sets errno to EAGAIN.

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 returns a value of -1 and sets errno to EAGAIN.

The write() function causes the SIGTTOU signal 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.

fs
The file or socket descriptor.

buf
The pointer to the buffer holding the data to be written.

N
The length in bytes or the buffer pointed to by the *buf* parameter.

If not enough buffer space is available 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 value of -1 and sets errno to EWOULDBLOCK. For a description of how to set the nonblocking mode, see "fcntl() — Control Open File Descriptors" on page 207 or "ioctl() — Control Device" on page 417.

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 value of 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 returns a value of -1 and sets errno to EAGAIN.

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*.

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 incomplete.

If unsuccessful, write() returns a value of -1 and sets errno to one of the following values under the conditions described.

EAGAIN
Resources are 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 virtual machine.

EFBIG
Writing to the output file would exceed the maximum file size supported by the implementation.

EINTR
A signal interrupted write() before it had written any output.

EINVAL
The request is invalid or not supported.

EIO
One of the following conditions applies:

- The process is in a background process group and is attempting to write to its controlling terminal, but TOSTOP (defined in the termios.h 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 has 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
No space is available on the output device.

ENOTCONN
The socket is not connected.

EPIPE
The write() function 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.

EWOULDBLOCK
The socket is in nonblocking mode, and data is not available to write.

Example

CBC3BW35

```
/* CBC3BW35
   This example writes a certain amount of bytes to a file via 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 25
- “termios.h” on page 49
- “unistd.h” on page 51
- “connect() — Connect a Socket” on page 138
- “creat() — Create a New File or Rewrite an Existing One” on page 146"
- “dup() — Duplicate an Open File Descriptor” on page 180
- “fcntl() — Control Open File Descriptors” on page 207
- “fwrite() — Write Items” on page 308
- “getsockopt() — Get Options Associated with a Socket” on page 370
- “ioctl() — Control Device” on page 417
- “lseek() — Change the Offset of a File” on page 469
- “open() — Open a File” on page 527
- “pipe() — Create an Unnamed Pipe” on page 539
- “read() — Read From a File ” on page 628
- “readv() — Read Data on a Socket and Store in a Set of Buffers” on page 635
- “recv() — Receive Data on a Socket” on page 640
- “recvfrom() — Receive Messages on a Socket” on page 642
- “recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers” on page 645
- “select() — Monitor Activity on Files/Sockets and Message Queues” on page 680
- “selectex() — Monitor Activity on Files/Sockets and Message Queues” on page 685
- “send() — Send Data on a Socket” on page 694
- “sendmsg() — Send Messages on a Socket” on page 697
- “sendto() — Send Data on a Socket” on page 700
- “setsockopt() — Set Options Associated with a Socket” on page 734
- “socket() — Create a Socket” on page 791
- “writev() — Write Data on a Socket from an Array” on page 979

writev() — Write Data on a Socket from an Array

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

Format

X/Open

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

ssize_t writev(int socket, const struct iovec *iov, int iovcnt);
```

Berkeley Sockets

```
#define _OE_SOCKETS
#include <sys/uio.h>

int writev(int socket, struct iovec *iov, int iovcnt);
```

General Description

Writes data to a socket with descriptor *socket* 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.

socket
The socket descriptor.

iov
A pointer to an array of `iovec` buffers.

iovcnt
The number of buffers pointed to by the *iov* parameter.

The `iovec` structure is defined in uio.h and contains the following elements:

iov_base
A pointer to the buffer.

iov_len
The length of the buffer.

This call writes the sum of the *iov_len* bytes of data.

If not enough buffer space is available 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 value of `-1` and sets errno to EWOULDBLOCK. For a description of how to set nonblocking mode, see "fcntl() — Control Open File Descriptors" on page 207 or "ioctl() — Control Device" on page 417.

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 value of `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 returns a value of `-1` and sets errno to EAGAIN.

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, writev() returns the number of bytes written from the buffer.

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 incomplete.

If unsuccessful, writev() returns a value of `-1` and sets errno to one of the following values under the conditions described:

EAGAIN
Resources are temporarily unavailable. Subsequent requests may complete normally.

EBADF
socket is not a valid 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 *iov* and *iovcnt* parameters would result in an attempt to access storage outside the caller's virtual machine.

EINTR
A signal interrupted writev() before any data was transmitted.

EINVAL
An incorrect value for *iovcnt* was detected.

EMSGSIZE
The message was too big to be sent as a single datagram.

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.

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
The socket is in nonblocking mode, and data buffers are not available.

Related Information

- “connect() — Connect a Socket” on page 138
- “fcntl() — Control Open File Descriptors” on page 207
- “getsockopt() — Get Options Associated with a Socket” on page 370
- “ioctl() — Control Device” on page 417
- “read() — Read From a File ” on page 628
- “readv() — Read Data on a Socket and Store in a Set of Buffers” on page 635
- “recv() — Receive Data on a Socket” on page 640
- “recvfrom() — Receive Messages on a Socket” on page 642
- “recvmsg() — Receive Messages on a Socket and Store in an Array of Message Headers” on page 645
- “select() — Monitor Activity on Files/Sockets and Message Queues” on page 680
- “selectex() — Monitor Activity on Files/Sockets and Message Queues” on page 685
- “send() — Send Data on a Socket” on page 694
- “sendmsg() — Send Messages on a Socket” on page 697
- “sendto() — Send Data on a Socket” on page 700
- “setsockopt() — Set Options Associated with a Socket” on page 734

- "socket() — Create a Socket" on page 791
- "write() — Write Data on a File or Socket" on page 975

w_statvfs() — Get File System Status

Standards / Extensions	C or C++	Dependencies
OpenExtensions	both	

Format

```
#define _OPEN_SOURCE 2
#include <sys/statvfs.h>

int w_statvfs(const char *filesystem, struct statvfs *buffer, size_t buflen);
```

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 for the mount C Library function.
- The name specified in the *file_system_name* parameter for the BPX1MNT Callable Service.
- The name specified in an OPENVM MOUNT command.

buffer
A buffer into which the status information is put. 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 806.

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 a value of `-1` and sets errno to the following value under the conditions described:

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 46
- "fstatvfs() — Get File System Information" on page 297
- "mount() — Make a File System Available" on page 508
- "statvfs() — Get File System Information" on page 806

y0(), y1(), yn() — Bessel Functions of the Second Kind

Standards

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 *x* argument must be positive. The *n* argument should be greater than or equal to zero. If *n* is less than zero, the result will contain a negative exponent.

Note: This function works in both IEEE floating-point and hexadecimal floating-point formats. See "IEEE floating-point" on page 63 for more information about IEEE floating-point.

Returned Value

For y0(), y1(), or yn(), if *x* is negative, the function sets errno to EDOM and returns -HUGE_VAL. For y0(), y1(), or yn(), if *x* causes overflow, the function sets errno to ERANGE and returns -HUGE_VAL.

The calculated value is returned otherwise.

Special Behavior for IEEE

If *x* is negative, y0(), y1(), and yn() return the value NaNQ. If *x* is 0, y0(), y1(), and yn() return the value -HUGE_VAL. In all cases, errno remains unchanged.

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 32
- “erf() - erfc() — Calculate Error and Complementary Error Functions” on page 189
- “gamma() — Calculate Gamma Function” on page 310
- “j0(), j1(), jn() — Bessel Functions of the First Kind” on page 438

Library Functions for the System Programming C Facilities

The library functions specific to the System Programming C environment are described in *z/OS: XL C/C++ Programming Guide*. These system programming functions are as follows:

- _xhotc()
- _xhotl()
- _xhott()
- _xhotu()
- _xregs()
- _xsacc()
- _xsrvc()
- _xusr()
- _xusr2()
- _24malc()
- _4kmalc()

Appendix A. Table of C Macros

You can use the macros listed in Table 33 on page 985 to write programs that use built-in services of the IBM XL C/C++ for z/VM product. The general-purpose macros, labeled *General*, are predefined 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 IBM XL C/C++ for z/VM functionality. Internal macros are defined in corresponding include files.

Table 33. z/VM C Macros

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

© Copyright IBM Corp. 2002, 2024

Table 33. z/VM C Macros (continued)		
Macro	**Include File**	**General or 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
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

Table 33. z/VM C Macros (continued)		
Macro	**Include File**	**General or Internal**
__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
__ctypec	ctype.h	Internal
__ctypec_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

Table 33. z/VM C Macros (continued)		
Macro	**Include File**	**General or Internal**
__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
__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

Table 33. z/VM C Macros (continued)		
Macro	**Include File**	**General or Internal**
dynfree(x)	dynit.h	General
dyninit(__dynp)	dynit.h	General
__dynit	dynit.h	Internal
EDOM	errno.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
EXIT_FAILURE	stdlib.h	General
EXIT_SUCCESS	stdlib.h	General
exp(x)	math.h	General
__F	dynit.h	General
_F_	dynit.h	General
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
__floath	float.h	Internal
flocate(w,x,y,z)	stdio.h	General
FLT_DIG	float.h	General

Table 33. z/VM C Macros (continued)		
Macro	**Include File**	**General or Internal**
__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
IN6_IS_ADDR_LINKLOCAL	netinet/in.h	General

Table 33. z/VM C Macros (continued)		
Macro	**Include File**	**General or Internal**
IN6_IS_ADDR_LOOPBACK	netinet/in.h	General
IN6_IS_ADDR_MC_GLOBAL	netinet/in.h	General
IN6_IS_ADDR_MC_LINKLOCAL	netinet/in.h	General
IN6_IS_ADDR_MC_NODELOCAL	netinet/in.h	General
IN6_IS_ADDR_MC_ORGLOCAL	netinet/in.h	General
IN6_IS_ADDR_MC_SITELOCAL	netinet/in.h	General
IN6_IS_ADDR_MULTICAST	netinet/in.h	General
IN6_IS_ADDR_SITELOCAL	netinet/in.h	General
IN6_IS_ADDR_UNSPECIFIED	netinet/in.h	General
IN6_IS_ADDR_V4COMPAT	netinet/in.h	General
IN6_IS_ADDR_V4MAPPED	netinet/in.h	General
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
__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

Table 33. z/VM C Macros (continued)		
Macro	**Include File**	**General or Internal**
__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

Table 33. z/VM C Macros (continued)		
Macro	**Include File**	**General or Internal**
__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
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

Table 33. z/VM C Macros (continued)		
Macro	**Include File**	**General or 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
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

Table 33. z/VM C Macros (continued)		
Macro	**Include File**	**General or Internal**
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
MON_11	nl_langinfo.h	General
MON_12	nl_langinfo.h	General
__MSGFILE	stdio.h	General
__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

Table 33. z/VM C Macros (continued)		
Macro	**Include File**	**General or Internal**
__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
_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

Table 33. z/VM C Macros (continued)		
Macro	**Include File**	**General or Internal**
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

Table 33. z/VM C Macros (continued)		
Macro	**Include File**	**General or Internal**
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
__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
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

Table 33. z/VM C Macros (continued)		
Macro	**Include File**	**General or Internal**
__TOUPPER_INDEX	ctype.h	Internal
towlower	wchar.h	General
towupper	wchar.h	General
__TPUT_WRITE	stdio.h	General
__TRK	dynit.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_PAT	stdio.h	General
__VSAM_OPEN_FAIL	stdio.h	General

Table 33. z/VM C Macros (continued)		
Macro	**Include File**	**General or Internal**
__VSAM_OPEN_KSDS	stdio.h	General
__VSAM_OPEN_KSDS_PAT	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
YESEXPR	nl_langinfo.h	General

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

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 grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

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 jurisdictions 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 websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide 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 Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

© Copyright IBM Corp. 2002, 2024

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 document 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.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

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.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain 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 actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information may contain sample application programs in source language, which illustrate programming techniques on various operating platforms. 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 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. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Programming Interface Information

This publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM XL C/C++ for z/VM and z/VM.

Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., in the United States and/or other countries. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on IBM Copyright and trademark information (https://www.ibm.com/legal/copytrade).

UNIX is a registered trademark of The Open Group in the United States and other countries.

Terms and Conditions for Product Documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal Use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see:

- The section entitled **IBM Websites** at IBM Privacy Statement (https://www.ibm.com/privacy)
- Cookies and Similar Technologies (https://www.ibm.com/privacy#Cookies_and_Similar_Technologies)

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.

Bibliography

This topic lists the publications in the z/VM library. For abstracts of the z/VM publications, see *z/VM: General Information*.

Where to Get z/VM Information

The current z/VM product documentation is available in IBM Documentation - z/VM (https://www.ibm.com/docs/en/zvm).

z/VM Base Library

Overview

- *z/VM: License Information*, GI13-4377
- *z/VM: General Information*, GC24-6286

Installation, Migration, and Service

- *z/VM: Installation Guide*, GC24-6292
- *z/VM: Migration Guide*, GC24-6294
- *z/VM: Service Guide*, GC24-6325
- *z/VM: VMSES/E Introduction and Reference*, GC24-6336

Planning and Administration

- *z/VM: CMS File Pool Planning, Administration, and Operation*, SC24-6261
- *z/VM: CMS Planning and Administration*, SC24-6264
- *z/VM: Connectivity*, SC24-6267
- *z/VM: CP Planning and Administration*, SC24-6271
- *z/VM: Getting Started with Linux on IBM Z*, SC24-6287
- *z/VM: Group Control System*, SC24-6289
- *z/VM: I/O Configuration*, SC24-6291
- *z/VM: Running Guest Operating Systems*, SC24-6321
- *z/VM: Saved Segments Planning and Administration*, SC24-6322
- *z/VM: Secure Configuration Guide*, SC24-6323

Customization and Tuning

- *z/VM: CP Exit Customization*, SC24-6269
- *z/VM: Performance*, SC24-6301

Operation and Use

- *z/VM: CMS Commands and Utilities Reference*, SC24-6260
- *z/VM: CMS Primer*, SC24-6265
- *z/VM: CMS User's Guide*, SC24-6266
- *z/VM: CP Commands and Utilities Reference*, SC24-6268

© Copyright IBM Corp. 2002, 2024

- *z/VM: System Operation*, SC24-6326
- *z/VM: Virtual Machine Operation*, SC24-6334
- *z/VM: XEDIT Commands and Macros Reference*, SC24-6337
- *z/VM: XEDIT User's Guide*, SC24-6338

Application Programming

- *z/VM: CMS Application Development Guide*, SC24-6256
- *z/VM: CMS Application Development Guide for Assembler*, SC24-6257
- *z/VM: CMS Application Multitasking*, SC24-6258
- *z/VM: CMS Callable Services Reference*, SC24-6259
- *z/VM: CMS Macros and Functions Reference*, SC24-6262
- *z/VM: CMS Pipelines User's Guide and Reference*, SC24-6252
- *z/VM: CP Programming Services*, SC24-6272
- *z/VM: CPI Communications User's Guide*, SC24-6273
- *z/VM: ESA/XC Principles of Operation*, SC24-6285
- *z/VM: Language Environment User's Guide*, SC24-6293
- *z/VM: OpenExtensions Advanced Application Programming Tools*, SC24-6295
- *z/VM: OpenExtensions Callable Services Reference*, SC24-6296
- *z/VM: OpenExtensions Commands Reference*, SC24-6297
- *z/VM: OpenExtensions POSIX Conformance Document*, GC24-6298
- *z/VM: OpenExtensions User's Guide*, SC24-6299
- *z/VM: Program Management Binder for CMS*, SC24-6304
- *z/VM: Reusable Server Kernel Programmer's Guide and Reference*, SC24-6313
- *z/VM: REXX/VM Reference*, SC24-6314
- *z/VM: REXX/VM User's Guide*, SC24-6315
- *z/VM: Systems Management Application Programming*, SC24-6327
- *z/VM: z/Architecture Extended Configuration (z/XC) Principles of Operation*, SC27-4940

Diagnosis

- *z/VM: CMS and REXX/VM Messages and Codes*, GC24-6255
- *z/VM: CP Messages and Codes*, GC24-6270
- *z/VM: Diagnosis Guide*, GC24-6280
- *z/VM: Dump Viewing Facility*, GC24-6284
- *z/VM: Other Components Messages and Codes*, GC24-6300
- *z/VM: VM Dump Tool*, GC24-6335

z/VM Facilities and Features

Data Facility Storage Management Subsystem for z/VM

- *z/VM: DFSMS/VM Customization*, SC24-6274
- *z/VM: DFSMS/VM Diagnosis Guide*, GC24-6275
- *z/VM: DFSMS/VM Messages and Codes*, GC24-6276
- *z/VM: DFSMS/VM Planning Guide*, SC24-6277

- *z/VM: DFSMS/VM Removable Media Services*, SC24-6278
- *z/VM: DFSMS/VM Storage Administration*, SC24-6279

Directory Maintenance Facility for z/VM

- *z/VM: Directory Maintenance Facility Commands Reference*, SC24-6281
- *z/VM: Directory Maintenance Facility Messages*, GC24-6282
- *z/VM: Directory Maintenance Facility Tailoring and Administration Guide*, SC24-6283

Open Systems Adapter

- Open Systems Adapter/Support Facility on the Hardware Management Console (https://www.ibm.com/docs/en/SSLTBW_2.3.0/pdf/SC14-7580-02.pdf), SC14-7580
- Open Systems Adapter-Express ICC 3215 Support (https://www.ibm.com/docs/en/zos/2.3.0?topic=osa-icc-3215-support), SA23-2247
- Open Systems Adapter Integrated Console Controller User's Guide (https://www.ibm.com/docs/en/SSLTBW_2.3.0/pdf/SC27-9003-02.pdf), SC27-9003
- Open Systems Adapter-Express Customer's Guide and Reference (https://www.ibm.com/docs/en/SSLTBW_2.3.0/pdf/ioa2z1f0.pdf), SA22-7935

Performance Toolkit for z/VM

- *z/VM: Performance Toolkit Guide*, SC24-6302
- *z/VM: Performance Toolkit Reference*, SC24-6303

The following publications contain sections that provide information about z/VM Performance Data Pump, which is licensed with Performance Toolkit for z/VM.

- *z/VM: Performance*, SC24-6301. See z/VM Performance Data Pump.
- *z/VM: Other Components Messages and Codes*, GC24-6300. See Data Pump Messages.

RACF® Security Server for z/VM

- *z/VM: RACF Security Server Auditor's Guide*, SC24-6305
- *z/VM: RACF Security Server Command Language Reference*, SC24-6306
- *z/VM: RACF Security Server Diagnosis Guide*, GC24-6307
- *z/VM: RACF Security Server General User's Guide*, SC24-6308
- *z/VM: RACF Security Server Macros and Interfaces*, SC24-6309
- *z/VM: RACF Security Server Messages and Codes*, GC24-6310
- *z/VM: RACF Security Server Security Administrator's Guide*, SC24-6311
- *z/VM: RACF Security Server System Programmer's Guide*, SC24-6312
- *z/VM: Security Server RACROUTE Macro Reference*, SC24-6324

Remote Spooling Communications Subsystem Networking for z/VM

- *z/VM: RSCS Networking Diagnosis*, GC24-6316
- *z/VM: RSCS Networking Exit Customization*, SC24-6317
- *z/VM: RSCS Networking Messages and Codes*, GC24-6318
- *z/VM: RSCS Networking Operation and Use*, SC24-6319
- *z/VM: RSCS Networking Planning and Configuration*, SC24-6320

TCP/IP for z/VM

- *z/VM: TCP/IP Diagnosis Guide*, GC24-6328
- *z/VM: TCP/IP LDAP Administration Guide*, SC24-6329
- *z/VM: TCP/IP Messages and Codes*, GC24-6330
- *z/VM: TCP/IP Planning and Customization*, SC24-6331
- *z/VM: TCP/IP Programmer's Reference*, SC24-6332
- *z/VM: TCP/IP User's Guide*, SC24-6333

Prerequisite Products

Device Support Facilities

- Device Support Facilities (ICKDSF): User's Guide and Reference (https://www.ibm.com/docs/en/SSLTBW_2.5.0/pdf/ickug00_v2r5.pdf), GC35-0033

Related Products

XL C++ for z/VM

- *XL C/C++ for z/VM: Runtime Library Reference*, SC09-7624
- *XL C/C++ for z/VM: User's Guide*, SC09-7625

z/OS

IBM Documentation - z/OS (https://www.ibm.com/docs/en/zos)

Index

Special Characters

__CURRENT macro 42
__CURRENT_LOWER macro 42
__ftp.h header file 26
__LOWER macro 42
_ALL_SOURCE feature test macro 15
_ALL_SOURCE_NO_THREADS feature test macro 15
_atoe_l() library function 87
_atoe() library function 87
_Ccsid.h header file 19
_CS_PATH symbolic constant 137
_CS_SHELL symbolic constant 137
_csplist macro 151
_CURRENT clrmemf() argument 131
_CURRENT LOWER clrmemf() argument 131
_EDC_COMPAT environment variable 234, 292, 909
_etoa() library function 191
_etoal() library function 191
_exit() library function 195
_ftchep entry point in fetchep() 231
_ICONV_UCS2 environment variable 402
_ICONV_UCS2_PREFIX environment variable 402
_Ieee754.h header file 27
_librel() library function 449
_longjmp() library function 463
_LOWER clrmemf() argument 131
_MSE_PROTOS feature test macro 15
_Nascii.h header file 34
_OE_SOCKETS feature test macro 15
_OPEN_DEFAULT feature test macro 15
_OPEN_SOURCE feature test macro 14
_OPEN_SYS feature test macro 14
_OPEN_SYS_IPC_EXTENSIONS feature test macro 14
_OPEN_SYS_PTY_EXTENSIONS feature test macro 14
_OPEN_SYS_SOCK_EXT feature test macro 15
_OPEN_SYS_SOCK_IPV6 15
_OPEN_THREADS feature test macro 14
_POSIX_C_SOURCE feature test macro 14
_POSIX1_SOURCE feature test macro 14
_setjmp() library function 717
_signgam() library function 771
_utmpxname() library function 915
_VARARG_EXT_ feature test macro 15
_XOPEN_SOURCE feature test macro 14
_XOPEN_SOURCE_EXTENDED feature test macro 14

A

a64l() library function 90
abend
    user 66
abnormal program termination 66
abort() library function 66
abs() library function 68
absf() library function 68
absl() library function 68
absolute value
    long integer 443
absolute value, calculating 68
acc parameter for fopen() 256
accept() library function 69
access mode, fopen() 254
access() library function 71
acos() library function 73
acosf() library function 73
acosh() library function 75
acosl() library function 73
address, host 335
AF_INET domain
    example 97, 142
    servers 139
    socket descriptor created in 93
AF_INET6 domain
    example 142
    servers 139
    socket descriptor created in 94
AF_IUCV domain
    example 98, 142
    servers 139
    socket descriptor created in 94
AF_UNIX domain
    example 98, 143
    servers 140
    socket descriptor created in 95
aio.h header file 18
alarm() library function 76
AMODE, switching 223
arccosine library function 73
arccosine, calculating 73
arcsine, calculating 79
arctangent, calculating 83
arguments, accessing 916
arpa/inet.h header file 18
arpa/nameser.h header file 18
asctime() library function 78
asin() library function 79
asinf() library function 79
asinh() library function 81
asinl() library function 79
assert.h header file 18, 81
assert() macro 18, 81
assertion diagnostic 81
assigning buffers 703
asynchronous
    signal catching 752
    signal, waiting for an 784
at (@) character, mapping to 61
atan() library function 83
atan2() library function 83
atan2f() library function 83
atan2l() library function 83
atanf() library function 83
atanh() library function 84

atanl() library function 83
atexit() library function 85
atof() library function 88
atoi(); library function 89
atol() library function 89
attributes, terminal 869
audit flags 114, 201

B

base 10 logarithm 462
base e logarithm 460
base-64 representation to long integer, converting 90
basename() library function 91
baud rate
    input
        determining 108
        termios 112
    output
        determining 110
        termios 113
bcmp() library function 91
bcopy() library function 92
bessel library functions
    first kind 438
    second kind 983
binary
    files 256
    search 100
    tree
        deleting 882
        finding node 884, 896
        traversing binary search 899
        walk 899
bind() library function 92
blank character 427, 433
blksize parameter 256
bpxhmnt.h header file 19
break condition 873
brk() library function 98
broadcast, unblocking a thread 560
bsd_signal() library function 99
bsearch() library function 100
buffers
    assigning 703
    BUFSIZ macro 41
    comparing 493
    copying 495, 496
    data stored in 635
    flushing 233
    formatting and printing data 924
    receiving data and storing in 640
    receiving messages and storing in 642, 645
    searching 492
    setting characters 497
BUFSIZ macro 41
bytes, copying 849
byteseek parameter in fopen() 256
bzero() library function 102

C

C macros 985
calloc() library function 103
cancel a thread 555
cancelability
    point, establishing 610
    PTHREAD_INTR_ASYNCHRONOUS type 555
    PTHREAD_INTR_CONTROLLED type 555
    PTHREAD_INTR_DISABLE type 555
    PTHREAD_INTR_ENABLE type 555
canonical input processing 877
cassert header file 19
category argument of setlocale 721
cbrt() library function 104
cclass() library function 104
CCSID (coded character set ID) 19
cctype header file 19
cds() library function 106
cdump() library function 106
ceeedcct.h header file 19
ceil() library function 107
ceilf() library function 107
ceiling of value, determining 107
ceill() library function 107
cerrno header file 19
cfgetispeed() library function 108
cfgetospeed() library function 110
cfloat header file 19
cfsetispeed() library function 112
cfsetospeed library function 113
character
    classification with ctype.h 22
    classifying 419
    finding in a string 673, 831
    multibyte
        conversion using mbrtowc() 484
        conversion using mbstowcs() 489
        conversion using mbtowc() 490
        length 480, 482
    property 434
    property classification 968
    reading
        with fgetc() 235
        with getc() and getchar() 312
        with getwc() 384
        with getwchar() 385
        without echo 352
    searching for 407, 811
    setting 497
    testing 419
    ungetting 907, 908
    writing with
        fputc() 272
        fputs() 273
        fputwc() 274
        fputws() 276
character string to double, converting 88
character string to integer, converting 89
character string to long, converting 89
characters mapping from 61
characters mapping to 61
chaudit() library function 114
chdir() library function 116
child process 259
chmod() library function 117
chown() library function 120

CICS (Customer Information Control System)
    cics.h header file 19
    verify running 428
ciso646 header file 20
classifying characters 419
cleanup thread handler 557, 558
clear error indicators 124
clearenv() library function 122
clearerr() library function 124
climits header file 20
clocale header file 20
clock ticking in times() library function 887
clock() library function 125
CLOCKS_PER_SEC 125
close() library function 127
closedir() library function 129
closing
    streams 537
clrmemf() library function
    associated macros 42
cmath header file 20
code set conversion utilities
    iconv, usage 399
coded character set
    ID (CCSID) 19
collate.h header file 20
collating elements
    get next 345
    maximum 479
    multicharacter 429
    wide character 386
collequiv() library function 132
collorder() library function 133
collrange() library function 134
colltostr() library function 136
commands, invoking from library function 858
comparing
    buffers 493
    bytes in memory 91
    strings 809, 812, 813, 816, 826, 828
comparison functions, cs() 148
compressing
    dn_comp() 174
concatenation
    strings 809, 827
concurrent access 211
condition
    attribute object
        destroying an 568
        initializing an 571
    variable
        waiting on a 566
        waiting on for a limited time 564
configuration
    system 856
configuration variable 533
confstr() library function 137
connect() library function 138
connection between sockets 138
connection request 69
connection, duplex shutdown 748
control block information 244
controlling terminal, path name 151
conversions
conversions *(continued)*
    character
        multibyte to wide with mbrtowc() 484
        multibyte to wide with mbstowcs() 489
        multibyte to wide with mbtowc() 490
        string to double 840
        to lowercase 895, 896
        to uppercase 895, 896
    code set 399
    date and time 322, 832
    double to string 311
    EBCDID bytes to ISO8859-1 191
    EBCDID character string to ISO8859-1 191
    floating-point numbers to integers and fractions 507
    Internet address
        binary to text 413
        text to binary 414
    long integer to base-64 string representation 474
    specifier
        argument in fscanf() and scanf() library function 285
        fscanf() and scanf() library functions 284
        used by strftime() 822
        used by strptime() 832
    string to unsigned integer 845
    string, multibyte to wide 487
    time structure to string 78
    time to character string 153
    wide character to multibyte 933
convert double to string 183
converting
    ISO8859 character to EBCDIC 87
    ISO8859–1 bytes to EBCDIC 87
copying
    bytes 495, 496, 849
    bytes in memory 92, 492
    strings 815, 830
cos() library function 143
cosf() library function 143
cosh() library function 145
coshf() library function 145
coshl() library function 145
cosine
    calculating 143
    hyperbolic, calculating 145
cosl() library function 143
cpio.h header file 20
creat() library function 146
creating
    socket 791
    symbolic link to path name 854
    temporary file 888, 889
    thread key identifiers 584
    threads 573
cs() library function 148
csetjmp header file 21
csid() library function 149
csignal header file 21
csnap() library function 150
csp.h header file 21
csplist macro 21
cstdarg header file 21
cstddef header file 21
cstdio header file 21
cstdlib header file 22

cstring header file 22
ctermid() library function 151
ctest.h header file 22
ctest() library function 152
ctime header file 22
ctime() library function 153
ctrace() library function 155
ctype.h header file 22, 419
current file position, changing 292
current host address 340
current terminal 344
cuserid() library function 156
cwchar header file 23
cwctype header file 23

D

data
    buffers, stored in 635
    receiving 640
    sending on socket 700
    storing in buffers 640
data items
    reading 277
data set
    host information 183, 712
    network information 728
    network services, closing 185
data types
    fixed-point decimal 23
    floating-point 25, 282
    limits 28
database
    closing 158
    deleting record 159
    error indicator
        checking 159
        clearing 157
        indicator 671
    getting
        content 160
        first key 161
        next key 162
    group 330, 331
    opening 163
    storing record 164
    user 361, 362
    utmpx
        closing 186
        finding current user's slot 898
        reading next entry 379
        renaming 915
        resetting to start 741
        searching by ID 380
        searching by line 381
        writing entry to 616
datagram
    sending on socket 694
date conversion 832
date format 942
dbm_clearerr() library function 157
dbm_close() library function 158
dbm_delete() library function 158
dbm_error() library function 159
dbm_fetch() library function 160
dbm_firstkey() library function 161
DBM_INSERT symbolic constant 165
dbm_nextkey() library function 162
dbm_open; library function 163
DBM_REPLACE symbolic constant 165
dbm_store() library function 164
deadlocks 212
Debug Tool library function 152
debugging
    ctest.h 22
    with __errno2(), reason codes 190
decimal
    fixed-point operations in decimal.h 23
decimal host address
    from network number 411
decimal.h header file 23
default locale category 724
delete
    mutex object 588
deleting a mutex object 588
deleting VSAM records 216
descriptor, socket 93, 94
destroying
    condition variable attribute objects 568
    condition variables 561
    mutex attribute objects 595
    thread attributes object 544
destructor routine 584
detach a thread 575
detachstate attribute
    getting 545
    setting 551
device ID 471, 803
diagnostics, specifying 81
difftime() library function 166
directories (HFS)
    closing 129
    entry removal 910
    mode
        changing 117
    opening 531
    reading with readdir() 632
    removing 674
    renaming 661
    reporting the parent of a path name 167
    repositioning 672
    rewinding 672
    working 320
directory operations 23
dirent.h header file 23
dirname() library function 167
div_t structure 168
div() library function 168
division
    integral 446
dlclose() library function 168
dlerror() library function 169
dll.h header file 23
dlopen() library function 170
dlsym() library function 172
dn_comp() library function 174
dn_expand() library function 175
dn_find() library function 176

dn_skipname() library function 177
DNS (Domain Name Server) 665, 666, 668, 669
domain
    servers in the AF_INET 139
    servers in the AF_INET6 139
    servers in the AF_IUCV 139
    servers in the AF_UNIX 140
domain name
    compression 174
    expansion 175
    find 176
    skipping 177
Domain Name Server (DNS) 665, 666, 668, 669
drand48() library function 178
dtconv structure 31, 454
dumps, formatted 106
dup() library function 180
dup2() library function 181
duplex connection 748
dynamic
    allocations in dynit.h 23
    function call 231
dynit.h header file 23

E

ecvt() library function 183
effective ID
    group 325, 711
    user 327, 739
endhostent() library function 183
ending a process or program 66
endnetent() library function 184
endprotoent() library function 185
endservent() library function 185
endutxent() library function 186
env.h header file 24
environ 615
environment
    table 326
    variables
        _EDC_COMPAT 234, 292, 909
        adding, deleting, or changing 614, 708
        setting 24
environment variables
    clearing 122
EOF (end of file)
    clearing 671
    indicator reset 124
eof macro 41
erand48() library function 187
erf() library function 189
erfc() library function 189
errno
    in perror() 538
    values 24
errno values 64
errno.h header file 24
errno2 library function 190
ESDS (Entry-Sequenced Data
        Set)
    use of 307
establish
    cancelability point 610
establish *(continued)*
    cleanup thread handler 558
examples
    online examples 63
exception handling
    clearerr() library function 124
    signal.h header file 38
exception header file 24
exec family of functions
    described 192
    execl() library function 192
    execle() library function 192
    execlp() library function 192
    execv() library function 192
    execve() library function 192
    execvp() library function 192
    invoking POSIX programs 7
exit a thread 577
EXIT_FAILURE macro 43
EXIT_SUCCESS macro 43
exit() library function 197
exiting a program 197
exp() library function 199
expanding
    dn_expand() 175
expf() library function 199
expl() library function 199
expm1() library function 200
exponential functions 199

F

fabs() library function 200
fabsf() library function 200
fabsl() library function 200
fchaudit() library function 201
fchmod() library function 203
fchown() library function 204
fclose() library function 206
fcntl.h header file 25
fcntl() library function 207
fcvt() library function 215
fdelrec() library function 216
fdopen() library function 217
feature test macro 13
features.h header file 25
FECB (fetch control block) 231
feof() library function 219
ferror() library function 220
fetch
    control block 231
    fetchable module, program flow 222
fetch() library function
    alternatives under C++ 224
    examples of alternatives under C++ 226
FETCHABLE preprocessor directive 223
fetchep() library function 231
fflush() library function 233
ffs() library function 235
fgetc() library function 235
fgetpos() library function 236
fgets() library function 238
fgetwc() library function 239
fgetws() library function 241

FIFO
    creating special files 499, 501
file
    network services, opening 732
    protocol, opening 731
file descriptor, controlling 207
file system
    mounting a 508
    removing 903
FILE type 40
file.h header file 44
FILENAME_MAX macro 41
fileno() library reference 242
files
    changing mode 120
    closing 127
    descriptor
        duplicate 180, 181
        open stream 242
    descriptor flags 208
    errors in 124
    locking 211, 213, 458
    maximum opened 41
    name
        length 41
        temporary 883, 889
    offset 527, 628, 975
    opening 254
    positioning 236, 292, 294, 300, 671
    record locking 458
    renaming 661
    setting access and modification times 914
    status flags 209
    time access 912
    type=record used with putwc() or putwchar() 621
    writing to 975
finding
    dn_find() 176
    domain name 176
fixed-point decimal
    decimal.h 23
flags
    audit 114
    file descriptor 210
    open
        append mode 211
        asynchronous update 211
        extracting 211
        file access mode 211
        synchronous update 211
fldata_t data structure elements 245
fldata() library function
    associated macros 41
float.h header file
    constants defined in 25
    defined 25
floating-point
    breaking down value 282
    breaking up value 507
    data type 25
    IEEE 63
flocate() library function
    associated macros 41
floor() library function 249
floorf() library function 249
floorl() library function 249
flushing, buffers 233
fmod() library function 250
fmodf() library function 250
fmodl() library function 250
fmtmsg.h header file 26
fmtmsg() library function 251
fnmatch.h header file 26
fnmatch() library function 253
FOPEN_MAX macro 41
fopen() library function
    maximum simultaneous files 41
fork() library function 259
format specification
    fprintf family 264
    fscanf() and scanf() library functions 284
format specification for fprintf() family 265
formatted I/O 264
formatted time 821
fpathconf() library function 261
fpos_t type in stdio.h file 40
fprintf() library function 264
fputc() library function 272
fputs() library function 273
fputwc() library function 274
fputws() library function 276
fpxcp.h header file 26
fread() library function 277
free() library function 278
freeaddrinfo() library function 279
freeing storage 278
freeing storage allocated by glob() 391
freopen() library function 280
frexp() library function 282
fscanf() library function 283
fseek() library function 292
fsetpos() library function 294
fstat() library function 296
fstatvfs() library function 297
fsync() library function 299
ftell() library function 300
ftime() library function 302
ftok() library function 302
ftruncate() library function 303
FTW_CHDIR symbolic constant 520
FTW_D symbolic constant 305, 521
FTW_DEPTH symbolic constant 521
FTW_DNR symbolic constant 305, 521
FTW_DP symbolic constant 521
FTW_F symbolic constant 305, 521
FTW_MOUNT symbolic constant 521
FTW_NS symbolic constant 306, 521
FTW_PHYS symbolic constant 521
FTW_SL symbolic constant 305, 521
FTW_SLN symbolic constant 521
ftw.h header file 26
ftw() library function 305
functions
    arguments 916
    spawn() 795
    spawnp() 795
fupdate() library function 307
fwrite() library function 308

G

gai_strerror() library function 309
gamma() library function 310
gcvt() library function 311
gderr() library function 324
get
    condition variable attribute object 569
    file position, fgetpos() 236
getaddrinfo() library function 313
GETALL symbolic constant 688
getc() library function 312
getchar() library function 312
getclientid() library function 316
getcontext() library function 318
getcwd() library function 320
getdate() library function 322
getdtablesize() library function 324
getegid() library function 325
getenv() library function 326
geteuid() library function 327
getgid() library function 329
getgrgid() library function 330
getgrnam() library function 331
getgroups() library function 332
getgroupsbyname() library function 334
gethostbyaddr() library function 335
gethostbyname() library function 337
gethostent() library function 339
gethostid() library function 340
gethostname() library function 340
getibmsockopt() library function 341
getlogin() library function 344
getmccoll() library function 345
getnameinfo() library function 346
GETNCNT symbolic constant 688
getnetbyaddr() library function 347
getnetbyname() library function 348
getnetent() library function 350
getopt() library function 351
getpagesize() library function 352
getpass() library function 352
getpeername() library function 353
getpgrp() library function 354
GETPID symbolic constant 688
getpid() library function 355
getppid() library function 356
getprotobyname() library function 358
getprotobynumber() library function 359
getprotoent() library function 360
getpwnam() library function 361
getpwuid() library function 362
gets() library function 364
getservbyname() library function 365
getservbyport() library function 366
getservent() library function 367
getsockname() library function 368
getsockopt() library function 370
getstablesize() library function 374
getsubopt() library function 375
getsyntx() library function 376
gettimeofday() library function 377
getuid() library function 378
getutxent() library function 379
getutxid() library function 380
getutxline() library function 381
GETVAL symbolic constant 688
getw() library function 383
getwc() library function 384
getwchar() library function 385
getwmccoll() library function 386
GETZCNT symbolic constant 688
givesocket() library function 387
GLOB_ABORTED symbolic constant 390
GLOB_APPEND symbolic constant 389
GLOB_DOOFFS symbolic constant 389
GLOB_ERR symbolic constant 390
GLOB_MARK symbolic constant 390
GLOB_NOCHECK symbolic constant 390
GLOB_NOESCAPE symbolic constant 390
GLOB_NOMATCH symbolic constant 390
GLOB_NOSORT symbolic constant 390
GLOB_NOSPACE symbolic constant 390
glob.h header file 26
glob() library function 389
globfree() library function 391
gmtime() library function 391
group database, getgrgid() library function 330
group database, getgrnam() library function 331
group ID
    effective 325
    job control 729
    process 354, 731
    real 329
    setting 706
    supplementary 332, 334
grp.h header file 26

H

handle, character property class 968
handling interrupt signals 767
hash tables
    creating 393
    destroying 394
    searching 394
hcreate() library function 393
hdestroy() library function 394
header files
    __ftp.h header file 26
    __ussos.h header file 48
    _Ccsid.h header file 19
    _Ieee754.h header file 27
    _Nascii.h header file 34
    aio.h header file 18
    arpa/inet.h header file 18
    arpa/nameser.h header file 18
    assert.h header file 18
    bpxhmnt.h header file 19
    cassert header file 19
    cctype header file 19
    ceeedcct.h header file 19
    cerrno header file 19
    cfloat header file 19
    cics.h header file 19
    ciso646 header file 20
    climits header file 20
    clocale header file 20

header files *(continued)*
cmath header file 20
collate.h header file 20
cpio.h header file 20
csetjmp header file 21
csignal header file 21
csp.h header file 21
cstdarg header file 21
cstddef header file 21
cstdio header file 21
cstdlib header file 22
cstring header file 22
ctest.h header file 22
ctime header file 22
ctype.h header file 22
cwchar header file 23
cwctype header file 23
decimal.h header file 23
dirent.h header file 23
dll.h header file 23
dynit.h header file 23
env.h header file 24
errno.h header file 24
exception header file 24
fcntl.h header file 25
features.h header file 25
float.h header file 25
fmtmsg.h header file 26
fnmatch.h header file 26
fpxcp.h header file 26
ftw.h header file 26
glob.h header file 26
grp.h header file 26
iconv.h header file 27
ims.h header file 27
inttypes.h header file 27
iso646.h header file 27
langinfo.h header file 28
lc_core.h header file 28
lc_sys.h header file 28
leawi.h header file 28
libgen.h header file 28
limits.h header file 28
localdef.h header file 30
locale.h header file 30
math.h header file 32
memory.h header file 33
monetary.h header file 33
msgcat.h header file 33
mtf.h header file 34
ndbm.h header file 34
net/if.h header file 34
net/rtrouteh.h header file 34
netdb.h header file 34
netinet/in.h header file 34
new header file 34
new.h header file 35
nl_langinfo.h header file 36
nl_types.h header file 36
nlist.h header file 35
poll.h header file 36
pthread.h header file 36
pwd.h header file 37
re_comp.h header file 37

header files *(continued)*
regex.h header file 37
regexp.h header file 38
resolv.h header file 38
rexec.h header file 38
saiucv.h header file 38
search.h header file 38
setjmp.h header file 38
signal.h header file 38
spawn.h header file 39
spc.h header file 39
stdarg.h header file 39
stddef.h header file 40
stdefs.h header file 40
stdio.h header file 40
stdlib.h header file 42
string.h header file 43
strings.h header file 44
stropts.h header file 44
sys/__cpl.h header file 44
sys/__getipc.h header file 44
sys/acl.h header file 44
sys/file.h header file 44
sys/ioctl.h header file 44
sys/ipc.h header file 45
sys/layout.h header file 45
sys/mman.h header file 45
sys/modes.h header file 45
sys/msg.h header file 45
sys/ps.h header file 45
sys/resource.h header file 45
sys/sem.h header file 45
sys/shm.h header file 46
sys/socket.h header file 46
sys/stat.h header file 46
sys/statfs.h header file 46
sys/statvfs.h header file 46
sys/time.h header file 46
sys/timeb.h header file 46
sys/times.h header file 46
sys/ttydev.h header file 46
sys/types.h header file 46
sys/uio.h header file 48
sys/un.h header file 48
sys/utsname.h header file 48
sys/wait.h header file 48
sys/wlm.h header file 48
syslog.h header file 44
tar.h header file 49
terminat.h header file 49
termios.h header file 49
time.h header file 49
typeinfo header file 50
typeinfo.h header file 51
ucontext.h header file 51
uheap.h header file 51
ulimit.h header file 51
unexpect.h header file 51
unistd.h header file 51
utime.h header file 52
utmpx.h header file 52
varargs.h header file 52
variant.h header file 52
wchar.h header file 53

header files *(continued)*
wcstr.h header file 54
wctype.h header file 54
wordexp.h header file 54
xti.h header file 54
HFS (Hierarchical File System)
adding system 508
large files 459, 470, 472
host address 335
host byte order
long integer, translating to 525
short integer, translating to 526
host information data set
closing 183
opening 712
host name 337
host name entry 339
hsearch() library function 394
htonl() library function 395
htons() library function 396
HUGE_VAL macro 33
hyperbolic arccosine, calculating 75
hyperbolic arcsine, calculating 81
hyperbolic arctangent, calculating 84
hyperbolic cosine, calculating 145
hyperbolic sine, calculating 786
hyperbolic tangent, calculating 862
hypot() library function 397

I

I/O
errors 124
opening files 254
write to a file 975
ibmsflush() library function 398
iconv_close() library function 401
iconv_open() library function 402
iconv.h header file 27
iconv() library function 399
IEEE floating-point 63
if_freenameindex() library function 404
if_indextoname() library function 404
if_nameindex() library function 405
if_nametoindex() library function 406
ilogb() library function 406
ims.h header file 27
in.h header file 34
index() library function 407
indicators, error 124
inet_addr() library function 408
inet_lnaof() library function 409
inet_makeaddr() library function 410
inet_netof() library function 410
inet_network() library function 411
inet_ntoa() library function 412
inet_ntop() library function 413
inet_pton() library function 414
initialization of
a mutex 589
a mutex attribute object 598
a state array using seed 801
condition attribute objects 571
condition variables 562
initialization of *(continued)*
pseudorandom numbers 444, 679, 801
res_init() 663
state array 415
strings 830
thread attributes objects 549
initstate() library function 415
inode 471, 803
input
baud rate
determining 108
termios 112
insque() library function 416
integer
division 446
long absolute value 443
long, translated to host byte order 525
pseudo-random 187
pseudorandom 178, 436, 444, 467, 510, 523, 626, 627, 679, 801
short, translated to host byte order 526
translate to a 7-bit ASCII Character 890
translating unsigned short 396
wide 432
internationalization header file 30
Internet
servers 665, 666, 668–670
internet host address
creating 410
decimal 412
from network number 410
into network byte order 408
interrupt signal 767
intrinsic functions 63
inttypes.h header file 27
invoke a function once 601
invoking commands from a library function 858
ioctl.h header file 44
ioctl() library function 417
IOFBF macro 41
IOLBF macro 41
IONBF macro 41
IP address
resolution 663
IPC_CREAT symbolic constant 513, 690, 746
IPC_EXCL symbolic constant 513, 690, 747
IPC_NOWAIT symbolic constant 515, 517, 519, 693
IPC_PRIVATE symbolic constant 690, 746
IPC_RMID symbolic constant 512, 689, 745
IPC_SET symbolic constant 512, 689, 744
IPC_STAT symbolic constant 512, 689, 744
ipc.h header file 45
isalnum() library function 419
isalpha() library function 419
isascii() library function 421
isatty() library function 426
isblank() library function 427
iscics() library function 428
iscntrl() library function 419
isdigit() library function 419
isgraph() library function 419
islower() library function 419
ismccollel() library function 429
isnan() library function 431

ISO646 20
iso646.h header file 27
ISO8859 character to EBCDIC, converting 87
ISO8859–1 bytes to EBCDIC, converting 87
isprint() library function 419
ispunct() library function 419
isspace() library function 419
isupper() library function 419
iswalnum() library function 432
iswblank() library function 433
iswcntrl() library function 432
iswctype() library function 434
iswdigit() library function 432
iswgraph() library function 432
iswlower() library function 432
iswprint() library function 432
iswpunct() library function 432
iswspace() library function 432
iswupper() library function 432
iswxdigit() library function 432
isxdigit() library function 419

J

j0() library function 438
j1() library function 438
jn() library function 438
job control process group ID 729
jrand48(); library function 436

K

key identifier
    creating thread-specific data key 584
    getting the specific value 579
    setting the specific value 608
keyword parameters 256
kill() library function 439
killpg() library function 441
kind attribute
    getting from a mutex attribute object 596
    setting from a mutex attribute object 599
KSDS (Key Sequenced Data Set)
    PATH 307

L

L_ctermid macro 41
L_tmpnam macro 41
l64a() library function 474
labs() library function 443
langinfo.h header file
    described 28
    items values defined in 28
language collation string comparison 938
Language Environment
    effect of setlocale 725
lc_core.h header file 28
LC_CTYPE locale variable 241
LC_MONETARY locale variable 820
LC_SYNTAX locale variable 376, 819
lc_sys.h header file 28
lcong48() library function 444
lconv structure, elements of 30
ldexp() library function 445
ldexpf() library function 445
ldexpl() library function 445
ldiv() library function 446
LEAWI_INCLUDED macro 28
leawi.h header file 28
length function 825
lfind() library function 447
lgamma() library function 448
libgen.h header file 28
library release 449
limits
    resource 28
limits.h header file 28
line
    mode 877
    reading with fgets() 238
    reading with fgetws() 241
    reading.with fgetwc() 239
    writing with puts() 615
link count 450
link() library function 450
LINKLOAD EXEC 222
listen() library function 452
load module
    fetchep() library function 231
    release() library function 657
load, search order for CMS LOAD 223
local time corrections 456
localdef.h header file 30
localdtconv() library function 453
locale
    _Ieee754.h header file 27
    categories
        LC_CTYPE locale variable 241
    character class 104
    collate.h header file 20
    collating elements
        converting 136
        equivalent 132
        list of 133
        rangelist 134
    default category 724
    elements
        converting collating 136
        equivalent collating 132
        list of collating 133
        rangelist of collating 134
    fpxcp.h header file 26
    iconv.h header file 27
    ims.h header file 27
    information not retrievable 522
    iso646.h header file 27
    library functions
        localeconv() 454
    locale.h header file 30
    nl_langinfo.h header file 36
    nl_types.h header file 36
    null-string ("") category 723
    retrieving information 522
    setlocale() library function 721
    strxfrm() library function 847
    time.h header file 49

locale *(continued)*
variant.h header file 52
localeconv() library function 454
localtime() library function 456
locating storage 278
lock
attempt to a mutex object 592
wait for on a mutex object 590
lockf() library function 458
log() library function 460
log10() library function 462
log10f() library function 462
log10l() library function 462
log1p() library function 461
logb() library function 461
logf() library function 460
logic errors 81
login name 344
logl() library function 460
long integer, translating 395
longjmp() library function 465
lrand48() library function 467
LRECL (logical record length) parameter
fopen() library function 256
LRECL parameter for fopen() library function
LRECL (logical record length) parameter
fopen() library function 256
lsearch() library function 468
lseek() library function 469
lstat() library function 471

M

macros
accessing arguments, variable-length lists 39
csplist 21
defined in assert.h 18
defined in csp.h 21
defined in dynit.h 23
defined in errno.h 24
defined in langinfo.h 28
defined in leawi.h 28
defined in locale.h 30
defined in math.h 32
defined in nl_langinfo.h 36
defined in regex.h 37
defined in signal.h 38
defined in stdarg.h 39
defined in stddef.h 40
defined in stdio.h 41
defined in stdlib.h header 43
defined in string.h header 45
defined in sys/modes.h header 45
defined in time.h header 49
defined in unistd.h header 52
defined in wchar.h header 53
feature test 13
HUGE_VAL 33
LEAWI_INCLUDED 28
NULL 40
offset of 40
OMIT_FC 28
regular expressions 37
use with _amrc structure 42
macros *(continued)*
use with clrmemf() 42
use with fldata() 41
use with flocate() 41
WEOF 54
macros, C 985
makecontext() library function 475
malloc() library function 477
matching failure 288
math functions
bessel library functions of the first kind 438
bessel library functions of the second kind 983
math.h header file 32
maxcoll() library function 479
maxdesc() library function 479
maximum
file names 41
opened files 41
temporary file name 41
MB_CUR_MAX macro 43
mblen() library function 480
mbrlen() library function 482
mbrtowc() library function 484
mbsinit() library function 486
mbsrtowcs() library function 487
mbstowcs() library function 489
mbtowc() library function 490
memccpy() library function 492
memchr() library function 492
memcmp() library function 493
memcpy() library function 495
memmove() library function 496
memory files, clearing 131
memory.h header file 33
memset() library function 497
messages
receiving and storing in buffers 642, 645
sending on socket 697
miscellaneous functions 81
mkdir() library function 498
mkfifo() library function 499
mknod() library function 501
mkstemp() library function 503
mktemp() library function 504
mktime() library function 505
mman.h header file 45
mntent.h header file 45
mode, changing 117, 203
modes.h header file 45
modf() library function 507
monetary.h header file 33
mount() library function 508
mrand48() library function 510
MSG_NOERROR symbolic constant 515, 519
msg.h header file 45
msgcat.h header file 33
msgctl() library function 512
msgget() library function 513
msgrcv() library function 515
msgsnd() library function 516
msgxrcv() library function 518
MTF (multitasking facility)
functions 34
mtf.h header file 34

multibyte characters;character set identifier 149
multiple entry points 231
mutex
  attribute object
    destroying a 595
    initializing a 598
    kind attribute, getting a 596
    kind attribute, setting a 599
  object
    deleting a 588
    initializing a 589
    lock, attempt to 592
    lock, wait for a 590
    unlocking a 593
MVS (Multiple Virtual System)
  compiling fetched modules 224

N

name, binding to a socket 92
natural logarithm 460
ndbm.h header file 34
NDEBUG compiler option 81
net/if.h header file 34
net/rtrouteh.h header file 34
netdb.h header file 34
netinet/in.h header file 34
network byte order 395, 396, 408
network entry 348
network information data set
  closing 184
  opening 728
network name 347
network number
  getting decimal host address 411
  getting internet host address 410
network protocol information data set 185
network service
  by name 365
  by port 366
network services information data set
  closing 185
network services information file
  opening 732
new header file 34
new.h header file 35
nextafter() library function 520
nftw() library function 520
nl_langinfo.h header file 36
nl_langinfo() library function 522
nl_types.h header file 36
nlist.h header file 35
nonlocal goto 463, 465, 717, 718
nonrecursive mutex 596, 599
nrand48() library function 523
ntohl() library function 525
ntohs() library function 526
NULL macro 40, 41
NULL pointer 40, 41
NULL pointer constant 43
null-string locale category 723

O

OFFSET compiler option 155
offset of macro 40
OMIT_FC macro 28
OPEN_MSGQ_EXT feature test macro 15
open() library function 527
opendir() library function 531
opening
  files 254
  streams 254, 280
options, socket 734
output
  baud rate
    determining 110
    termios 113
ownership of files/directories 120, 204

P

parameters, understanding the socket() 792
parent process ID 356
password structure 37
path name
  matching 253
  matching a pattern 389
  of controlling terminal 151
pathconf() library function 533
pause() library function 535
pclose() library function 537
perror() library function 538
pipe() library function 539
poll.h header file 36
popen() library function 541
position options for flocate() library function 247
positional parameters 254
POSIX
  function call from non-POSIX function 7
  making use of 7
pow() library function 543
power functions 445
precision argument, fprintf() family 266
printf() library function 264
process
  control from within programs 858
  creating 259
  data 970
  group 733
  group ID 354, 731, 733
  group ID, foreground 870, 880
  group leader 733
  ID 355, 356, 733
  signal 439
protocol
  getting name by name 358
  getting name by number 359
  getting next entry 360
  information filet, opening 731
ps.h header file 45
pseudo-random integers 187
pseudorandom integers 178, 436, 444, 467, 510, 523, 626, 627, 679
pthread_attr_destroy() library function 544
pthread_attr_getdetachstate() library function 545

pthread_attr_getstacksize() library function 547
pthread_attr_getweight_np() library function 548
pthread_attr_init() library function 549
pthread_attr_setdetachstate() library function 551
pthread_attr_setstacksize() library function 552
pthread_attr_setweight-np() library function 554
pthread_cancel() library function 555
pthread_cleanup_pop() library function 557
pthread_cleanup_push() library function 558
pthread_cond_broadcast() library function 560
pthread_cond_destroy() library function 561
pthread_cond_init() library function 562
pthread_cond_signal() library function 563
pthread_cond_timedwait() library function 564
pthread_cond_wait() library function 566
pthread_condattr_destroy() library function 568
pthread_condattr_getkind_np() library function 569
pthread_condattr_init() library function 571
pthread_condattr_setkind_np() library function 572
pthread_create() library function 573
pthread_detach() library function 575
pthread_equal() library function 576
pthread_exit() library function 577
pthread_getspecific() library function 579
PTHREAD_INTR_ASYNCHRONOUS cancelability type 555
PTHREAD_INTR_CONTROLLED cancelability type 555
PTHREAD_INTR_DISABLE cancelability type 555
PTHREAD_INTR_ENABLE cancelability type 555
pthread_join_d4_np() library function 582
pthread_join() library function 581
pthread_key_create() library function 584
pthread_kill() library function 586
pthread_mutex_destroy() library function 588
pthread_mutex_init() library function 589
pthread_mutex_lock() library function 590
pthread_mutex_trylock() library function 592
pthread_mutex_unlock() library function 593
pthread_mutexattr_destroy() library function 595
pthread_mutexattr_getkind_np() library function 596
pthread_mutexattr_init() library function 598
pthread_mutexattr_setkind_np() library function 599
pthread_once() library function 601
pthread_self() library function 603
pthread_setintr() library function 604
pthread_setintrtype() library function 606
pthread_setspecific() library function 608
pthread_testintr() library function 610
pthread_yield() library function 611
pthread.h header file
    symbols defined in 37
ptrdiff_t type in stddef header file 40
pushing characters back onto input stream 907
putc() library function 613
putchar() library function 613
putenv() library function 614
puts() library function 615
pututxline() library function 616
putw() library function 618
putwc() library function 619
putwchar() library function 620
pwd.h header file 37

Q

qsort() library function 622
quick sort 622

R

raise() library function 623
RAND_MAX macro 43
rand() library function 626
random
    access 292, 300
    changing generator for 739
    initializing generator for 415
    number generator 178, 187, 436, 467, 510, 523, 626, 627, 800
    number initializer 444, 679, 801
random() library function 627
re_comp.h header file 37
re_comp() library function 648
re_exec() library function 650
read operations with fgetc() 235
read() library function 628
readdir() library function 632
reading
    buffers, data stored in 635
    character from stdin 235, 312, 384, 385
    character from stream 312, 384, 385
    data items from stream 277
    data, and storing in buffers 635
    directory, readdir() library function 632
    formatted 283
    from file, read() library function 628
    line from stdin 364
    line from stream 238, 239, 241
    line with fgets() 238
    line with fgetwc() 239
    line with fgetws() 241
    lock while reading a file 211
    read a string, fgets() library function 238
    scanning 283
    string without echo 352
    value of symbolic link, readlink() library function 633
readlink() library function 633
readv() library function 635
real
    group ID 329, 711
    user ID 378, 739
realloc() library function 637
reallocation of block size 637
realpath() library function 639
reason codes
    debugging with __errno2() 190
receiving
    data and storing in buffers 640
    messages and storing in buffers 642, 645
recfm parameter 256
record
    format parameter 256
recursive mutex 596, 599
recv() library function 640
recvfrom() library function 642
recvmsg() library function 645
redirection

redirection *(continued)*
streams, using freopen() 280
reentrant modules
modules, fetched 222
REG_EXTENDED macro 37
REG_ICASE macro 37
REG_NEWLINE macro 37
REG_NOSUB macro 37
REG_NOTEOL macro 37
regcomp() library function 651
regerror() library function 653
regex.h header file 37
regexec() library function 655
regexp.h header file 38
regfree() library function 657
regular expressions 37, 648, 651
release
__librel() library function 449
level macro 449
release processor to other threads 611
release() library function 657
remainder
floating-point 250
remainder of division 168, 658
remainder() library function 658
remove cleanup thread handler 557
remove() library function 659
remque() library function 660
rename() library function 661
renaming files 661
reopening streams 280
res_init() library function 663
res_mkquery() library function 665
res_query() library function 666
res_querydomain() library function 667
res_search() library function 668
res_send() library function 669
reserved names 61
resolv.h header file 38
resolver function
build domain name 667
initialization 663
query DNS 665, 666
search query 668
send query 669
resource limits defined 28
resource.h header file 45
response match 676
rewind a stream 671
rewind() library function 671
rewinddir() library function 672
rexec.h header file 38
rindex() library function 673
rint() library function 674
rmdir() library function 674
rounding
down 249
rounding up 107
rpmatch() library function 676
RRDS (Relative Record Data Set) 307

S

S_IRGRP symbolic constant 513, 690, 747
S_IROTH symbolic constant 514, 691, 747
S_IRUSR symbolic constant 513, 690, 747
S_ISBLK(mode) macro 472, 804
S_ISCHR(mode) macro 472, 804
S_ISDIR(mode) macro 472, 804
S_ISEXTL(mode) macro 472, 804
S_ISFIFO(mode) macro 472, 804
S_ISLNK(mode) macro 472, 804
S_ISREG(mode) macro 472, 804
S_ISSOCK(mode) macro 804
S_IWGRP symbolic constant 514, 691, 747
S_IWOTH symbolic constant 514, 691, 747
S_IWUSR symbolic constant 513, 690, 747
safety, signal 749
saiucv.h header file 38
saved set-user-ID 739
sbrk() library function 677
scalb() library function 678
scanf() library function 283
search.h header file 38
searching
binary tree 896
buffers 492
for character 673
qsort 622
res_search() 668
strings 811, 831
strings for tokens 842
traversing binary tree 899
seed for random numbers 800
seed48() library function 679
SEEK_CUR macro
effects of ungetc() and ungetwc() 292
SEEK_END macro 41
SEEK_SET macro 41
select()library function 680
selectex() library function 685
SEM_UNDO symbolic constant 694
sem.h header file 45
semaphore operations 687, 690, 692
semctl() library function 687
semget() library function 690
semop() library function 692
send a signal to a process group 441
send a signal to a thread 586
send() library function 694
sending
res_send() 669
sendmsg() library function 697
sendto() library function 700
serial number 471, 803
server
AF_INET6 domain 139
in the AF_INET domain 139
in the AF_IUCV domain 139
in the AF_UNIX domain 140
incoming client requests 452
session leader 733
set a condition variable attribute object 572
SETALL symbolic constant 689
setbuf() library function 703
setcontext() library function 704
setegid() library function 706
setenv() library function 708

seteuid() library function 710
setgid() library function 711
sethostent() library function 712
setibmsockopt() library function 713
setjmp.h header file 38
setjmp() library function 718
setlocale() library function 721
setnetent() library function 728
setpgid() library function 729
setpgrp() library function 731
setprotoent() library function 731
setservent() library function 732
setsid() library function 733
setsockopt() library function 734
setstate() library function 739
setuid() library function 739
setutxent library function 741
SETVAL symbolic constant 688
setvbuf() library function 742
shared memory
    attaching 743
    controlling 744
    detaching 745
    getting 746
SHM_RDONLY symbolic constant 743
SHM_RND symbolic constant 743
shm.h header file 46
shmat() library function 743
shmctl() library function 744
shmdt() library function 745
shmget() library function 746
shut down duplex connection 748
shutdown() library function 748
sig argument in signal() library function 767
SIG_DFL signal action 777
SIG_HOLD 777
SIG_IGN signal action 777
sigaction() library function 749
sigaddset() library function 754
SIGALRM signal 76
sigaltstack() library function 756
sigdelset() library function 757
sigemptyset() library function 758
sigfillset() library function 760
sighold() library function 761
sigignore() library function 762
siginterrupt() library function 762
sigismember() library function 763
siglongjmp() library function 764
signal
    adding to a thread's signal mask 761
    bsd version 99
    changing an action 776
    changing mask and suspending thread 778, 781
    handler 767
    interrupting functions 762
    mask 535
    pending 772
    removing from a thread 776
    safety 749
    sending to a process group 441
    sending to a thread 586
    sets 754
    setting and/or getting from alternate stack 756
signal *(continued)*
    setting stack 780
    setting to ignore 762
    unblocking a thread 563
    unblocking and waiting for 771
signal.h header file 38
signal() library function 767
sigpause() library function 771
sigpending() library function 772
sigprocmask() library function 773
sigrelse() library function 776
sigset() library function 776
sigsetjmp() library function 778
sigstack() library function 780
sigsuspend() library function 781
SIGTTOU signal in tcdrain() library function 863
sigwait() library function 784
sin() library function 785
sine
    calculating 785
    hyperbolic, calculating 786
sinh() library function 786
size_t structure 40
sleep() library function 787
sock_debug_bulk_perf0() library function 789
sock_debug() library function 789
sock_do_bulkmode() library function 790
sock_do_teststor() library function 791
socket
    creating 791
    data, sending on 700
    data, writing 979
    datagrams, sending on 694
    descriptor AF_UNIX domain 95
    descriptor in AF_INET domain 93
    descriptor in AF_INET6 domain 94
    descriptor in AF_IUCV domain 94
    getting name 368
    messages, sending on 697
    operating characteristics, specifying 417
    options, getting 370
    options, setting 734
    peer connected to 353
    send data on 694, 700
    send messages on 697
    shutdown 748
    writing data on 979
socket.h header file 46
socket() library function 791
socket() parameters, understanding the 792
sorting 622
space= parameter 256
spawn.h header file 39, 798
spawn() 795
spawnp() 795
spc.h header file 39
special file, create a 501
specific value for a key
    getting 579
    setting 608
sprintf() library function 264
square root function 799
srand() library function 800
srand48() library function 801

srandom() library function 801
SS_DISABLE symbolic constant 756
SS_ONSTACK symbolic constant 756
sscanf() library function 283
stack
  restoring the environment 465
  saving an environment 718
  setting and/or getting signal from alternate 756
  setting signal 780
stacksize attribute
  getting 547
  setting 552
standard streams 40, 280
standards, indicated by table 61
START
  character 875
stat structure 803
stat.h header file 46
stat() library function 803
statfs.h header file 46
status analysis macros 928, 931
statvfs.h header file 46
statvfs() library function 806
stdarg.h header file 39
stddef.h header file 40
stdefs.h header file 40
stdio.h header file 40
stdlib.h header file 42
stdout, formatting and printing data 923
STIMER REAL TQE 125
stop bits 876
STOP character 875
stopping a process or program 66
storage
  allocation 39
  allocation, library function, calloc() 103
  free 278
  free, allocated by glob() 391
  locating 278
  reserving with malloc() 477
strcasecmp() library function 809
strcat() library function 809
strchr() library function 811
strcmp() library function 812
strcoll() library function 813
strcpy() library function 815
strcspn() library function 816
strdup() library function 817
streams
  access mode 280
  binary mode 280
  buffering 703
  changing current file position 292, 300
  changing file position 671
  closing 537
  format and print data 922
  formatted I/O 264, 283
  initiating pipe 541
  opening 254
  reading
    characters with fgetc() 235
    characters with getc() and getchar() 312
    characters with getwc() 384
streams *(continued)*
  reading *(continued)*
    characters with getwchar() 385
    data items with fread() 277
    lines with fgets() 238
    lines with fgetwc() 239
    lines with fgetws() 241
    lines with gets() 364
  redirection 280
  reopening 280
  rewinding 671
  text mode 280
  translation mode 280
  ungetting characters 907
  ungetting wide characters 908
  updating 254, 280
  writing
    characters with fputc() 272
    characters with fputwc() 274
    characters with putc() and putchar() 613
    lines with puts() 615
    strings 273, 276
    words with putw() 618
strerror() library function 817
strfmon() library function 818
strftime() library function 821
string.h header file 43
strings
  classification with ctype.h 22
  classifying 419
  comparing
    language collation 938
  concatenating 809, 827
  converting
    case 895, 896
    EBCDIC character string to ISO8859-1 191
    long integer to base-64 representation 474
    of double 311
    to double 840
    to double floating point 955
    to long integer 959
    to multibyte 960
    to unsigned integer 845
    to unsigned long integer 962
  copying 815, 830
  duplicating 817
  finding in a string 673, 831
  ignoring case 812, 816
  initializing 830
  length of 825
  locating 838
  multibyte
    conversion using mbrtowc() 484
    conversion using mbstowcs() 489
    conversion using mbtowc() 490
    conversion with mbsrtowcs() 487
    length 480, 482
  property 434
  property classification 968
  reading
    with fgetc() 235
    with getc() and getchar() 312
    with getwc() 384
    with getwchar() 385

strings *(continued)*
reading *(continued)*
without echo 352
searching for 407, 811, 831
searching for tokens 842
searching, strspn() library function 837
setting 497
testing 419
ungetting 907, 908
writing with
fputc() 272
fputs() 273
fputwc() 274
fputws() 276
putc() and putchar() 613
strings.h header file 44
strlen() library function 825
strncasecmp() library function 826
strncat() library function 827
strncmp() library function 828
strncpy() library function 830
stropts.h header file 44
strpbrk() library function 831
strptime() library function 832
strrchr() library function 836
strspn() library function 837
strstr() library function 838
strtocoll() library function 839
strtod() library function 840
strtok() library function 842
strtol() library function 843
strtoul() library function 845
strxfrm() library function 847
supplementary group ID 711
swab() library function 849
swapcontext library function 849
switching, AMODE 223
swprintf() library function 851
swscanf() library function 852
symbolic constants in errno.h 24
symlink() library function 854
syntax of format for fprintf() family 265
sys/__cpl.h header file 44
sys/__messag.h header file 45
sys/__ussos.h header file 48
sys/_getipc.h header file 44
sys/acl.h header file 44
sys/file.h header file 44
sys/ioctl.h header file 44
sys/ipc.h header file 45
sys/layout.h header file 45
sys/mman.h header file 45
sys/mntent.h header file 45
sys/modes.h header file 45
sys/msg.h header file 45
sys/ps.h header file 45
sys/resource.h header file 45
sys/sem.h header file 45
sys/server.h header file 45
sys/shm.h header file 46
sys/socket.h header file 46
sys/stat.h header file 46
sys/statfs.h header file 46
sys/statvfs.h header file 46
sys/time.h header file 46
sys/timeb.h header file 46
sys/times.h header file 46
sys/ttydev.h header file 46
sys/types.h header file 46
sys/uio.h header file 48
sys/un.h header file 48
sys/utsname.h header file 48
sys/wait.h header file 48
sys/wlm.h header file 48
sysconf() library function 856
syslog.h header file 44
system configuration options 856
system() library function
calls across mixed environments 859
calls, general discussion 858
programming environment 39

T

takesocket() library function 860
tan() library function 861
tangent
calculating 861
hyperbolic, calculating 862
tanh() library function 862
tar.h header file 49
tcdrain() library function 863
tcflow() library function 864
tcflush() library function 867
tcgetattr() library function 869
tcgetpgrp() library function 870
tcperror() library function 872
tcsendbreak() library function 873
tcsetattr() library function 874
tcsetpgrp() library function 880
tdelete() library function 882
tempnam() library function 883
temporary files
names 41, 883, 889
number of 41
terminals
attributes 869
break condition 873
control modes 876
display,I/O flush 867
I/O
flush 867
input modes 874
local modes 877
output modes 876
suspending/resuming data flow 864
terminat.h header file 49
termination
abort() library function 66
atexit() library function 85
process or program 66
program 197
terminat.h 49
termios structure 874
termios.h header file 49
testing
for 7-bit US-ASCII character
421

testing *(continued)*
for NaN 431
testing characters 419
text
files 255
tfind() library function 884
threads
adding a signal 761
asynchronous signal, waiting for an 784
attribute object
destroying the definition 544
detachstate, getting the current value 545
detachstate, setting the current value 551
initializing a 549
stacksize, getting the 547
stacksize, setting the 552
weight, getting the current 548
weight, setting the current 554
broadcasting a condition 560
caller's thread ID, getting the 603
cancelability point, establishing a 610
cancelability states, setting the calling thread's 604
cancelability types, setting the calling thread's 606
cancelling 555
changing signal mask 773
comparing thread IDs 576
condition variable attribute object, destroying 568
condition variable attribute object, getting 569
condition variable attribute object, initializing 571
condition variable attribute object, setting 572
creating 573
creating key identifier 584
deleting a mutex object 588
destroying a condition variable 561
destroying a mutex attribute object 595
destroying the thread attributes object 544
detaching 575
detachstate, getting the 545
detachstate, setting the 551
establishing cleanup handler 558
exiting 577
ID, getting the caller's 603
initializing a condition variable 562
initializing a mutex 589
initializing a mutex attribute object 598
initializing a thread attributes object 549
invoking a function once 601
kind attribute, getting from mutex attribute object 596
kind attribute, setting from a mutex attribute object 599
lock, attempt to a mutex object 592
lock, wait for on a mutex object 590
pthread.h header file 36
releasing processor to other threads 611
removing cleanup handler 557
sending a signal 586
signalling a condition 563
specific value for a key, getting the 579
specific value for a key, setting the 608
stacksize, getting the 547
stacksize, setting the 552
unlocking a mutex object 593
waiting for thread to end 581, 582
waiting on a condition variable 566
waiting on a condition variable for a limited time 564
threads *(continued)*
weight, getting the current 548
weight, setting the current 554
time
considerations 125
conversion 832
converting from long integer to string 153
converting time structure to string 78
correcting for local time 456
ctime header file 22
file access/modification 912
formatting 821, 942
zone, testing 900
time_t type 166
time.h header file 46, 49
time() library function 456, 885
times.h header file 46
times() library function 886
tm structure 391
TMP_MAX macro 41, 889
tmpfile() library function 888
tmpnam() library function
file name specs in stdio.h file 41
toascii() library function 890
tokens 842
tolower() library function 895
toupper() library function 895
towlower() library function 896
towupper() library function 896
traceback 155
transforming strings 847
trigonometric functions
hyperbolic sine 786
hyperbolic tangent 862
sine 785
tangent 861
tsearch() library function 896
ttydev.h header file 46
ttyname() library function 897
ttyslot() library function 898
twalk() library function 899
typedef
definitions in stddef.h 40, 45
typeinfo header file 50
typeinfo.h header file 51
types.h header file 46
TZ environment variable 900
tzset() library function 900

U

ucontext.h header file 51
UDP (user datagram protocol) 664
uheap.h header file 51
uio.h header file 48
ulimit.h header file 51
umask() library function 902
umount() library function 903
un.h header file 48
uname() library function 905
unblocking a thread 560, 563
underscore character mapping to __ 61
unexpect.h header file 51
ungetc() library function 907

ungetwc() library function 908
unistd.h header file 51
UNIX System Services
    debugging, OpenExtensions reason codes 190
    fcntl.h header file 25
    invoking POSIX programs 7
    overview 6
unlink() library function 910
unlocking a mutex object 593
unsigned short integer 396
user context
    modifying 475
user database 361, 362
user datagram protocol (UDP) 664
user ID
    effective 327
    real 378
    setting 710
usleep() library function 911
utime.h header file 52
utime() library function 912
utimes() library function 914
utmpx.h header file 52
utsname.h header file 48

V

va_arg() macro 916
va_end() macro 916
va_start() macro 916
valloc() library function 919
varargs.h header file 52
variables
    configurable path name 533
    configuration 261
    getting configurable 137
variant structure 376
variant.h header file 52
vfork() library function 919
vfprintf() library function 922
vprintf() library function 923
VSAM (Virtual Storage Access Method)
    I/O operations
        updating a record 307
vsprintf() library function 924
vswprintf() library function 925

W

w_getpsent() library function 970
w_statvfs() library function 982
wait
    asynchronous signal 784
    child process 927, 929, 930
    condition variable 566
    condition variable for a limited time 564
    thread to end 582
wait.h header file 48
wait() library function 927
waitid() library function 929
waitpid() library function 930
wchar.h header file 53
wcrtomb() library function 933
wcscat() library function 935
wcschr() library function 936
wcscmp() library function 937
wcscoll() library function 938
wcscpy() library function 940
wcscspn() library function 941
wcsftime() library function 942
wcsid() library function 943
wcslen() library function 944
wcsncat() library function 945
wcsncmp() library function 946
wcsncpy() library function 948
wcspbrk() library function 949
wcsrchr() library function 950
wcsrtombs() library function 951
wcsspn() library function 953
wcsstr() library function 954
wcstod() library function 955
wcstok() library function 957
wcstol() library function 959
wcstombs() library function 960
wcstoul() library function 962
wcstr.h header file 54
wcswcs() library function 963
wcswidth() library function 964
wcsxfrm() library function 965
wctob() library function 966
wctomb() library function 967
wctype.h header file 54
wctype() library function 968
wcwidth() library function 969
weight
    getting the current thread 548
    setting the current thread 554
WEOF macro 54
wide characters
    appending to strings 945
    breaking string into tokens 957
    character set ID 943
    comparing strings 946
    converting
        string to double floating point 955
        string to long integer 959
        string to multibyte 960
        string to unsigned long integer 962
        to byte 966
        to multibyte 933, 967
        to multibyte string 951
    copying strings with wcscpy() 940
    copying strings with wcsncpy() 948
    displaying width 964, 969
    I/O functions 384, 385
    locating in a string 949, 950
    locating sequence 954
    locating substring 963
    matching offset 941
    reading streams and files 241
    searching for 953
    string comparison 937
    string conversion 276
    string length 944
    string, reading 852
    substring 936
    transforming string 965

wide characters *(continued)*
    writing 925
    writing to wide-character array 851
wide integer 432
wordexp.h header file 54
wordexp() library function 972
wordfree() library function 975
working directory
    changing 116
    path name 320
wrapping of output 615
writable static
    shared 231
write
    data on sockets 979
    entry to utmpx database 616
    lock 211
    operations
        character to stdout 272, 274, 613
        character to stream 272, 274, 613, 907, 908
        formatted 264
        line to stream 615, 619, 620
        string to stream 273, 276
    word to stream 618
write() library function 975
writev() library function 979

X

xhotc() 984
xhotl() 984
xhott() 984
xregs() 984
xsacc() 984
xsrvc() 984
xti.h header file 54
xusr() 984
xusr2() 984

Y

y0() library function 983
y1() library function 983
yield (release processor to other threads) 611
yn() library function 983

Product Number: 5654-A22

Printed in USA

SC09-7624-74