



System i  
ILE C/C++ Runtime Library Functions

*Version 6 Release 1*

SC41-5607-03







System i  
ILE C/C++ Runtime Library Functions

*Version 6 Release 1*

SC41-5607-03

**Note**

Before using this information and the product it supports, be sure to read the information in Appendix B, "Notices," on page 555.

This edition applies to version 6, release 1, modification 0 of IBM i5/OS (product number 5761-SS1), and to all subsequent releases and modifications until otherwise indicated in new editions. This version does not run on all reduced instruction set computer (RISC) models nor does it run on CISC models.

© Copyright International Business Machines Corporation 1999, 2008. All rights reserved.

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

# Contents

<b>Tables</b> . . . . .	<b>ix</b>
-------------------------	-----------

## About ILE C/C++ Runtime Library Functions (SC41-5607) . . . . . **xi**

Who should read this book . . . . .	xi
A note about examples. . . . .	xi
Prerequisite and related information . . . . .	xi
How to send your comments . . . . .	xii

## Summary of Changes . . . . . **xiii**

## Part 1. Runtime Library Functions . . . . . **1**

### Chapter 1. Include Files . . . . . **3**

<assert.h> . . . . .	3
<ctype.h> . . . . .	3
<decimal.h> . . . . .	3
<errno.h> . . . . .	4
<except.h> . . . . .	4
<float.h>. . . . .	7
<inttypes.h> . . . . .	7
<langinfo.h> . . . . .	7
<limits.h> . . . . .	7
<locale.h> . . . . .	7
<math.h> . . . . .	8
<mallocinfo.h>. . . . .	8
<monetary.h> . . . . .	8
<nl_types.h> . . . . .	9
<pointer.h> . . . . .	9
<recio.h> . . . . .	9
<regex.h> . . . . .	12
<setjmp.h> . . . . .	13
<signal.h> . . . . .	13
<stdarg.h>. . . . .	13
<stddef.h>. . . . .	13
<stdint.h> . . . . .	14
<stdio.h> . . . . .	15
<stdlib.h> . . . . .	17
<string.h> . . . . .	17
<strings.h> . . . . .	17
<time.h> . . . . .	18
<wchar.h>. . . . .	18
<wctype.h> . . . . .	19
<xxcvt.h> . . . . .	19
<xxdtaa.h> . . . . .	19
<xxenv.h> . . . . .	19
<xxfdbk.h> . . . . .	19
Machine Interface (MI) Include Files . . . . .	20

### Chapter 2. Library Functions . . . . . **21**

The C/C++ Library. . . . .	21
Error Handling . . . . .	21
Searching and Sorting . . . . .	22
Mathematical. . . . .	22

Time Manipulation . . . . .	24
Type Conversion . . . . .	25
Conversion . . . . .	26
Record Input/Output . . . . .	26
Stream Input/Output . . . . .	27
Handling Argument Lists. . . . .	30
Pseudorandom Numbers . . . . .	31
Dynamic Memory Management . . . . .	31
Memory Objects. . . . .	31
Environment Interaction . . . . .	32
String Operations . . . . .	32
Character Testing . . . . .	33
Multibyte Character Testing . . . . .	34
Character Case Mapping . . . . .	34
Multibyte Character Manipulation. . . . .	34
Data Areas . . . . .	36
Message Catalogs . . . . .	36
Regular Expression . . . . .	36
abort() — Stop a Program . . . . .	36
abs() — Calculate Integer Absolute Value . . . . .	37
acos() — Calculate Arccosine . . . . .	38
asctime() — Convert Time to Character String . . . . .	39
asctime_r() — Convert Time to Character String (Restartable) . . . . .	41
asin() — Calculate Arcsine . . . . .	42
assert() — Verify Condition . . . . .	43
atan() - atan2() — Calculate Arctangent . . . . .	44
atexit() — Record Program Ending Function . . . . .	45
atof() — Convert Character String to Float . . . . .	46
atoi() — Convert Character String to Integer . . . . .	48
atol() — atoll() — Convert Character String to Long or Long Long Integer . . . . .	49
Bessel Functions. . . . .	50
bsearch() — Search Arrays . . . . .	51
btowc() — Convert Single Byte to Wide Character . . . . .	53
_C_Get_Ssn_Handle() — Handle to C Session . . . . .	55
calloc() — Reserve and Initialize Storage. . . . .	55
catclose() — Close Message Catalog . . . . .	57
catgets() — Retrieve a Message from a Message Catalog. . . . .	58
catopen() — Open Message Catalog . . . . .	59
ceil() — Find Integer >=Argument. . . . .	61
clearerr() — Reset Error Indicators. . . . .	62
clock() — Determine Processor Time . . . . .	63
cos() — Calculate Cosine . . . . .	64
cosh() — Calculate Hyperbolic Cosine . . . . .	65
_C_Quickpool_Debug() — Modify Quick Pool Memory Management Characteristics. . . . .	66
_C_Quickpool_Init() — Initialize Quick Pool Memory Management. . . . .	68
_C_Quickpool_Report() — Generate Quick Pool Memory Management Report . . . . .	70
ctime() — Convert Time to Character String . . . . .	71
ctime64() — Convert Time to Character String. . . . .	73
ctime_r() — Convert Time to Character String (Restartable) . . . . .	74

ctime64_r() — Convert Time to Character String (Restartable) . . . . .	76	hypot() — Calculate Hypotenuse . . . . .	168
_C_TS_malloc_debug() — Determine amount of teraspace memory used (with optional dumps and verification) . . . . .	77	isalnum() - isxdigit() — Test Integer Value . . . . .	169
_C_TS_malloc_info() — Determine amount of teraspace memory used . . . . .	79	isascii() — Test for Character Representable as ASCII Value . . . . .	171
difftime() — Compute Time Difference . . . . .	82	isblank() — Test for Blank or Tab Character . . . . .	172
difftime64() — Compute Time Difference . . . . .	84	iswalnum() to iswxdigit() — Test Wide Integer Value . . . . .	173
div() — Calculate Quotient and Remainder . . . . .	86	iswctype() — Test for Character Property . . . . .	175
erf() - erfc() — Calculate Error Functions . . . . .	87	_itoa - Convert Integer to String . . . . .	176
exit() — End Program . . . . .	88	labs() — llabs() — Calculate Absolute Value of Long and Long Long Integer . . . . .	177
exp() — Calculate Exponential Function . . . . .	89	ldexp() — Multiply by a Power of Two . . . . .	178
fabs() — Calculate Floating-Point Absolute Value . . . . .	90	ldiv() — lldiv() — Perform Long and Long Long Division . . . . .	179
fclose() — Close Stream . . . . .	91	localeconv() — Retrieve Information from the Environment . . . . .	181
fdopen() — Associates Stream With File Descriptor . . . . .	92	localtime() — Convert Time . . . . .	185
feof() — Test End-of-File Indicator . . . . .	95	localtime64() — Convert Time . . . . .	187
ferror() — Test for Read/Write Errors . . . . .	95	localtime_r() — Convert Time (Restartable) . . . . .	188
fflush() — Write Buffer to File . . . . .	96	localtime64_r() — Convert Time (Restartable) . . . . .	189
fgetc() — Read a Character . . . . .	98	log() — Calculate Natural Logarithm . . . . .	191
fgetpos() — Get File Position . . . . .	99	log10() — Calculate Base 10 Logarithm . . . . .	191
fgets() — Read a String . . . . .	101	_ltoa - Convert Long Integer to String . . . . .	192
fgetwc() — Read Wide Character from Stream . . . . .	102	longjmp() — Restore Stack Environment . . . . .	193
fgetws() — Read Wide-Character String from Stream . . . . .	104	malloc() — Reserve Storage Block . . . . .	195
fileno() — Determine File Handle . . . . .	106	mblen() — Determine Length of a Multibyte Character . . . . .	197
floor() — Find Integer <=Argument . . . . .	107	mbrlen() — Determine Length of a Multibyte Character (Restartable) . . . . .	199
fmod() — Calculate Floating-Point Remainder . . . . .	108	mbrtowc() — Convert a Multibyte Character to a Wide Character (Restartable) . . . . .	201
fopen() — Open Files . . . . .	109	mbsinit() — Test State Object for Initial State . . . . .	205
fprintf() — Write Formatted Data to a Stream . . . . .	116	mbsrtowcs() — Convert a Multibyte String to a Wide Character String (Restartable) . . . . .	206
fputc() — Write Character . . . . .	118	mbstowcs() — Convert a Multibyte String to a Wide Character String . . . . .	207
_fputchar - Write Character . . . . .	120	mbtowc() — Convert Multibyte Character to a Wide Character . . . . .	211
fputs() — Write String . . . . .	121	memchr() — Search Buffer . . . . .	212
fputwc() — Write Wide Character . . . . .	122	memcmp() — Compare Buffers . . . . .	213
fputws() — Write Wide-Character String . . . . .	124	memcpy() — Copy Bytes . . . . .	214
fread() — Read Items . . . . .	126	memcmp() - Compare Bytes . . . . .	215
free() — Release Storage Blocks . . . . .	128	memmove() — Copy Bytes . . . . .	217
freopen() — Redirect Open Files . . . . .	130	memset() — Set Bytes to Value . . . . .	218
frexp() — Separate Floating-Point Value . . . . .	132	mktime() — Convert Local Time . . . . .	218
fscanf() — Read Formatted Data . . . . .	132	mktime64() — Convert Local Time . . . . .	220
fseek() — fseeko() — Reposition File Position . . . . .	134	modf() — Separate Floating-Point Value . . . . .	222
fsetpos() — Set File Position . . . . .	136	nextafter() — nextafterl()— nexttoward() —	
ftell() — ftello() — Get Current Position . . . . .	138	nexttowardl() — Calculate the Next Representable	
fwide() — Determine Stream Orientation . . . . .	140	Floating-Point Value . . . . .	223
fwprintf() — Format Data as Wide Characters and Write to a Stream . . . . .	143	nl_langinfo() —Retrieve Locale Information . . . . .	224
fwrite() — Write Items . . . . .	146	perror() — Print Error Message . . . . .	227
fwscanf() — Read Data from Stream Using Wide Character . . . . .	147	pow() — Compute Power . . . . .	228
gamma() — Gamma Function . . . . .	150	printf() — Print Formatted Characters . . . . .	229
_gcvt - Convert Floating-Point to String . . . . .	151	putc() - putchar() — Write a Character . . . . .	239
getc() - getchar() — Read a Character . . . . .	152	putenv() — Change/Add Environment Variables . . . . .	240
getenv() — Search for Environment Variables . . . . .	154	puts() — Write a String . . . . .	241
_GetExData() — Get Exception Data . . . . .	154	putwc() — Write Wide Character . . . . .	242
gets() — Read a Line . . . . .	156	putwchar() — Write Wide Character to stdout . . . . .	244
getwc() — Read Wide Character from Stream . . . . .	157	qsort() — Sort Array . . . . .	245
getwchar() — Get Wide Character from stdin . . . . .	159	QXXCHGDA() — Change Data Area . . . . .	247
gmtime() — Convert Time . . . . .	161		
gmtime64() — Convert Time . . . . .	163		
gmtime_r() — Convert Time (Restartable) . . . . .	165		
gmtime64_r() — Convert Time (Restartable) . . . . .	167		

QXXDTOP() — Convert Double to Packed Decimal	248	setvbuf() — Control Buffering	344
QXXDTOZ() — Convert Double to Zoned Decimal	249	signal() — Handle Interrupt Signals	346
QXXITOP() — Convert Integer to Packed Decimal	250	sin() — Calculate Sine	348
QXXITOZ() — Convert Integer to Zoned Decimal	250	sinh() — Calculate Hyperbolic Sine	349
QXXPTOD() — Convert Packed Decimal to Double	251	snprintf() — Print Formatted Data to Buffer	350
QXXPTOI() — Convert Packed Decimal to Integer	252	sprintf() — Print Formatted Data to Buffer	352
QXXRTVDA() — Retrieve Data Area	252	sqrt() — Calculate Square Root	353
QXXZTOD() — Convert Zoned Decimal to Double	254	srand() — Set Seed for rand() Function	354
QXXZTOI() — Convert Zoned Decimal to Integer	255	sscanf() — Read Data	355
raise() — Send Signal	255	strcasemp() — Compare Strings without Case Sensitivity	357
rand(), rand_r() — Generate Random Number	256	strcat() — Concatenate Strings	358
_Racquire() — Acquire a Program Device	257	strchr() — Search for Character	359
_Rclose() — Close a File	258	strcmp() — Compare Strings	360
_Rcommit() — Commit Current Record	259	strcmpi() - Compare Strings Without Case Sensitivity	362
_Rdelete() — Delete a Record	261	strcoll() — Compare Strings	363
_Rdevatr() — Get Device Attributes	263	strcpy() — Copy Strings	364
realloc() — Change Reserved Storage Block Size	264	strcspn() — Find Offset of First Character Match	365
regcomp() — Compile Regular Expression	267	strdup - Duplicate String	366
regerror() — Return Error Message for Regular Expression	269	strerror() — Set Pointer to Runtime Error Message	367
regexec() — Execute Compiled Regular Expression	271	strfmon() — Convert Monetary Value to String	368
regfree() — Free Memory for Regular Expression	273	strftime() — Convert Date/Time to String	370
remove() — Delete File	274	stricmp() - Compare Strings without Case Sensitivity	374
rename() — Rename File	275	strlen() — Determine String Length	375
rewind() — Adjust Current File Position	276	strncasemp() — Compare Strings without Case Sensitivity	376
_Rfeed() — Force the End-of-Data	278	strncat() — Concatenate Strings	377
_Rfeov() — Force the End-of-File	279	strncmp() — Compare Strings	379
_Rformat() — Set the Record Format Name	280	strncpy() — Copy Strings	380
_Rindara() — Set Separate Indicator Area	282	strnicmp - Compare Substrings Without Case Sensitivity	382
_Riofbk() — Obtain I/O Feedback Information	284	strnset - strset - Set Characters in String	383
_Rlocate() — Position a Record	286	strpbrk() — Find Characters in String	384
_Ropen() — Open a Record File for I/O Operations	289	strptime() — Convert String to Date/Time	385
_Ropnfbk() — Obtain Open Feedback Information	293	strrchr() — Locate Last Occurrence of Character in String	389
_Rpgmdev() — Set Default Program Device	294	strspn() — Find Offset of First Non-matching Character	390
_Rread() — Read a Record by Relative Record Number	295	strstr() — Locate Substring	391
_Rreadf() — Read the First Record	297	strtod() — strtod() — strtold — Convert Character String to Double, Float, and Long Double	392
_Rreadindv() — Read from an Invited Device	299	strtod32() — strtod64() — strtod128() — Convert Character String to Decimal Floating-Point	395
_Rreadk() — Read a Record by Key	302	strtok() — Tokenize String	398
_Rreadl() — Read the Last Record	305	strtok_r() — Tokenize String (Restartable)	399
_Rreadn() — Read the Next Record	306	strtol() — strtoll() — Convert Character String to Long and Long Long Integer	400
_Rreadnc() — Read the Next Changed Record in a Subfile	308	strtoul() — strtoull() — Convert Character String to Unsigned Long and Unsigned Long Long Integer	403
_Rreadp() — Read the Previous Record	310	strxfrm() — Transform String	404
_Rreads() — Read the Same Record	312	swprintf() — Format and Write Wide Characters to Buffer	406
_Rrelease() — Release a Program Device	314	swscanf() — Read Wide Character Data	407
_Rrlslck() — Release a Record Lock	316	system() — Execute a Command	408
_Rrollbck() — Roll Back Commitment Control Changes	317	tan() — Calculate Tangent	409
_Rupdate() — Update a Record	319	tanh() — Calculate Hyperbolic Tangent	410
_Rupfb() — Provide Information on Last I/O Operation	320	time() — Determine Current Time	411
_Rwrite() — Write the Next Record	322	time64() — Determine Current Time	412
_Rwrited() — Write a Record Directly	324	tmpfile() — Create Temporary File	414
_Rwriterd() — Write and Read a Record	327		
_Rwrread() — Write and Read a Record (separate buffers)	328		
scanf() — Read Data	330		
setbuf() — Control Buffering	336		
setjmp() — Preserve Environment	338		
setlocale() — Set Locale	339		

tmpnam() — Produce Temporary File Name . . . . .	414	wcstod() — Convert Wide-Character String to Double . . . . .	476
tolower() — Convert Character to Character Representable by ASCII . . . . .	415	wcstod32() — wcstod64() — wcstod128()— Convert Wide-Character String to Decimal Floating-Point . . . . .	478
toupper() — Convert Character Case . . . . .	416	wctok() — Tokenize Wide-Character String . . . . .	480
towctrans() — Translate Wide Character . . . . .	417	wcstol() — wcstoll() — Convert Wide Character String to Long and Long Long Integer . . . . .	481
towlower() —towupper() — Convert Wide Character Case . . . . .	418	wcstombs() — Convert Wide-Character String to Multibyte String . . . . .	483
_ultoa - Convert Unsigned Long Integer to String . . . . .	419	wcstoul() — wcstoull() — Convert Wide Character String to Unsigned Long and Unsigned Long Long Integer . . . . .	486
ungetc() — Push Character onto Input Stream . . . . .	420	wcswcs() — Locate Wide-Character Substring . . . . .	488
ungetwc() — Push Wide Character onto Input Stream . . . . .	422	wcswidth() — Determine the Display Width of a Wide Character String . . . . .	489
va_arg() - va_end() - va_start() — Access Function Arguments . . . . .	423	wcsxfrm() — Transform a Wide-Character String . . . . .	490
vfprintf() — Print Argument Data to Stream . . . . .	425	wctob() — Convert Wide Character to Byte . . . . .	491
vfscanf() — Read Formatted Data . . . . .	427	wctomb() — Convert Wide Character to Multibyte Character. . . . .	492
vfwprintf() — Format Argument Data as Wide Characters and Write to a Stream. . . . .	428	wctrans() —Get Handle for Character Mapping . . . . .	493
vfwscanf() — Read Formatted Wide Character Data . . . . .	430	wctype() — Get Handle for Character Property Classification . . . . .	495
vprintf() — Print Argument Data . . . . .	432	wcwidth() — Determine the Display Width of a Wide Character. . . . .	497
vscanf() — Read Formatted Data . . . . .	433	wfopen() —Open Files . . . . .	498
vsprintf() — Print Argument Data to Buffer . . . . .	435	wmemchr() —Locate Wide Character in Wide-Character Buffer . . . . .	498
vsprintf() — Print Argument Data to Buffer . . . . .	436	wmemcmp() —Compare Wide-Character Buffers . . . . .	499
vsscanf() — Read Formatted Data . . . . .	437	wmemcpy() —Copy Wide-Character Buffer . . . . .	500
vsprintf() — Format and Write Wide Characters to Buffer . . . . .	439	wmemmove() — Copy Wide-Character Buffer . . . . .	501
vswscanf() — Read Formatted Wide Character Data . . . . .	441	wmemset() — Set Wide Character Buffer to a Value . . . . .	502
vwprintf() — Format Argument Data as Wide Characters and Print . . . . .	443	wprintf() — Format Data as Wide Characters and Print . . . . .	503
vwsscanf() — Read Formatted Wide Character Data . . . . .	445	wscanf() — Read Data Using Wide-Character Format String . . . . .	504
wcrtomb() — Convert a Wide Character to a Multibyte Character (Restartable). . . . .	446		
wcscat() — Concatenate Wide-Character Strings . . . . .	451		
wcschr() — Search for Wide Character . . . . .	452		
wcscmp() — Compare Wide-Character Strings . . . . .	453		
wscoll() —Language Collation String Comparison . . . . .	455		
wscncpy() — Copy Wide-Character Strings . . . . .	456		
wcscspn() — Find Offset of First Wide-Character Match . . . . .	457		
wcsftime() — Convert to Formatted Date and Time . . . . .	458		
__wcsicmp() — Compare Wide Character Strings without Case Sensitivity. . . . .	460		
wcslen() — Calculate Length of Wide-Character String . . . . .	461		
wcslocaleconv() — Retrieve Wide Locale Information . . . . .	462		
wcsncat() — Concatenate Wide-Character Strings . . . . .	463		
wcsncmp() — Compare Wide-Character Strings . . . . .	464		
wcsncpy() — Copy Wide-Character Strings . . . . .	466		
__wcsnicmp() — Compare Wide Character Strings without Case Sensitivity. . . . .	467		
wcsprbrk() — Locate Wide Characters in String . . . . .	468		
wcsptime()— Convert Wide Character String to Date/Time . . . . .	469		
wcsrchr() — Locate Last Occurrence of Wide Character in String . . . . .	471		
wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable) . . . . .	473		
wcsspn() — Find Offset of First Non-matching Wide Character. . . . .	474		
wcsstr() — Locate Wide-Character Substring . . . . .	475		
		<b>Chapter 3. Runtime Considerations</b>	<b>507</b>
		errno Macros . . . . .	507
		errno Values for Integrated File System Enabled C Stream I/O . . . . .	508
		Record Input and Output Error Macro to Exception Mapping . . . . .	510
		Signal Handling Action Definitions . . . . .	511
		Signal to i5/OS Exception Mapping . . . . .	513
		Cancel Handler Reason Codes. . . . .	514
		Exception Classes . . . . .	515
		Data Type Compatibility . . . . .	516
		Runtime Character Set . . . . .	523
		Understanding CCSIDs and Locales . . . . .	524
		CCSIDs of Characters and Character Strings . . . . .	524
		Wide Characters . . . . .	527
		Asynchronous Signal Model . . . . .	529
		Unicode Support . . . . .	530
		Reasons to Use Unicode Support . . . . .	531
		Pseudo-CCSID Neutrality . . . . .	531
		Unicode from Other ILE Languages . . . . .	532
		Standard Files . . . . .	534
		Considerations . . . . .	534
		Default File CCSID . . . . .	535
		Newline Character . . . . .	536
		Conversion Errors . . . . .	536

<b>Appendix A. Library Functions and Extensions</b>	<b>537</b>
Standard C Library Functions Table, By Name	537
ILE C Library Extensions to C Library Functions Table	551
<b>Appendix B. Notices</b>	<b>555</b>

Programming interface information	556
Trademarks	557
<b>Bibliography</b>	<b>559</b>
<b>Index</b>	<b>561</b>



---

## Tables

1. Grouping Example . . . . .	183	20. Exception Classes . . . . .	515
2. Monetary Formatting Example . . . . .	183	21. ILE C Data Type Compatibility with ILE RPG	517
3. Monetary Fields . . . . .	183	22. ILE C Data Type Compatibility with ILE	
4. Values of Precision . . . . .	235	COBOL . . . . .	518
5. . . . .	288	23. ILE C Data Type Compatibility with ILE CL	519
6. Return values of <code>strcasemp()</code> . . . . .	357	24. ILE C Data Type Compatibility with OPM	
7. Flags . . . . .	369	RPG/400 . . . . .	519
8. Conversion Characters . . . . .	370	25. ILE C Data Type Compatibility with OPM	
9. Return values of <code>strncasemp()</code> . . . . .	376	COBOL/400 . . . . .	520
10. Return values of <code>__wcsicmp()</code> . . . . .	460	26. ILE C Data Type Compatibility with CL	521
11. Return values of <code>__wcsicmp()</code> . . . . .	468	27. Arguments Passed From a Command Line	
12. <code>errno</code> Macros . . . . .	507	CL Call to an ILE C Program . . . . .	522
13. <code>errno</code> Values for Integrated File System		28. CL Constants Passed from a Compiled CL	
Enabled C Stream I/O . . . . .	508	Program to an ILE C Program . . . . .	522
14. Record Input and Output Error Macro to		29. CL Variables Passed from a Compiled CL	
Exception Mapping . . . . .	510	Program to an ILE C Program . . . . .	522
15. Handling Action Definitions for Signal Values	511	30. Invariant Characters . . . . .	523
16. Default Actions for Signal Values . . . . .	512	31. Variant Characters in Different CCSIDs	523
17. Signal to i5/OS Exception Mapping . . . . .	513	32. Standard C Library Functions . . . . .	537
18. Determining Canceled Invocation Reason		33. ILE C Library Extensions . . . . .	551
Codes . . . . .	514		
19. Common Reason Code for Cancelling			
Invocations . . . . .	515		



---

## About ILE C/C++ Runtime Library Functions (SC41-5607)

This book provides reference information about:

- Include files
- | • Runtime functions
- | • Runtime considerations

Use this book as a reference when you write Integrated Language Environment<sup>®</sup> (ILE) C and C++ applications.

This book does not describe how to program in the C or C++ programming languages, nor does it explain the concepts of ILE. Companion publications for this reference are:

- *C/C++ Legacy Class Libraries Reference*, SC09-7652-00
- *ILE Concepts*, SC41-5606-08
- *ILE C/C++ for AS/400 MI Library Reference*, SC09-2418-00
- *Standard C/C++ Library Reference*, SC09-4949-01
- *WebSphere Development Studio: ILE C/C++ Compiler Reference*, SC09-4816-04
- *WebSphere Development Studio: ILE C/C++ Language Reference*, SC09-7852-01
- *WebSphere Development Studio: ILE C/C++ Programmer's Guide*, SC09-2712-06

For other prerequisite and related information, see “Prerequisite and related information” and the “Bibliography” on page 559.

---

### Who should read this book

This book is intended for programmers who are familiar with the C/C++ programming language and who want to write or maintain ILE C/C++ applications. You must have experience in using applicable i5/OS<sup>®</sup> menus, and displays or control language (CL) commands. You also need knowledge of Integrated Language Environment as explained in the *ILE Concepts* manual.

### A note about examples

- | The examples in this book that illustrate the use of library functions are written in a simple style. The
- | examples do not demonstrate all possible uses of C/C++ language constructs. Some examples are only
- | code fragments and do not compile without additional code. The examples all assume that the C locale is
- | used.

All complete runnable examples for library functions and machine interface instructions are in library QCPPL, in source file QACSRC. Each example name is the same as the function name or instruction name. For example, the source code for the example illustrating the use of the `_Rcommit()` function in this book is in library QCPPL, file QACSRC, member RCOMMIT. The QSYSINC library must be installed.

---

### Prerequisite and related information

Use the IBM<sup>®</sup> i5/OS Information Center as your starting point for IBM System i<sup>™</sup> technical information.

You can access the information center two ways:

- From the following Web site:  
<http://www.ibm.com/systems/i/infocenter/>

- From the *i5/OS Information Center CD*, SK3T-4091. This CD-ROM is included with your new System i hardware or i5/OS software upgrade order. You can also order the CD-ROM from the IBM Publications Center:

<http://www.ibm.com/shop/publications/order>

The i5/OS Information Center contains new and updated system information such as software and hardware installation, Linux<sup>®</sup>, WebSphere<sup>®</sup>, Java<sup>™</sup>, high availability, database, logical partitions, CL commands, and system application programming interfaces (APIs). In addition, it provides advisors and finders to assist in planning, troubleshooting, and configuring your system hardware and software.

With every new hardware order, you receive the *System i Access for Windows DVD*, SK3T-4098. This DVD provides for the installation for IBM System i Access for Windows licensed program. System i Access Family offers client and server capabilities for connecting PCs to System i models.

For other related information, see the “Bibliography” on page 559.

---

## How to send your comments

Your feedback is important in helping to provide the most accurate and high-quality information. If you have any comments about this book or any other System i documentation, fill out the readers’ comment form at the back of this book.

- If you prefer to send comments by mail, use the readers’ comment form with the address that is printed on the back. If you are mailing a readers’ comment form from a country or region other than the United States, you can give the form to the local IBM branch office or IBM representative for postage-paid mailing.
- If you prefer to send comments by FAX, use either of the following numbers:
  - United States, Canada, and Puerto Rico: 1-800-937-3430
  - Other countries or regions: 1-507-253-5192
- If you prefer to send comments electronically, use one of these e-mail addresses:
  - Comments on books:  
RCHCLERK@us.ibm.com
  - Comments on the i5/OS Information Center:  
RCHINFOC@us.ibm.com

Be sure to include the following:

- The name of the book or i5/OS Information Center topic.
- The publication number of a book.
- The page number or topic of a book to which your comment applies.

---

## Summary of Changes

Here are the changes to this information for this edition.

- V6R1 updates

- ILE C runtime now supports the new data type decimal floating-point:

- strtod32()–strtod64()–strtod128()

- wcstod32()–wcstod64()–wcstod128()

- <float.h>

- printf()

- scanf()

- 64-bit time functions have been added to ILE C runtime:

- ctime64()

- ctime64\_r()

- difftime64()

- gmtime64()

- gmtime64\_r()

- localtime64()

- localtime64\_r()

- mktime64()

- time64()

- V5R4 updates

The topic was not updated in V5R4. The following sections have been updated or added in this release for the V5R4 updates since the previous edition.

- \_C\_Quickpool\_Debug()

- \_C\_Quickpool\_Init()

- \_C\_Quickpool\_Report()

- <inttypes.h>

- nextafter(), nextafterl(), nexttoward(), nexttowardl()

- <stdint.h>

- strtod(), strtof(), strtold()

- Understanding CCSIDs and Locales

- Unicode from other ILE languages



---

## Part 1. Runtime Library Functions



---

## Chapter 1. Include Files

The include files that are provided with the runtime library contain macro and constant definitions, type definitions, and function declarations. Some functions require definitions and declarations from include files to work properly. The inclusion of files is optional, as long as the necessary statements from the files are coded directly into the source.

This section describes each include file, explains its contents, and lists the functions that are declared in the file.

- | The QSYSINC (system openness includes) library must be installed on your i5/OS operating system.
- | QSYSINC contains include files useful for C/C++ users, such as system API, Dynamic Screen Manager (DSM), and ILE header files. The QSYSINC library contains header files that include the prototypes and templates for the machine interface (MI) built-ins and the ILE C/C++ MI functions. See the *ILE C/C++ for AS/400 MI Library Reference* for more information about these header files.

---

### <assert.h>

The <assert.h> include file defines the `assert` macro. You must include `assert.h` when you use `assert`.

The definition of `assert` is in an `#ifndef` preprocessor block. If you have not defined the identifier `NDEBUG` through a `#define` directive or on the compilation command, the `assert` macro tests the assertion expression. If the assertion is false, the system prints a message to `stderr`, and raises an abort signal for the program. The system also does a Dump Job (DMPJOB) OUTPUT(\*PRINT) when the assertion is false.

If `NDEBUG` is defined, `assert` is defined to do nothing. You can suppress program assertions by defining `NDEBUG`.

---

### <ctype.h>

The <ctype.h> include file defines functions that are used in character classification. The functions that are defined in <ctype.h> are:

<code>isascii</code> <sup>1</sup>	<code>isblank</code> <sup>2</sup>	<code>isgraph</code>	<code>ispunct</code>	<code>toascii</code> <sup>1</sup>
<code>isalnum</code>	<code>isctrl</code>	<code>islower</code>	<code>isspace</code>	<code>tolower</code>
<code>isalpha</code>	<code>isdigit</code>	<code>isprint</code>	<code>isupper</code>	<code>toupper</code>
			<code>isxdigit</code>	

---

**Note:** <sup>1</sup> These functions are not available when `LOCALETYPE(*CLD)` is specified on the compilation command.

**Note:** <sup>2</sup> This function is applicable to C++ only.

---

### <decimal.h>

The <decimal.h> include file contains definitions of constants that specify the ranges of the packed decimal type and its attributes. The <decimal.h> file must be included with a `#include` directive in your source code if you use the keywords `decimal`, `digitsof`, or `precisionof`.

---

## <errno.h>

The <errno.h> include file defines macros that are set to the errno variable. The <errno.h> include file defines macros for values that are used for error reporting in the C library functions and defines the macro errno. An integer value can be assigned to errno, and its value can be tested during run time. See "Checking the Errno Value" in the *WebSphere Development Studio: ILE C/C++ Programmer's Guide* for information about displaying the current errno value.

**Note:** To test the value of errno after library function calls, set it to 0 before the call because its value may not be reset during the call.

---

## <except.h>

The <except.h> include file declares types and macros that are used in ILE C exception handling.

The definition of \_INTRPT\_Hndlr\_Parms\_T is:

```
typedef _Packed struct {
    unsigned int    Block_Size;
    _INVFLAGS_T    Tgt_Flags;
    char            reserved[8];
    _INVPTR        Target;
    _INVPTR        Source;
    _SPCPTR        Com_Area;
    char            Compare_Data[32];
    char            Msg_Id[7];
    char            reserved1;
    _INTRPT_Mask_T Mask;
    unsigned int    Msg_Ref_Key;
    unsigned short  Exception_Id;
    unsigned short  Compare_Data_Len;
    char            Signal_Class;
    char            Priority;
    short           Severity;
    char            reserved3[4];
    int             Msg_Data_Len;
    char            Mch_Dep_Data[10];
    char            Tgt_Inv_Type;
    _SUSPENDPTR    Tgt_Suspend;
    char            Ex_Data[48];
} _INTRPT_Hndlr_Parms_T;
```

### Element

#### Description

#### Block\_Size

The size of the parameter block passed to the exception handler.

#### Tgt\_Flags

Contains flags that are used by the system.

#### reserved

An eight byte reserved field.

**Target** An invocation pointer to the call stack entry that enabled the exception handler.

#### Source

An invocation pointer to the call stack entry that caused the exception. If that call stack entry no longer exists, then this is a pointer to the call stack entry where control resumes when the exception is handled.

#### Com\_Area

A pointer to the communications area variable specified as the second parameter on the #pragma exception\_handler. If a communication area was not specified, this value is NULL.

**Compare\_Data**

The compare data consists of 4 bytes of message prefix, for example CPF, MCH, followed by 28 bytes which are taken from the message data of the related message. In the case where the message data is greater than 28 these are the first 28 bytes. For MCH messages, these are the first 28 bytes of the exception related data that is returned by the system (substitution text).

**Msg\_Id**

A message identifier, for example CPF123D. \*STATUS message types are not updated in this field.

**reserved1**

A 1 byte pad.

**Mask** This is an 8-byte exception mask, identifying the type of the exception that occurred, for example a decimal data error. The possible types are shown in Table 20 on page 515.

**Msg\_Ref\_Key**

A key used to uniquely identify the message.

**Exception\_Id**

Binary value of the exception id, for example, 0x123D. To display value, use conversion specifier %x as information is stored in hex value.

**Compare\_Data\_Len**

The length of the compare data.

**Signal\_Class**

Internal signal class.

**Priority**

The handler priority.

**Severity**

The message severity.

**reserved3**

A 4-byte reserved field.

**Msg\_Data\_Len**

The length of available message data.

**Mch\_Dep\_Data**

Machine-dependent data.

**Tgt\_Inv\_Type**

Invocation type. Macros are defined in <mimchobs.h>.

**Tgt\_Suspend**

Suspend pointer of the target.

**Ex\_Data**

The first 48 bytes of exception data.

The definition of `_CNL_Hndlr_Parms_T` is:

```
typedef _Packed struct {
    unsigned int    Block_Size;
    _INVFLAGS_T    Inv_Flags;
    char           reserved[8];
    _INVPTR        Invocation;
    _SPCPTR        Com_Area;
    _CNL_Mask_T    Mask;
} _CNL_Hndlr_Parms_T;
```

**Element****Description**

**Block\_Size**

The size of the parameter block passed to the cancel handler.

**Inv\_Flags**

Contains flags that are used by the system.

**reserved**

An eight byte reserved field.

**Invocation**

An invocation pointer to the invocation that is being cancelled.

**Com\_Area**

A pointer to the handler communications area defined by the cancel handler.

**Mask**

A 4 byte value indicating the cancel reason.

The following built-ins are defined in <except.h>:

**Built-in****Description****\_\_EXBDY**

The purpose of the \_\_EXBDY built-in or \_EXBDY macro is to act as a boundary for exception-sensitive operations. An exception-sensitive operation is one that may signal an exception. An EXBDY enables programmers to selectively suppress optimizations that do code motion. For example, a divide is an exception-sensitive operation because it can signal a divide-by-zero. An execution path containing both an EXBDY and a divide will perform the two in the same order with or without optimization. For example:

```

b = exp1;
c = exp2;
...
__EXBDY();
a = b/c;

```

**\_\_VBDY**

The purpose of a \_\_VBDY built-in or \_VBDY macro is to ensure the home storage locations are current for variables that are potentially used on exception paths. This ensures the visibility of the current values of variables in exception handlers. A VBDY enables programmers to selectively suppress optimizations, such as redundant store elimination and forward store motion to enforce sequential consistency of variable updates. In the following example, the VBDYs ensure that state is in its home storage location before each block of code that may signal an exception. A VBDY is often used in combination with an EXBDY to ensure that earlier assignments to state variables really update home storage locations and that later exception sensitive operations are not moved before these assignments.

```

state = 1;
__VBDY();
/* Do stuff that may signal an exception. */
state = 2;
__VBDY();
/* More stuff that may signal an exception. */
state = 3;
__VBDY();

```

For more information about built-ins, see the *ILE C/C++ for AS/400 MI Library Reference* .

---

## <float.h>

The <float.h> include file defines constants that specify the ranges of binary floating-point data types. For example, the maximum number of digits for objects of type `double` or the minimum exponent for objects of type `float`. In addition, if the macro variable `__STDC_WANT_DEC_FP__` is defined, the include file also defines constants that specify ranges of decimal floating-point data types. For example, the maximum number of digits for objects of type `_Decimal64` or the minimum exponent for objects of type `_Decimal32`.

---

## <inttypes.h>

The <inttypes.h> include file includes <stdint.h> and extends it with additional facilities.

The following macros are defined for format specifiers. These macros are defined for C programs. They are defined for C++ only when `__STDC_FORMAT_MACROS` is defined before <inttypes.h> is included.

<code>PRId8</code>	<code>PRIo8</code>	<code>PRIx8</code>	<code>SCnd16</code>	<code>SCnuLEAST16</code>
<code>PRId16</code>	<code>PRIo16</code>	<code>PRIx16</code>	<code>SCnd32</code>	<code>SCnuLEAST32</code>
<code>PRId32</code>	<code>PRIo32</code>	<code>PRIx32</code>	<code>SCnd64</code>	<code>SCnuLEAST64</code>
<code>PRId64</code>	<code>PRIo64</code>	<code>PRIx64</code>	<code>SCndFAST16</code>	<code>SCnuMAX</code>
<code>PRIdFAST8</code>	<code>PRIoFAST8</code>	<code>PRIXFAST8</code>	<code>SCndFAST32</code>	<code>SCnx16</code>
<code>PRIdFAST16</code>	<code>PRIoFAST16</code>	<code>PRIXFAST16</code>	<code>SCndFAST64</code>	<code>SCnx32</code>
<code>PRIdFAST32</code>	<code>PRIoFAST32</code>	<code>PRIXFAST32</code>	<code>SCndLEAST16</code>	<code>SCnx64</code>
<code>PRIdFAST64</code>	<code>PRIoFAST64</code>	<code>PRIXFAST64</code>	<code>SCndLEAST32</code>	<code>SCnxFAST16</code>
<code>PRIdLEAST8</code>	<code>PRIoLEAST8</code>	<code>PRIXLEAST8</code>	<code>SCndLEAST64</code>	<code>SCnxFAST32</code>
<code>PRIdLEAST16</code>	<code>PRIoLEAST16</code>	<code>PRIXLEAST16</code>	<code>SCndMAX</code>	<code>SCnxFAST64</code>
<code>PRIdLEAST32</code>	<code>PRIoLEAST32</code>	<code>PRIXLEAST32</code>	<code>SCNo16</code>	<code>SCnxLEAST16</code>
<code>PRIdLEAST64</code>	<code>PRIoLEAST64</code>	<code>PRIXLEAST64</code>	<code>SCNo32</code>	<code>SCnxLEAST32</code>
<code>PRIdMAX</code>	<code>PRIoMAX</code>	<code>PRIXMAX</code>	<code>SCNo64</code>	<code>SCnxLEAST64</code>
<code>PRi8</code>	<code>PRiU8</code>	<code>PRIX8</code>	<code>SCNoFAST16</code>	<code>SCnxMAX</code>
<code>PRi16</code>	<code>PRiU16</code>	<code>PRIX16</code>	<code>SCNoFAST32</code>	
<code>PRi32</code>	<code>PRiU32</code>	<code>PRIX32</code>	<code>SCNoFAST64</code>	
<code>PRi64</code>	<code>PRiU64</code>	<code>PRIX64</code>	<code>SCNoLEAST16</code>	
<code>PRiFAST8</code>	<code>PRiUFAST8</code>	<code>PRIXFAST8</code>	<code>SCNoLEAST32</code>	
<code>PRiFAST16</code>	<code>PRiUFAST16</code>	<code>PRIXFAST16</code>	<code>SCNoLEAST64</code>	
<code>PRiFAST32</code>	<code>PRiUFAST32</code>	<code>PRIXFAST32</code>	<code>SCNoMAX</code>	
<code>PRiFAST64</code>	<code>PRiUFAST64</code>	<code>PRIXFAST64</code>	<code>SCNu16</code>	
<code>PRiLEAST8</code>	<code>PRiULEAST8</code>	<code>PRIXLEAST8</code>	<code>SCNu32</code>	
<code>PRiLEAST16</code>	<code>PRiULEAST16</code>	<code>PRIXLEAST16</code>	<code>SCNu64</code>	
<code>PRiLEAST32</code>	<code>PRiULEAST32</code>	<code>PRIXLEAST32</code>	<code>SCNuFAST16</code>	
<code>PRiLEAST64</code>	<code>PRiULEAST64</code>	<code>PRIXLEAST64</code>	<code>SCNuFAST32</code>	
<code>PRiMAX</code>	<code>PRiUMAX</code>	<code>PRIXMAX</code>	<code>SCNuFAST64</code>	

---

## <langinfo.h>

The <langinfo.h> include file contains the declarations and definitions that are used by `n1_langinfo`.

---

## <limits.h>

The <limits.h> include file defines constants that specify the ranges of integer and character data types. For example, the maximum value for an object of type `char`.

---

## <locale.h>

The <locale.h> include file declares the `setlocale()`, `localeconv()`, and `wcslocaleconv()` library functions. These functions are useful for changing the C locale when you are creating applications for international markets.

The <locale.h> include file also declares the type struct lconv and the following macro definitions:

NULL	LC_ALL	LC_C	LC_C_FRANCE
LC_C_GERMANY	LC_C_ITALY	LC_C_SPAIN	LC_C_UK
LC_C_USA	LC_COLLATE	LC_CTYPE	LC_MESSAGES
LC_MONETARY	LC_NUMERIC	LC_TIME	LC_TOD
LC_UCS2_ALL	LC_UCS2_COLLATE	LC_UCS2_CTYPE	LC_UNI_ALL
LC_UNI_COLLATE	LC_UNI_CTYPE	LC_UNI_TIME	LC_UNI_NUMERIC
LC_UNI_MESSAGES	LC_UNI_MONITARY	LC_UNI_TOD	

---

## <math.h>

The <math.h> include file declares all the floating-point math functions:

acos	cosh	frexp	nextafter	sqrt
asin	erf	gamma	nextafterl	tan
atan	erfc	hypot	nexttoward	tanh
atan2	exp	ldexp	nexttowardl	
Bessel	fabs	log	pow	
ceil	floor	log10	sin	
cos	fmod	modf	sinh	

### Notes:

1. The Bessel functions are a group of functions named j0, j1, jn, y0, y1, and yn.
2. Floating-point numbers are only guaranteed 15 significant digits. This can greatly affect expected results if multiple floating-point numbers are used in a calculation.

<math.h> defines the macro HUGE\_VAL, which expands to a positive double expression, and possibly to infinity on systems that support infinity.

For all mathematical functions, a **domain error** occurs when an input argument is outside the range of values that are allowed for that function. In the event of a domain error, 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 zero.

---

## <mallocinfo.h>

Include file with \_C\_TS\_malloc\_info and \_C\_TS\_malloc\_debug.

---

## <monetary.h>

The <monetary.h> header file contains declarations and definitions that are related to the output of monetary quantities. The following monetary functions are defined: strfmon() and wcsfmon(). The strfmon() function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. The wcsfmon() function is available only when LOCALETYPE(\*LOCALEUTF) is specified on the compilation command.

---

## <nl\_types.h>

The <nl\_types.h> header file contains catalog definitions and the following catalog functions: `catclose()`, `catgets()`, and `catopen()`. These definitions are not available when either `LOCALETYPE(*CLD)` or `SYSIFCOPT(*NOIFSIO)` is specified on the compilation command.

---

## <pointer.h>

| The <pointer.h> include file contains typedefs and pragma directives for the i5/OS pointer types: space pointer, open pointer, invocation pointer, label pointer, system pointer, and suspend pointer. The typedefs | `_ANYPTR` and `_SPCPTRCN` are also defined in <pointer.h>.

---

## <recio.h>

The <recio.h> include file defines the types and macros, and prototypes functions for all the ILE C record input and output (I/O) operations.

The following functions are defined in <recio.h>:

<code>_Racquire</code>	<code>_Rclose</code>	<code>_Rcommit</code>	<code>_Rdelete</code>
<code>_Rdevatr</code>	<code>_Rfeod</code>	<code>_Rfeov</code>	<code>_Rformat</code>
<code>_Rindara</code>	<code>_Riofbk</code>	<code>_Rlocate</code>	<code>_Ropen</code>
<code>_Ropnfbk</code>	<code>_Rpgmdev</code>	<code>_Rreadd</code>	<code>_Rreadf</code>
<code>_Rreadindv</code>	<code>_Rreadk</code>	<code>_Rreadl</code>	<code>_Rreadn</code>
<code>_Rreadnc</code>	<code>_Rreadp</code>	<code>_Rreads</code>	<code>_Rrelease</code>
<code>_Rrlslck</code>	<code>_Rrollbck</code>	<code>_Rupdate</code>	<code>_Rupfb</code>
<code>_Rwrite</code>	<code>_Rwrited</code>	<code>_Rwriterd</code>	<code>_Rwrread</code>

The following positioning macros are defined in <recio.h>:

<code>__END</code>	<code>__END_FRC</code>	<code>__FIRST</code>	<code>__KEY_EQ</code>
<code>__KEY_GE</code>	<code>__KEY_GT</code>	<code>__KEY_LE</code>	<code>__KEY_LT</code>
<code>__KEY_NEXTEQ</code>	<code>__KEY_NEXTUNQ</code>	<code>__KEY_PREVEQ</code>	<code>__KEY_PREVUNQ</code>
<code>__KEY_LAST</code>	<code>__KEY_NEXT</code>	<code>__NO_POSITION</code>	<code>__PREVIOUS</code>
<code>__PRIOR</code>	<code>__RRN_EQ</code>	<code>__START</code>	<code>__START_FRC</code>
<code>__LAST</code>	<code>__NEXT</code>		

The following macros are defined in <recio.h>:

<code>__DATA_ONLY</code>	<code>__DFT</code>	<code>__NO_LOCK</code>	<code>__NULL_KEY_MAP</code>
--------------------------	--------------------	------------------------	-----------------------------

The following directional macros are defined in <recio.h>:

<code>__READ_NEXT</code>	<code>__READ_PREV</code>
--------------------------	--------------------------

The following functions and macros support locate or move mode:

<code>_Rreadd</code>	<code>_Rreadf</code>	<code>_Rreadindv</code>	<code>_Rreadk</code>
<code>_Rreadl</code>	<code>_Rreadn</code>	<code>_Rreadnc</code>	<code>_Rreadp</code>
<code>_Rreads</code>	<code>_Rupdate</code>	<code>_Rwrite</code>	<code>_Rwrited</code>
<code>_Rwriterd</code>	<code>_Rwrread</code>		

Any of the record I/O functions that include a buffer parameter may work in move mode or locate mode. In move mode, data is moved between the user-supplied buffer and the system buffer. In locate mode, the user must access the data in the system buffer. Pointers to the system buffers are exposed in the `_RFILE` structure. To specify that locate mode is being used, the buffer parameter of the record I/O function is coded as `NULL`.

A number of the functions include a size parameter. For move mode, this is the number of data bytes that are copied between the user-supplied buffer and the system buffer. All of the record I/O functions work with one record at a time regardless of the size that is specified. The size of this record is defined by the file description. It may not be equal to the size parameter that is specified by the user on the call to the record I/O functions. The amount of data that is moved between buffers is equal to the record length of the current record format or specified minimum size, whichever is smaller. The size parameter is ignored for locate mode.

The following types are defined in `recio.h`:

### Information for controlling opened record I/O operations

```
typedef _Packed struct {
    char                reserved1[16];
    volatile void *const *const in_buf;
    volatile void *const *const out_buf;
    char                reserved2[48];
    _RIOFB_T           riofb;
    char                reserved3[32];
    const unsigned int  buf_length;
    char                reserved4[28];
    volatile char *const in_null_map;
    volatile char *const out_null_map;
    volatile char *const null_key_map;
    char                reserved5[48];
    const int           min_length;
    short               null_map_len;
    short               null_key_map_len;
    char                reserved6[8];
} _RFILE;
```

Element	Description
<b>in_null_map</b>	Specifies which fields are to be considered NULL when you read from a database file.
<b>out_null_map</b>	Specifies which fields are to be considered NULL when you write to a database file.
<b>null_key_map</b>	Specifies which fields contain NULL if you are reading a database by key.
<b>null_map_len</b>	Specifies the lengths of the <code>in_null_map</code> and <code>out_null_map</code> .
<b>null_key_map_len</b>	Specifies the length of the <code>null_key_map</code> .

### Record I/O Feedback Information

```
typedef struct {
    unsigned char      *key;
    _Sys_Struct_T      *sysparm;
    unsigned long      rrrn;
    long               num_bytes;
    short              blk_count;
    char               blk_filled_by;
    int                dup_key :1;
    int                icf_locate :1;
    int                reserved1 :6;
    char               reserved2[20];
} _RIOFB_T;
```

<b>Element</b>	<b>Description</b>
<b>key</b>	If you are processing a file using a keyed sequence access path, this field contains a pointer to the key value of the record successfully positioned to, read or written.
<b>sysparm</b>	This field is a pointer to the major and minor return code for ICF, display, and printer files.
<b>rrn</b>	This field contains the relative record number of the record that was successfully positioned to, read or written.
<b>num_bytes</b>	This field contains the number of bytes that are read or are written.
<b>blk_count</b>	This field contains the number of records that remain in the block. If the file is open for input, blkrcd=y is specified, and a read function is called, this field will be updated with the number of records remaining in the block.
<b>blk_filled_by</b>	This field indicates the operation that filled the block. If the file is open for input, blkrcd=y is specified, and a read function is called. This field will be set to the <code>__READ_NEXT</code> macro if the <code>_Rreadn</code> function filled the block or to the <code>__READ_PREV</code> macro if the <code>_Rreadp</code> function filled the block.

### System-Specific Information

```
typedef struct {
    void          *sysparm_ext;
    _Maj_Min_rc_T  _Maj_Min;
    char          reserved1[12];
} _Sys_Struct_T;
```

### Major and Minor Return Codes

```
typedef struct {
    char  major_rc[2];
    char  minor_rc[2];
} _Maj_Min_rc_T;
```

The following macros are defined in `recio.h`:

<b><code>__FILENAME_MAX</code></b>	Expands to an integral constant expression that is the size of a character array large enough to hold the longest file name. This is the same as the stream I/O macro.
<b><code>__ROPEN_MAX</code></b>	Expands to an integral constant expression that is the maximum number of files that can be opened simultaneously.

The following null field macros are defined in `recio.h`:

<b>Element</b>	<b>Description</b>
<b><code>__CLEAR_NULL_MAP(file, type)</code></b>	Clears the null output field map that indicates that there are no null fields in the record to be written to <i>file</i> . <i>type</i> is a typedef that corresponds to the null field map for the current record format.
<b><code>__CLEAR_UPDATE_NULL_MAP(file, type)</code></b>	Clears the null input field map that indicates that no null fields are in the record to be written to <i>file</i> . <i>type</i> is a typedef that corresponds to the null field map for the current record format.

### **`_QRY_NULL_MAP(file, type)`**

Returns the number of fields that are null in the previously read record. *type* is a typedef that corresponds to the null field map for the current record format.

### **`_CLEAR_NULL_KEY_MAP(file, type)`**

Clears the null key field map so that it indicates no null key fields in the record to be written to file. *type* is a typedef that corresponds to the null key field map for the current record format.

### **`_SET_NULL_MAP_FIELD(file, type, field)`**

Sets the specified field in the output null field map so that *field* is considered NULL when the record is written to *file*.

### **`_SET_UPDATE_NULL_MAP_FIELD(file, type, field)`**

Sets the specified field in the input null field map so that *field* is considered null when the record is written to *file*. *type* is a typedef that corresponds to the null key field map for the record format.

### **`_QRY_NULL_MAP_FIELD(file, type, field)`**

Returns 1 if the specified field in the null input field map indicates that the *field* is to be considered null in the previously read record. If *field* is not null, it returns zero. *type* is a typedef that corresponds to the NULL key field map for the current record format.

### **`_SET_NULL_KEY_MAP_FIELD(file, type, field)`**

Sets the specified field map that indicates that the field will be considered null when the record is read from *file*. *type* is a typedef that corresponds to the null key field map for the current record format.

### **`_QRY_NULL_KEY_MAP(file, type)`**

Returns the number of fields that are null in the key of the previously read record. *type* is a typedef that corresponds to the null field map for the current record format.

### **`_QRY_NULL_KEY_MAP_FIELD(file, type, field)`**

Returns 1 if the specified field in the null key field map indicates that *field* is to be considered null in the previously read record. If *field* is not null, it returns zero. *type* is a typedef that corresponds to the null key field map for the current record format.

---

## **<regex.h>**

The <regex.h> include file defines the following regular expression functions:

`regcomp()`                      `regerror()`                      `regexexec()`                      `regfree()`

The <regex.h> include file also declares the *regmatch\_t* type, the *regex\_t* type, which is capable of storing a compiled regular expression, and the following macros:

Values of the *cflags* parameter of the `regcomp()` function:

`REG_BASIC`  
`REG_EXTENDED`  
`REG_ICASE`  
`REG_NEWLINE`  
`REG_NOSUB`

Values of the *eflags* parameter of the `regexexec()` function:

`REG_NOTBOL`  
`REG_NOTEOL`

Values of the *errcode* parameter of the `regerror()` function:

REG\_NOMATCH  
REG\_BADPAT  
REG\_ECOLLATE  
REG\_ECTYPE  
REG\_EESCAPE  
REG\_ESUBREG  
REG\_EBRACK  
REG\_EPAREN  
REG\_EBRACE  
REG\_BADBR  
REG\_ERANGE  
REG\_ESPACE  
REG\_BADRPT  
REG\_ECHAR  
REG\_EBOL  
REG\_EEOL  
REG\_ECOMP  
REG\_EEXEC  
REG\_LAST

These declarations and definitions are not available when `LOCALETYPE(*CLD)` is specified on the compilation command.

---

## <setjmp.h>

The `<setjmp.h>` include file declares the `setjmp()` macro and `longjmp` function. It also defines a buffer type, `jmp_buf`, that the `setjmp()` macro and `longjmp` function use to save and restore the program state.

---

## <signal.h>

The `<signal.h>` include file defines the values for signals and declares the `signal()` and `raise()` functions.

The `<signal.h>` include file also defines the following macros:

<code>SIGABRT</code>	<code>SIG_ERR</code>	<code>SIGILL</code>	<code>SIGOTHER</code>	<code>SIGUSR1</code>
<code>SIGALL</code>	<code>SIGFPE</code>	<code>SIGINT</code>	<code>SIGSEGV</code>	<code>SIGUSR2</code>
<code>SIG_DFL</code>	<code>SIG_IGN</code>	<code>SIGIO</code>	<code>SIGTERM</code>	

`<signal.h>` also declares the function `_getExcData`, an i5/OS extension to the C standard library.

---

## <stdarg.h>

The `<stdarg.h>` include file defines macros that allow you access to arguments in functions with variable-length argument lists: `va_arg()`, `va_start()`, and `va_end()`. The `<stdarg.h>` include file also defines the type `va_list`.

---

## <stddef.h>

The `<stddef.h>` include file declares the commonly used pointers, variables, and types as listed below:

**ptrdiff\_t**

typedef for the type of the difference of two pointers

**size\_t** typedef for the type of the value that is returned by `sizeof`

## wchar\_t

typedef for a wide character constant.

The <stddef.h> include file also defines the macros NULL and offsetof. NULL is a pointer that is guaranteed not to point 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)
```

The <stddef.h> include file also declares the extern variable \_EXCP\_MSGID, an i5/OS extension to C.

---

## <stdint.h>

The <stdint.h> include file declares sets of integer types that have specified widths and defines corresponding sets of macros. It also defines macros that specify limits of integer types corresponding to the types defined in other standard include files.

The following exact-width integer types are defined:

int8_t	int32_t	uint8_t	uint32_t
int16_t	int64_t	uint16_t	uint64_t

The following minimum-width integer types are defined:

int_least8_t	int_least32_t	uint_least8_t	uint_least32_t
int_least16_t	int_least64_t	uint_least16_t	uint_least64_t

The following fastest minimum-width integer types are defined:

int_fast8_t	int_fast32_t	uint_fast8_t	uint_fast32_t
int_fast16_t	int_fast64_t	uint_fast16_t	uint_fast64_t

The following greatest-width integer types are defined:

intmax_t
uintmax_t

The following macros are defined for limits of exact-width integer types (See note 1 on page 15):

INT8_MAX	INT16_MIN	INT64_MAX	UINT16_MAX
INT8_MIN	INT32_MAX	INT64_MIN	UINT32_MAX
INT16_MAX	INT32_MIN	UINT8_MAX	UINT64_MAX

The following macros are defined for limits of minimum-width integer types (See note 1 on page 15):

INT_LEAST8_MAX	INT_LEAST16_MIN	INT_LEAST64_MIN	UINT_LEAST16_MAX
INT_LEAST8_MIN	INT_LEAST32_MAX	INT_LEAST64_MIN	UINT_LEAST32_MAX
INT_LEAST16_MAX	INT_LEAST32_MIN	UINT_LEAST8_MAX	UINT_LEAST64_MAX

The following macros are defined for limits of fastest minimum-width integer types (See note 1 on page 15):

INT_FAST8_MAX	INT_FAST16_MIN	INT_FAST64_MIN	UINT_FAST16_MAX
INT_FAST8_MIN	INT_FAST32_MAX	INT_FAST64_MIN	UINT_FAST32_MAX
INT_FAST16_MAX	INT_FAST32_MIN	UINT_FAST8_MAX	UINT_FAST64_MAX

| The following macros are defined for limits of greatest-width integer types (See note 1):

INTMAX_MIN
INTMAX_MAX
UINTMAX_MAX

| The following macros are defined for limits for other integer types (See note 1):

PTRDIFF_MAX	SIG_ATOMIC_MIN	WCHAR_MIN
PTRDIFF_MIN	SIZE_MAX	WINT_MAX
SIG_ATOMIC_MAX	WCHAR_MAX	WINT_MIN

| The following macros are defined for minimum-width integer constant expressions (See note 2):

INT8_C	INT32_C	UINT8_C	UINT32_C
INT16_C	INT64_C	UINT16_C	UINT64_C

| The following macros are defined for greatest-width integer constant expressions (See note 2):

INTMAX_C
UINTMAX_C

| **Notes:**

- | 1. These macros are defined for C programs. They are defined for C++ only when `__STDC_LIMIT_MACROS` is defined before `<stdint.h>` is included.
- | 2. These macros are defined for C programs. They are defined for C++ only when `__STDC_CONSTANT_MACROS` is defined before `<stdint.h>` is included.

---

## <stdio.h>

The `<stdio.h>` include file defines constants, macros, and types, and declares stream input and output functions. The stream I/O functions are:

<code>_C_Get_Ssn_Handle</code>	<code>fprintf</code>	<code>fwrite</code>	<code>remove</code>	<code>vfscanf</code>
<code>clearerr</code>	<code>fputc</code>	<code>fwscanf</code> <sup>1</sup>	<code>rename</code>	<code>vwprintf</code> <sup>1</sup>
<code>fclose</code>	<code>_fputc</code>	<code>getc</code>	<code>rewind</code>	<code>vwscanf</code> <sup>1</sup>
<code>fdopen</code> <sup>2</sup>	<code>fputs</code>	<code>getchar</code>	<code>scanf</code>	<code>vprintf</code>
<code>feof</code>	<code>fputc</code> <sup>1</sup>	<code>gets</code>	<code>setbuf</code>	<code>vscanf</code>
<code>ferror</code>	<code>fputwc</code> <sup>1</sup>	<code>getwc</code> <sup>1</sup>	<code>setvbuf</code>	<code>vsscanf</code>
<code>fflush</code>	<code>fread</code>	<code>getwchar</code> <sup>1</sup>	<code>snprintf</code>	<code>vsprintf</code>
<code>fgetc</code>	<code>freopen</code>	<code>perror</code>	<code>sprintf</code>	<code>vsprintf</code>
<code>fgetpos</code>	<code>fscanf</code>	<code>printf</code>	<code>sscanf</code>	<code>vwprintf</code> <sup>1</sup>
<code>fgets</code>	<code>fseek</code>	<code>putc</code>	<code>tmpfile</code>	<code>wscanf</code> <sup>1</sup>
<code>fgetwc</code> <sup>1</sup>	<code>fsetpos</code>	<code>putchar</code>	<code>tmpnam</code>	<code>wfopen</code> <sup>2</sup>
<code>fgetws</code> <sup>1</sup>	<code>ftell</code>	<code>puts</code>	<code>ungetc</code>	<code>wprintf</code> <sup>1</sup>
<code>fileno</code> <sup>2</sup>	<code>fwide</code> <sup>1</sup>	<code>putwc</code> <sup>1</sup>	<code>ungetwc</code> <sup>1</sup>	<code>wscanf</code> <sup>1</sup>
<code>fopen</code>	<code>fwprintf</code> <sup>1</sup>	<code>putwchar</code> <sup>1</sup>	<code>vfprintf</code>	

**Note:** <sup>1</sup> These functions are not available when either `LOCALETYPE(*CLD)` or `SYSIFCOPT(*NOIFSIO)` is specified on the compilation command.

**Note:** <sup>2</sup> These functions are available when `SYSIFCOPT(*IFSIO)` is specified on the compilation command.

The `<stdio.h>` include file also defines the macros that are listed below. You can use these constants in your programs, but you should not alter their values.

**BUFSIZ** Specifies the buffer size that the `setbuf` library function will use when you are allocating buffers for stream I/O. This value establishes the size of system-allocated buffers and is used with `setbuf`.

**EOF** The value that is returned by an I/O function when the end of the file (or in some cases, an error) is found.

**FOPEN\_MAX**

The number of files that can be opened simultaneously.

**FILENAME\_MAX**

The longest file name that is supported. If there is no reasonable limit, `FILENAME_MAX` will be the recommended size.

**L\_tmpnam**

The size of the longest temporary name that can be generated by the `tmpnam` function.

**TMP\_MAX**

The minimum number of unique file names that can be generated by the `tmpnam` function.

**NULL** A pointer guaranteed not to point to a data object.

The `FILE` structure type is defined in `<stdio.h>`. Stream I/O 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.

When integrated file system is enabled with a compilation parameter `SYSIFCOPT(*IFSIO)`, `ifs.h` is included into `<stdio.h>`.

The C standard streams `stdin`, `stdout`, and `stderr` are also defined in `<stdio.h>`.

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 type `fpos_t` is defined in `<stdio.h>` for use with `fgetpos()` and `fsetpos()`.

See “`<stddef.h>`” on page 13 for more information about `NULL`.

---

## <stdlib.h>

The <stdlib.h> include file declares the following functions:

abort	_C_Quickpool_Report	ldiv	realloc	strtoul
abs	div	lldiv	srand	strtoull
atexit	exit	malloc	strtod	system
atof	free	mblen	strtod32	_ultoa <sup>1</sup>
atoi	_gcvt <sup>1</sup>	mbstowcs	strtod64	wcstombs
atol	getenv	mbtowc	strtod128	wctomb
bsearch	_itoa <sup>1</sup>	putenv	strtod	
calloc	_ltoa <sup>1</sup>	qsort	strtol	
_C_Quickpool_Debug	labs	rand	strtold	
_C_Quickpool_Init	llabs	rand_r	strtoll	

---

**Note:** <sup>1</sup> These functions are applicable to C++ only.

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

**NULL** The NULL pointer value.

### EXIT\_SUCCESS

Expands to 0; used by the `atexit` function.

### EXIT\_FAILURE

Expands to 8; used by the `atexit` function.

### RAND\_MAX

Expands to an integer that represents the largest number that the `rand` function can return.

### MB\_CUR\_MAX

Expands to an integral expression to represent the maximum number of bytes in a multibyte character for the current locale.

For more information about NULL and the types `size_t` and `wchar_t`, see “<stddef.h>” on page 13.

---

## <string.h>

The <string.h> include file declares the string manipulation functions:

memchr	strcat	strcspn	strncmp	strset <sup>1</sup>
memcmp	strchr	strdup <sup>1</sup>	strncpy	strspn
memcpy	strcmp	strerror	strnicmp <sup>1</sup>	strstr
memicmp <sup>1</sup>	strcmpi <sup>1</sup>	stricmp <sup>1</sup>	strnset <sup>1</sup>	strtok
memmove	strcoll	strlen	strpbrk	strtok_r
memset	strcpy	strncat	strchr	strxfrm

---

**Note:** <sup>1</sup> These functions are available for C++ programs. They are available for C only when the program defines the `__cplusplus__strings__` macro.

The <string.h> include file also defines the macro NULL, and the type `size_t`.

For more information about NULL and the type `size_t`, see “<stddef.h>” on page 13.

---

## <strings.h>

Contains the functions `strcasemp` and `strncasemp`.

---

## <time.h>

The <time.h> include file declares the time and date functions:

asctime	ctime_r	gmtime64	localtime_r	strptime <sup>1</sup>
asctime_r	ctime64_r	gmtime_r	localtime64_r	time
clock	difftime	gmtime64_r	mktime	time64
ctime	difftime64	localtime	mktime64	
ctime64	gmtime	localtime64	strftime	

| **Note:** <sup>1</sup> These functions are not available when LOCALETYPE(\*CLD) is specified on the compilation command.  
|

The <time.h> include file also provides:

- A structure tm that contains the components of a calendar time. See “gmtime() — Convert Time” on page 161 for a list of the tm structure members.
- A macro CLOCKS\_PER\_SEC equal to the number per second of the value that is returned by the clock function.
- | • Types clock\_t, time\_t, time64\_t, and size\_t.
- The NULL pointer value.

For more information about NULL and the type size\_t, see “<stddef.h>” on page 13.

---

## <wchar.h>

The <wchar.h> header file contains declarations and definitions that are related to the manipulation of wide character strings. Any functions which deal with files are accessible if SYSIFCOPT(\*IFSIO) is specified.

btowc <sup>1</sup>	mbsrtowcs <sup>1</sup>	wscmp	wcsrchr	wcswcs
fgetwc <sup>2</sup>	putwc <sup>2</sup>	wscoll <sup>1</sup>	wcsrtombs <sup>1</sup>	wcswidth <sup>1</sup>
fgetws <sup>2</sup>	putwchar <sup>2</sup>	wcscpy	wcsspn	wcsxfrm <sup>1</sup>
fputwc <sup>2</sup>	swprintf <sup>1</sup>	wcscspn	wcsstr	wctob <sup>1</sup>
fputws <sup>2</sup>	swscanf <sup>2</sup>	wcsftime <sup>1</sup>	wcstod <sup>1</sup>	wcwidth <sup>1</sup>
fwide <sup>2</sup>	ungetwc <sup>2</sup>	__wcsicmp <sup>1</sup>	wcstod32 <sup>1</sup>	wmemchr
fwprintf <sup>2</sup>	vfwprintf <sup>2</sup>	wcslen	wcstod64 <sup>1</sup>	wmemcmp
fwscanf <sup>2</sup>	vswscanf <sup>1</sup>	wcsncat	wcstod128 <sup>1</sup>	wmemcpy
getwc <sup>2</sup>	vswprintf <sup>1</sup>	wcsncmp	wcstok	wmemmove
getwchar <sup>2</sup>	vwprintf <sup>2</sup>	wcsncpy	wcstol <sup>1</sup>	wmemset
mbrlen <sup>1</sup>	wcrtomb <sup>1</sup>	__wcsnicmp <sup>1</sup>	wcstoll <sup>1</sup>	wprintf <sup>2</sup>
mbrtowc <sup>1</sup>	wcscat	wcspbrk	wcstoul <sup>1</sup>	wscanf <sup>2</sup>
mbsinit <sup>1</sup>	wcschr	wcsptime <sup>3</sup>	wcstoul1 <sup>1</sup>	

| **Note:** <sup>1</sup> These functions are not available when LOCALETYPE(\*CLD) is specified on the compilation command.

| **Note:** <sup>2</sup> These functions are available only when SYSIFCOPT(\*IFSIO) and LOCALETYPE(\*LOCALE) are specified on the compilation command.

| **Note:** <sup>3</sup> These functions are available only when LOCALETYPE(\*LOCALEUTF) is specified on the compilation command.

| <wchar.h> also defines the macro NULL and the types size\_t and wchar\_t.

| For more information about NULL and the types size\_t and wchar\_t, see “<stddef.h>” on page 13.

---

## <wctype.h>

The <wctype.h> header file declares the following wide character functions:

iswalnum	iswgraph	iswspace	towlower	wctrans
iswalpha	iswlower	iswupper	towupper	
iswcntrl	iswprint	iswxdigit	towctrans	
iswdigit	iswpunct	iswctype	wctype	

The <wctype.h> header file also contains declarations and definitions for wide character classification. These declarations and definitions are not available when LOCALETYPE(\*CLD) is specified on the compilation command.

---

## <xxcvt.h>

The <xxcvt.h> include file contains the declarations that are used by the QXXDTP, QXXDTPZ, QXXITOP, QXXITOPZ, QXXPTOI, QXXPTOD, QXXZTOD, and QXXZTOI conversion functions.

---

## <xxdtaa.h>

The <xxdtaa.h> include file contains the declarations for the data area interface functions QXXCHGDA, QXXRTVDA, and the type \_DTAA\_NAME\_T.

The definition of \_DTAA\_NAME\_T is:

```
typedef struct _DTAA_NAME_T {
    char dtaa_name[10];
    char dtaa_lib[10];
} _DTAA_NAME_T;
```

---

## <xxenv.h>

The <xxenv.h> include file contains the declarations for the QPXXCALL and QPXXDLTE EPM environment handling program. ILE procedures cannot be called from this interface.

The definition of \_ENVPGM\_T is:

```
typedef struct _ENVPGM_T {
    char pgmname[10];
    char pgmlib[10];
} _ENVPGM_T;
```

---

## <xxfdbk.h>

- | The <xxfdbk.h> include file contains the declarations that are used by the i5/OS feedback areas. To retrieve information from feedback areas, see “\_Riofbk() — Obtain I/O Feedback Information” on page 284 and “\_Ropnfbk() — Obtain Open Feedback Information” on page 293.

The following is an example of a type that is defined in the <xxfdbk.h> include file:

```
typedef _Packed struct _XXIOFB_T {
    short    file_dep_fb_offset;
    int      write_count;
    int      read_count;
    int      write_read_count;
    int      other_io_count;
    char     reserved1;
    char     cur_operation;
    char     rec_format[10];
    char     dev_class[2];
};
```

```
    char    dev_name[10];
    int     last_io_rec_len;
    char    reserved2[80];
    short   num_recs_retrieved;
    short   last_io_rec_len2;
    char    reserved3[2];
    int     cur_blk_count;
    char    reserved4[8];
} _XXIOFB_T;
```

For further information about the open feedback areas, see the Files and file systems category in the Information Center.

---

## Machine Interface (MI) Include Files

See the *ILE C/C++ for AS/400 MI Library Reference* for a description of the MI header files.

---

## Chapter 2. Library Functions

This chapter describes the standard C/C++ library functions and the ILE C/C++ extensions to the library functions, except for the ILE C/C++ MI functions. See the *ILE C/C++ for AS/400 MI Library Reference* for more information about the MI functions.

Each library function that is listed in this section contains:

- A format description that shows the include file that declares the function.
- The data type that is returned by the function.
- The required data types of the arguments to the function.

This example shows the format of the `log()` function:

```
#include <math.h>
double log(double x);
```

The example shows that:

- you must include the file `math.h` in the program.
- the `log()` function returns type `double`.
- the `log()` function requires an argument `x` of type `double`.

Examples throughout the section illustrate the use of library functions and are not necessarily complete.

This chapter lists the library functions in alphabetic order. If you are unsure of the function you want to use, see the summary of the library functions in “The C/C++ Library.”

**Note:** All functions are considered threadsafe unless noted otherwise.

---

## The C/C++ Library

This chapter summarizes the available C/C++ library functions and their location in this book. It also briefly describes what the function does. Each library function is listed according to the type of function it performs.

### Error Handling

Function	Header File	Page	Description
<code>assert()</code>	<code>assert.h</code>	43	Prints diagnostic messages.
<code>atexit()</code>	<code>stdlib.h</code>	45	Registers a function to be executed at program end.
<code>clearerr()</code>	<code>stdio.h</code>	62	Resets error indicators.
<code>feof()</code>	<code>stdio.h</code>	95	Tests end-of-file indicator for stream input.
<code>ferror()</code>	<code>stdio.h</code>	95	Tests the error indicator for a specified stream.
<code>_GetExcData()</code>	<code>signal.h</code>	154	Retrieves information about an exception from within a C signal handler. This function is not defined when <code>SYSIFCOPT(*SYNCSIGNAL)</code> is specified on the compilation command.

Function	Header File	Page	Description
perror()	stdio.h	227	Prints an error message to stderr.
raise()	signal.h	255	Initiates a signal.
signal()	signal.h	346	Allows handling of an interrupt signal from the operating system.
strerror()	string.h	367	Retrieves pointer to system error message.

## Searching and Sorting

Function	Header File	Page	Description
bsearch()	stdlib.h	51	Performs a binary search of a sorted array.
qsort()	stdlib.h	245	Performs a quick sort on an array of elements.

## Mathematical

Function	Header File	Page	Description
abs()	stdlib.h	37	Calculates the absolute value of an integer.
ceil()	math.h	61	Calculates the double value representing the smallest integer that is greater than or equal to a number.
div()	stdlib.h	86	Calculates the quotient and remainder of an integer.
erf()	math.h	87	Calculates the error function.
erfc()	math.h	87	Calculates the error function for large numbers.
exp()	math.h	89	Calculates an exponential function.
fabs()	math.h	90	Calculates the absolute value of a floating-point number.
floor()	math.h	107	Calculates the double value representing the largest integer that is less than or equal to a number.
fmod()	math.h	108	Calculates the floating-point remainder of one argument divided by another.
frexp()	math.h	132	Separates a floating-point number into its mantissa and exponent.
gamma()	math.h	150	Calculates the gamma function.
hypot()	math.h	168	Calculates the hypotenuse.
labs()	stdlib.h	177	Calculates the absolute value of a long integer.
llabs()	stdlib.h	177	Calculates the absolute value of a long long integer.
ldexp()	math.h	178	Multiplies a floating-point number by an integral power of 2.
ldiv()	stdlib.h	179	Calculates the quotient and remainder of a long integer.

Function	Header File	Page	Description
lldiv()	stdlib.h	179	Calculates the quotient and remainder of a long long integer.
log()	math.h	191	Calculates natural logarithm.
log10()	math.h	191	Calculates base 10 logarithm.
modf()	math.h	222	Calculates the signed fractional portion of the argument.
nextafter()	math.h	223	Calculates the next representable floating-point value.
nextafterl()	math.h	223	Calculates the next representable floating-point value.
nexttoward()	math.h	223	Calculates the next representable floating-point value.
nexttowardl()	math.h	223	Calculates the next representable floating-point value.
pow()	math.h	228	Calculates the value of an argument raised to a power.
sqrt()	math.h	353	Calculates the square root of a number.

### *Trigonometric Functions*

Function	Header File	Page	Description
acos()	math.h	38	Calculates the arc cosine.
asin()	math.h	42	Calculates the arc sine.
atan()	math.h	44	Calculates the arc tangent.
atan2()	math.h	44	Calculates the arc tangent.
cos()	math.h	64	Calculates the cosine.
cosh()	math.h	65	Calculates the hyperbolic cosine.
sin()	math.h	348	Calculates the sine.
sinh()	math.h	349	Calculates the hyperbolic sine.
tan()	math.h	409	Calculates the tangent.
tanh()	math.h	410	Calculates the hyperbolic tangent.

### *Bessel Functions*

Function	Header File	Page	Description
j0()	math.h	50	0 order differential equation of the first kind.
j1()	math.h	50	1st order differential equation of the first kind.
jn()	math.h	50	<i>n</i> th order differential equation of the first kind.
y0()	math.h	50	0 order differential equation of the second kind.
y1()	math.h	50	1st order differential equation of the second kind.

Function	Header File	Page	Description
yn()	math.h	50	<i>n</i> th order differential equation of the second kind.

## Time Manipulation

Function	Header File	Page	Description
asctime()	time.h	39	Converts time stored as a structure to a character string in storage.
asctime_r()	time.h	41	Converts time stored as a structure to a character string in storage. (Restartable version of asctime())
clock()	time.h	63	Determines processor time.
ctime()	time.h	71	Converts time stored as a long value to a character string.
ctime64()	time.h	73	Converts time stored as a long long value to a character string.
ctime_r()	time.h	74	Converts time stored as a long value to a character string. (Restartable version of ctime())
ctime64_r()	time.h	76	Converts time stored as a long long value to a character string. (Restartable version of ctime64())
difftime()	time.h	82	Calculates the difference between two times.
difftime64()	time.h	84	Calculates the difference between two times.
gmtime()	time.h	161	Converts time to Coordinated Universal Time structure.
gmtime_r()	time.h	165	Converts time to Coordinated Universal Time structure. (Restartable version of gmtime())
gmtime64()	time.h	163	Converts time to Coordinated Universal Time structure.
gmtime64_r()	time.h	167	Converts time to Coordinated Universal Time structure. (Restartable version of gmtime64())
localtime()	time.h	185	Converts time to local time.
localtime64()	time.h	187	Converts time to local time.
localtime_r()	time.h	188	Converts time to local time. (Restartable version of localtime())
localtime64_r()	time.h	189	Converts time to local time. (Restartable version of localtime64())
mktime()	time.h	218	Converts local time into calendar time.
mktime64()	time.h	220	Converts local time into calendar time.
time()	time.h	411	Returns the time in seconds.
time64()	time.h	412	Returns the time in seconds.

## Type Conversion

Function	Header File	Page	Description
atof()	stdlib.h	46	Converts a character string to a floating-point value.
atoi()	stdlib.h	48	Converts a character string to an integer.
atol()	stdlib.h	49	Converts a character string to a long integer.
atoll()	stdlib.h	49	Converts a character string to a long integer.
_gcvt()	stdlib.h	151	Converts a floating-point value to a string.
_itoa()	stdlib.h	176	Converts an integer to a string.
_ltoa()	stdlib.h	192	Converts a long integer to a string.
strtod()	stdlib.h	392	Converts a character string to a double-precision binary floating-point value.
strtod32()	stblib.h	395	Converts a character string to a single-precision decimal floating-point value.
strtod64()	stblib.h	395	Converts a character string to a double-precision decimal floating-point value.
strtod128()	stblib.h	395	Converts a character string to a quad-precision decimal floating-point value.
strtof()	stblib.h	392	Converts a character string to a binary floating-point value.
strtol()	stdlib.h	400	Converts a character string to a long integer.
strtold()	stdlib.h	392	Converts a character string to a double-precision binary floating-point value.
strtoll()	stdlib.h	400	Converts a character string to a long long integer.
strtoul()	stdlib.h	403	Converts a string to an unsigned long integer.
strtoull()	stdlib.h	403	Converts a string to an unsigned long long integer.
toascii()	ctype.h	415	Converts a character to the corresponding ASCII value.
_ultoa()	stdlib.h	419	Converts an unsigned long integer to a string.
wcstod()	wchar.h	476	Converts a wide-character string to a double-precision binary floating-point value.
wcstod32()	wchar.h	478	Converts a wide-character string to a single-precision decimal floating-point value.
wcstod64()	wchar.h	478	Converts a wide-character string to a double-precision decimal floating-point value.
wcstod128()	wchar.h	478	Converts a wide-character string to a quad-precision decimal floating-point value.
wcstol()	wchar.h	481	Converts a wide-character string to a long integer.
wcstoll()	wchar.h	481	Converts a wide-character string to a long long integer.
wcstoul()	wchar.h	486	Converts a wide-character string to an unsigned long integer.
wcstoull()	wchar.h	486	Converts a wide-character string to an unsigned long long integer.

## Conversion

Function	Header File	Page	Description
QXXDTP()	xxcvt.h	248	Converts a floating-point value to a packed decimal value.
QXXDTPZ()	xxcvt.h	249	Converts a floating-point value to a zoned decimal value.
QXXITP()	xxcvt.h	250	Converts an integer value to a packed decimal value.
QXXITPZ()	xxcvt.h	250	Converts an integer value to a zoned decimal value.
QXXPTD()	xxcvt.h	251	Converts a packed decimal value to a floating-point value.
QXXPTDI()	xxcvt.h	252	Converts a packed decimal value to an integer value.
QXXZTD()	xxcvt.h	254	Converts a zoned decimal value to a floating-point value.
QXXZTDI()	xxcvt.h	255	Converts a zoned decimal value to an integer value.

## Record Input/Output

Function	Header File	Page	Description
_Racquire()	recio.h	257	Prepares a device for record I/O operations.
_Rclose()	recio.h	258	Closes a file that is opened for record I/O operations.
_Rcommit()	recio.h	259	Completes the current transaction, and establishes a new commitment boundary.
_Rdelete()	recio.h	261	Deletes the currently locked record.
_Rdevatr()	recio.h xxfdbk.h	263	Returns a pointer to a copy of the device attributes feedback area for the file reference by fp and the device pgmdev.
_Rfeed()	recio.h	278	Forces an end-of-file condition for the file referenced by fp.
_Rfeov()	recio.h	279	Forces an end-of-volume condition for tapes.
_Rformat()	recio.h	280	Sets the record format to fmt for the file referenced by fp.
_Rindara()	recio.h	282	Sets up the separate indicator area to be used for subsequent record I/O operations.
_Riofbk()	recio.h xxfdbk.h	284	Returns a pointer to a copy of the I/O feedback area for the file referenced by fp.
_Rlocate()	recio.h	286	Positions to the record in the files associated with fp and specified by the key, klen_rrn and opt parameters.
_Ropen()	recio.h	289	Opens a file for record I/O operations.
_Ropnfbk()	recio.h xxfdbk.h	293	Returns a pointer to a copy of the open feedback area for the file referenced by fp.

Function	Header File	Page	Description
<code>_Rpgmdev()</code>	recio.h	294	Sets the default program device.
<code>_Rreadd()</code>	recio.h	295	Reads a record by relative record number.
<code>_Rreadf()</code>	recio.h	297	Reads the first record.
<code>_Rreadindv()</code>	recio.h	299	Reads data from an invited device.
<code>_Rreadk()</code>	recio.h	302	Reads a record by key.
<code>_Rreadl()</code>	recio.h	305	Reads the last record.
<code>_Rreadn()</code>	recio.h	306	Reads the next record.
<code>_Rreadnc()</code>	recio.h	308	Reads the next changed record in the subfile.
<code>_Rreadp()</code>	recio.h	310	Reads the previous record.
<code>_Rreads()</code>	recio.h	312	Reads the same record.
<code>_Rrelease()</code>	recio.h	314	Makes the specified device ineligible for record I/O operations.
<code>_Rr1slck()</code>	recio.h	316	Releases the currently locked record.
<code>_Rrollbck()</code>	recio.h	317	Reestablishes the last commitment boundary as the current commitment boundary.
<code>_Rupdate()</code>	recio.h	319	Writes to the record that is currently locked for update.
<code>_Rupfb()</code>	recio.h	320	Updates the feedback structure with information about the last record I/O operation.
<code>_Rwrite()</code>	recio.h	322	Writes a record to the end of the file.
<code>_Rwrited()</code>	recio.h	324	Writes a record by relative record number. It will only write over deleted records.
<code>_Rwriterd()</code>	recio.h	327	Writes and reads a record.
<code>_Rwrread()</code>	recio.h	328	Functions as <code>_Rwriterd()</code> , except separate buffers can be specified for input and output data.

## Stream Input/Output

### *Formatted Input/Output*

Function	Header File	Page	Description
<code>fprintf()</code>	stdio.h	116	Formats and prints characters to the output stream.
<code>fscanf()</code>	stdio.h	132	Reads data from a stream into locations given by arguments.
<code>fwprintf()</code>	stdio.h	143	Formats data as wide characters, and writes to a stream.
<code>fwscanf()</code>	stdio.h	147	Reads wide data from stream into locations given by arguments.
<code>printf()</code>	stdio.h	229	Formats and prints characters to stdout.
<code>scanf()</code>	stdio.h	330	Reads data from stdin into locations given by arguments.
<code>snprintf()</code>	stdio.h	350	Same as <code>sprintf</code> , except that the <code>snprintf()</code> function will stop after <code>n</code> characters have been written to a buffer.

Function	Header File	Page	Description
sprintf()	stdio.h	352	Formats and writes characters to a buffer.
sscanf()	stdio.h	355	Reads data from a buffer into locations given by arguments.
swprintf()	wchar.h	406	Formats and writes wide characters to buffer.
swscanf()	wchar.h	407	Reads wide data from a buffer into locations given by arguments.
vfprintf()	stdio.h stdarg.h	425	Formats and prints characters to the output stream using a variable number of arguments.
vfscanf()	stdarg.h stdio.h	427	Reads data from a specified stream into locations given by a variable number of arguments.
vfwprintf()	stdio.h stdarg.h	428	Formats argument data as wide characters and writes to a stream using a variable number of arguments.
vfwscanf()	stdarg.h stdio.h	430	Reads wide data from a specified stream into locations given by a variable number of arguments.
vprintf()	stdarg.h stdio.h	432	Formats and writes characters to stdout using a variable number of arguments.
vscanf()	stdarg.h stdio.h	433	Reads data from stdin into locations given by a variable number of arguments.
vsnprintf()	stdio.h stdarg.h	435	Same as vsprintf, except that the vsnprintf function will stop after n characters have been written to a buffer.
vsprintf()	stdarg.h stdio.h	436	Formats and writes characters to a buffer using a variable number of arguments.
vsscanf()	stdarg.h stdio.h	437	Reads data from a buffer into locations given by a variable number of arguments.
vswprintf()	wchar.h stdarg.h	439	Formats and writes wide characters to buffer using a variable number of arguments.
vswscanf()	stdarg.h wchar.h	441	Reads wide data from a buffer into locations given by a variable number of arguments.
vwprintf()	wchar.h stdarg.h	443	Formats and writes wide characters to stdout using a variable number of arguments.
vwscanf()	stdarg.h stdio.h	445	Reads wide data from stdin into locations given by a variable number of arguments.
wprintf()	stdio.h	503	Formats and writes wide characters to stdout
wscanf()	stdio.h	504	Reads wide data from stdin into locations given by arguments.

## Character and String Input/Output

Function	Header File	Page	Description
fgetc()	stdio.h	98	Reads a character from a specified input stream.
fgets()	stdio.h	101	Reads a string from a specified input stream.
fgetwc()	stdio.h	102	Reads a wide character from a specified stream.
fgetws()	stdio.h	104	Reads a wide-character string from a specified stream.
fputc()	stdio.h	118	Prints a character to a specified output stream.
_fputchar()	stdio.h	120	Writes a character to stdout.
fputs()	stdio.h	121	Prints a string to a specified output stream.
fputwc()	stdio.h	122	Writes a wide character to a specified stream.
fputws()	stdio.h	124	Writes a wide-character string to a specified stream.
getc()	stdio.h	152	Reads a character from a specified input stream.
getchar()	stdio.h	152	Reads a character from stdin.
gets()	stdio.h	156	Reads a line from stdin.
getwc()	stdio.h	157	Reads a wide character from a specified stream.
getwchar()	stdio.h	159	Gets a wide character from stdin.
putc()	stdio.h	239	Prints a character to a specified output stream.
putchar()	stdio.h	239	Prints a character to stdout.
puts()	stdio.h	241	Prints a string to stdout.
putwc()	stdio.h	242	Writes a wide character to a specified stream.
putwchar()	stdio.h	244	Writes a wide character to stdout.
ungetc()	stdio.h	420	Pushes a character back onto a specified input stream.
ungetwc()	stdio.h	422	Pushes a wide character back onto a specified input stream.

## Direct Input/Output

Function	Header File	Page	Description
fread()	stdio.h	126	Reads items from a specified input stream.
fwrite()	stdio.h	146	Writes items to a specified output stream.

## File Positioning

Function	Header File	Page	Description
fgetpos()	stdio.h	99	Gets the current position of the file pointer.
fseek()	stdio.h	134	Moves the file pointer to a new location.
fseeko()	stdio.h	134	Same as fseek().
fsetpos()	stdio.h	136	Moves the file pointer to a new location.

Function	Header File	Page	Description
ftell()	stdio.h	138	Gets the current position of the file pointer.
ftello()	stdio.h	138	Same as ftell().
rewind()	stdio.h	276	Repositions the file pointer to the beginning of the file.

### *File Access*

Function	Header File	Page	Description
fclose()	stdio.h	91	Closes a specified stream.
fdopen()	stdio.h	92	Associates an input or output stream with a file.
fflush()	stdio.h	96	Causes the system to write the contents of a buffer to a file.
fopen()	stdio.h	109	Opens a specified stream.
freopen()	stdio.h	130	Closes a file and reassigns a stream.
fwide()	stdio.h	140	Determines stream orientation.
setbuf()	stdio.h	336	Allows control of buffering.
setvbuf()	stdio.h	344	Controls buffering and buffer size for a specified stream.
wfopen()	stdio.h	498	Opens a specified stream, accepting file name and mode as wide characters.

### *File Operations*

Function	Header File	Page	Description
fileno()	stdio.h	106	Determines the file handle.
remove()	stdio.h	274	Deletes a specified file.
rename()	stdio.h	275	Renames a specified file.
tmpfile()	stdio.h	414	Creates a temporary file and returns a pointer to that file.
tmpnam()	stdio.h	414	Produces a temporary file name.

## Handling Argument Lists

Function	Header File	Page	Description
va_arg()	stdarg.h	423	Allows access to variable number of function arguments.
va_end()	stdarg.h	423	Allows access to variable number of function arguments.
va_start()	stdarg.h	423	Allows access to variable number of function arguments.

## Pseudorandom Numbers

Function	Header File	Page	Description
rand(), rand_r()	stdlib.h	256	Returns a pseudorandom integer. (rand_r() is the restartable version of rand().)
srand()	stdlib.h	354	Sets the starting point for pseudorandom numbers.

## Dynamic Memory Management

Function	Header File	Page	Description
calloc()	stdlib.h	55	Reserves storage space for an array and initializes the values of all elements to 0.
_C_Quickpool_Debug()	stdlib.h	66	Modifies Quick Pool memory characteristics.
_C_Quickpool_Init()	stdlib.h	68	Initializes the use of the Quick Pool memory management algorithm.
_C_Quickpool_Report()	stdlib.h	70	Generates a spooled file that contains a snapshot of the memory used by the Quick Pool memory management algorithm in the current activation group.
_C_TS_malloc_debug()	mallocinfo.h	77	Returns the same information as _C_TS_malloc_info, plus produces a spool file of detailed information about the memory structure used by malloc functions when compiled with teraspace.
_C_TS_malloc_info()	mallocinfo.h	79	Returns the current memory usage information.
free()	stdlib.h	128	Frees storage blocks.
malloc()	stdlib.h	195	Reserves storage blocks.
realloc()	stdlib.h	264	Changes storage size allocated for an object.

## Memory Objects

Function	Header File	Page	Description
memchr()	string.h	212	Searches a buffer for the first occurrence of a given character.
memcmp()	string.h	213	Compares two buffers.
memcpy()	string.h	214	Copies a buffer.
memicmp()	string.h	215	Compare two buffers without regard to case.
memmove()	string.h	217	Moves a buffer.
memset()	string.h	218	Sets a buffer to a given value.
wmemchr()	wchar.h	498	Locates a wide character in a wide-character buffer.
wmemcmp()	wchar.h	499	Compares two wide-character buffers.
wmemcpy()	wchar.h	500	Copies a wide-character buffer.
wmemmove()	wchar.h	501	Moves a wide-character buffer.

Function	Header File	Page	Description
wmemset()	wchar.h	502	Sets a wide-character buffer to a given value.

## Environment Interaction

Function	Header File	Page	Description
abort()	stdlib.h	36	Ends a program abnormally.
_C_Get_Ssn_Handle()	stdio.h	55	Returns a handle to the C session for use with DSM APIs.
exit()	stdlib.h	88	Ends the program normally if called in the initial thread.
getenv()	stdlib.h	154	Searches environment variables for a specified variable.
localeconv()	locale.h	181	Formats numeric quantities in struct lconv according to the current locale.
longjmp()	setjmp.h	193	Restores a stack environment.
nl_langinfo()	langinfo.h	224	Retrieves information from the current locale.
putenv()	stdlib.h	240	Sets the value of an environment variable by altering an existing variable or creating a new one.
setjmp()	setjmp.h	338	Saves a stack environment.
setlocale()	locale.h	339	Changes or queries locale.
system()	stdlib.h	408	Passes a string to the operating system's command interpreter.
wcslocaleconv()	locale.h	462	Formats numeric quantities in struct wclconv according to the current locale.

## String Operations

Function	Header File	Page	Description
strcasemp()	strings.h	357	Compares strings without case sensitivity.
strcat()	string.h	358	Concatenates two strings.
strchr()	string.h	359	Locates the first occurrence of a specified character in a string.
strcmp()	string.h	360	Compares the value of two strings.
strcmpi()	string.h	362	Compares the value of two strings without regard to case.
strcoll()	string.h	363	Compares the locale-defined value of two strings.
strcpy()	string.h	364	Copies one string into another.
strcspn()	string.h	365	Finds the length of the first substring in a string of characters not in a second string.
strdup()	string.h	366	Duplicates a string.

Function	Header File	Page	Description
strfmon()	string.h	368	Converts monetary value to string.
strftime()	time.h	370	Converts date and time to a formatted string.
stricmp()	string.h	374	Compares the value of two strings without regard to case.
strlen()	string.h	375	Calculates the length of a string.
strncasecmp()	strings.h	376	Compares strings without case sensitivity.
strncat()	string.h	377	Adds a specified length of one string to another string.
strncmp()	string.h	379	Compares two strings up to a specified length.
strncpy()	string.h	380	Copies a specified length of one string into another.
strnicmp()	string.h	382	Compares the value of two substrings without regard to case.
strnset()	string.h	383	Sets character in a string.
strpbrk()	string.h	384	Locates specified characters in a string.
strptime()	time.h	385	Converts string to formatted time.
strrchr()	string.h	389	Locates the last occurrence of a character within a string.
strspn()	string.h	390	Locates the first character in a string that is not part of specified set of characters.
strstr()	string.h	391	Locates the first occurrence of a string in another string.
strtok()	string.h	398	Locates a specified token in a string.
strtok_r()	string.h	399	Locates a specified token in a string. (Restartable version of strtok()).
strxfrm()	string.h	404	Transforms strings according to locale.
wcsftime()	wchar.h	458	Converts to formatted date and time.
wcsptime()	wchar.h	469	Converts string to formatted time.
wcsstr()	wchar.h	475	Locates a wide-character substring.
wcstok()	wchar.h	480	Tokenizes a wide-character string.

## Character Testing

Function	Header File	Page	Description
isalnum()	ctype.h	169	Tests for alphanumeric characters.
isalpha()	ctype.h	169	Tests for alphabetic characters.
isascii()	ctype.h	171	Tests for ASCII values.
isblank()	ctype.h	172	Tests for blank or tab characters.
iscntrl()	ctype.h	169	Tests for control characters.
isdigit()	ctype.h	169	Tests for decimal digits.
isgraph()	ctype.h	169	Tests for printable characters excluding the space.
islower()	ctype.h	169	Tests for lowercase letters.

Function	Header File	Page	Description
isprint()	ctype.h	169	Tests for printable characters including the space.
ispunct()	ctype.h	169	Tests for punctuation characters as defined in the locale.
isspace()	ctype.h	169	Tests for white-space characters.
isupper()	ctype.h	169	Tests for uppercase letters.
isxdigit()	ctype.h	169	Tests for wide hexadecimal digits 0 through 9, a through f, or A through F.

## Multibyte Character Testing

Function	Header File	Page	Description
iswalnum()	wctype.h	173	Tests for wide alphanumeric characters.
iswalphabeta()	wctype.h	173	Tests for wide alphabetic characters.
iswcntrl()	wctype.h	173	Tests for wide control characters.
iswctype()	wctype.h	175	Tests for character property.
iswdigit()	wctype.h	173	Tests for wide decimal digits.
iswgraph()	wctype.h	173	Tests for wide printing characters excluding the space.
iswlower()	wctype.h	173	Tests for wide lowercase letters.
iswprint()	wctype.h	173	Tests for wide printing characters.
iswpunct()	wctype.h	173	Tests for wide punctuation characters as defined in the locale.
iswspace()	wctype.h	173	Tests for wide whitespace characters.
iswupper()	wctype.h	173	Tests for wide uppercase letters.
iswxdigit()	wctype.h	173	Tests for wide hexadecimal digits 0 through 9, a through f, or A through F.

## Character Case Mapping

Function	Header File	Page	Description
tolower()	ctype.h	416	Converts a character to lowercase.
toupper()	ctype.h	416	Converts a character to uppercase.
towlower()	ctype.h	418	Converts a wide character to lowercase.
towupper()	ctype.h	418	Converts a wide character to uppercase.

## Multibyte Character Manipulation

Function	Header File	Page	Description
btowc()	stdio.h wchar.h	53	Converts a single byte to a wide character.
mblen()	stdlib.h	197	Determines the length of a multibyte character.

Function	Header File	Page	Description
<code>mbrlen()</code>	<code>stdlib.h</code>	199	Determines the length of a multibyte character. (Restartable version of <code>mblen()</code> )
<code>mbrtowc()</code>	<code>stdlib.h</code>	201	Converts a multibyte character to a wide character. (Restartable version of <code>mbtowc()</code> )
<code>mbsinit()</code>	<code>stdlib.h</code>	205	Tests state object for initial state.
<code>mbsrtowcs()</code>	<code>stdlib.h</code>	206	Converts a multibyte string to a wide character string. (Restartable version of <code>mbstowcs()</code> )
<code>mbstowcs()</code>	<code>stdlib.h</code>	207	Converts a multibyte string to a wide character string.
<code>mbtowc()</code>	<code>stdlib.h</code>	211	Converts multibyte characters to a wide character.
<code>towctrans()</code>	<code>wctype.h</code>	417	Translates wide character.
<code>wcrtomb()</code>	<code>stdlib.h</code>	446	Converts a wide character to a multibyte character. (Restartable version of <code>wctomb()</code> ).
<code>wcscat()</code>	<code>wchar.h</code>	451	Concatenates wide character strings.
<code>wcschr()</code>	<code>wchar.h</code>	452	Searches a wide character string for a wide character.
<code>wscmp()</code>	<code>wchar.h</code>	453	Compares two wide character strings.
<code>wscoll()</code>	<code>wchar.h</code>	455	Compares the locale-defined value of two wide-character strings.
<code>wscpy()</code>	<code>wchar.h</code>	456	Copies a wide character string.
<code>wscspn()</code>	<code>wchar.h</code>	457	Searches a wide character string for characters.
<code>__wcsicmp()</code>	<code>wchar.h</code>	460	Compares two wide character strings without regard to case.
<code>wcslen()</code>	<code>wchar.h</code>	461	Finds length of a wide character string.
<code>wcsncat()</code>	<code>wchar.h</code>	463	Concatenates a wide character string segment.
<code>wcsncmp()</code>	<code>wchar.h</code>	464	Compares wide character string segments.
<code>wcsncpy()</code>	<code>wchar.h</code>	466	Copies wide character string segments.
<code>__wcsnicmp()</code>	<code>wchar.h</code>	467	Compares two wide character substrings without regard to case.
<code>wcspbrk()</code>	<code>wchar.h</code>	468	Locates wide characters in string.
<code>wcsrchr()</code>	<code>wchar.h</code>	471	Locates wide character in string.
<code>wcsrtombs()</code>	<code>stdlib.h</code>	473	Converts a wide character string to a multibyte character string. (Restartable version of <code>wcstombs()</code> ).
<code>wcsspn()</code>	<code>wchar.h</code>	474	Finds offset of first nonmatching wide character.
<code>wcstombs()</code>	<code>stdlib.h</code>	483	Converts a wide character string to a multibyte character string.
<code>wcswcs()</code>	<code>wchar.h</code>	488	Locates a wide character string in another wide character string.
<code>wcswidth()</code>	<code>wchar.h</code>	489	Determines the display width of a wide character string.
<code>wcsxfrm()</code>	<code>wchar.h</code>	490	Transforms wide-character strings according to locale.

Function	Header File	Page	Description
wctob()	stdlib.h	491	Converts a wide character to a single byte.
wctomb()	stdlib.h	492	Converts a wide character to multibyte characters.
wctrans()	wctype.h	493	Gets a handle for character mapping.
wctype()	wchar.h	495	Obtains a handle for character property classification.
wcwidth()	wchar.h	497	Determines the display width of a wide character.

## Data Areas

Function	Header File	Page	Description
QXXCHGDA()	xxdtaa.h	247	Changes the data area.
QXXRTVDA()	xxdtaa.h	252	Retrieves a copy of the data area specified by dtaname.

## Message Catalogs

Function	Header File	Page	Description
catclose()	nl_types.h	57	Closes a message catalog.
catgets()	nl_types.h	58	Reads a message from an opened message catalog.
catopen()	nl_types.h	59	Opens a message catalog.

## Regular Expression

Function	Header File	Page	Description
regcomp()	regex.h	267	Compiles a regular expression.
regerror()	regex.h	269	Returns error message for regular expression.
regexec()	regex.h	271	Executes a compiled regular expression.
regfree()	regex.h	273	Frees memory for regular expression.

---

## abort() — Stop a Program

### Format

```
#include <stdlib.h>
void abort(void);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `abort()` function causes an abnormal end of the program and returns control to the host environment. Like the `exit()` function, the `abort()` function deletes buffers and closes open files before ending the program.

Calls to the `abort()` function raise the `SIGABRT` signal. The `abort()` function will not result in the ending of the program if `SIGABRT` is caught by a signal handler, and the signal handler does not return.

**Note:** When compiled with `SYSIFCOPT(*ASYNCSIGNAL)`, the `abort()` function cannot be called in a signal handler.

### Return Value

There is no return value.

### Example that uses `abort()`

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("mylib/myfile", "r")) == NULL)
    {
        perror("Could not open data file");
        abort();
    }
}
```

### Related Information

- “`exit()` — End Program” on page 88
- “`signal()` — Handle Interrupt Signals” on page 346
- “`<stdlib.h>`” on page 17
- See the `signal()` API in the APIs topic in the i5/OS Information Center.

---

## **abs()** — Calculate Integer Absolute Value

### Format

```
#include <stdlib.h>
int abs(int n);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

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

### Return Value

There is no error return value. The result is undefined when the absolute value of the argument cannot be represented as an integer. The value of the minimum allowable integer is defined by `INT_MIN` in the `<limits.h>` include file.

### Example that uses abs()

This example calculates the absolute value of an integer  $x$  and assigns it to  $y$ .

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

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

    y = abs(x);

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

    /***** Output *****/
    The absolute value of x is 4.
    *****/
}
```

### Related Information

- “fabs() — Calculate Floating-Point Absolute Value” on page 90
- “labs() — llabs() — Calculate Absolute Value of Long and Long Long Integer” on page 177
- “<limits.h>” on page 7
- “<stdlib.h>” on page 17

---

## acos() — Calculate Arccosine

### Format

```
#include <math.h>
double acos(double x);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `acos()` function calculates the arccosine of  $x$ , expressed in radians, in the range 0 to  $\Pi$ .

### Return Value

The `acos()` function returns the arccosine of  $x$ . The value of  $x$  must be between -1 and 1 inclusive. If  $x$  is less than -1 or greater than 1, `acos()` sets `errno` to `EDOM` and returns 0.

### Example that uses acos()

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: %lf too large for acos\n", x );
else if ( x < MIN )
    printf( "Error: %lf too small for acos\n", x );
else {
    y = acos( x );
    printf( "acos( %lf ) = %lf\n", x, y );
}
}

/***** Expected output if 0.4 is entered: *****/

Enter x
acos( 0.400000 ) = 1.159279
*/

```

### Related Information

- “asin() — Calculate Arcsine” on page 42
- “atan() – atan2() — Calculate Arctangent” on page 44
- “cos() — Calculate Cosine” on page 64
- “cosh() — Calculate Hyperbolic Cosine” on page 65
- “sin() — Calculate Sine” on page 348
- “sinh() — Calculate Hyperbolic Sine” on page 349
- “tan() — Calculate Tangent” on page 409
- “tanh() — Calculate Hyperbolic Tangent” on page 410
- “<math.h>” on page 8

---

## asctime() — Convert Time to Character String

### Format

```

#include <time.h>
char *asctime(const struct tm *time);

```

**Language Level:** ANSI

**Threadsafe:** No. Use `asctime_r()` instead.

### Description

The `asctime()` function converts time, stored as a structure pointed to by *time*, to a character string. You can obtain the *time* value from a call to the `gmtime()`, `gmtime64()`, `localtime()`, or `localtime64()` function.

The string result that `asctime()` produces contains exactly 26 characters and has the format:

```

"% .3s % .3s%3d % .2d:% .2d:% .2d %d\n"

```

The following are examples of the string returned:

```

Sat Jul 16 02:03:55 1994\n\0
or
Sat Jul 16 2:03:55 1994\n\0

```

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 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 time and date functions begin at 00:00:00 Universal Time, January 1, 1970.

### Return Value

The `asctime()` function returns a pointer to the resulting character string. If the function is unsuccessful, it returns NULL.

**Note:** The `asctime()`, `ctime()` functions, and other time functions can use a common, statically allocated buffer to hold the return string. Each call to one of these functions might destroy the result of the previous call. The `asctime_r()`, `ctime_r()`, `gmtime_r()`, and `localtime_r()` functions do not use a common, statically allocated buffer to hold the return string. These functions can be used in place of the `asctime()`, `ctime()`, `gmtime()`, and `localtime()` functions if reentrancy is desired.

### Example that uses `asctime()`

This example polls the system clock and prints a message that gives 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);
    /* Convert it to the structure tm */
    newtime = localtime(&ltime);

    /* Print the local time as a string */
    printf("The current date and time are %s",
           asctime(newtime));
}

/***** Output should be similar to: *****/
The current date and time are Fri Sep 16 13:29:51 1994
*/
```

### Related Information

- “`asctime_r()` — Convert Time to Character String (Restartable)” on page 41
- “`ctime()` — Convert Time to Character String” on page 71
- | • “`ctime64()` — Convert Time to Character String” on page 73
- | • “`ctime64_r()` — Convert Time to Character String (Restartable)” on page 76
- “`ctime_r()` — Convert Time to Character String (Restartable)” on page 74
- “`gmtime()` — Convert Time” on page 161
- | • “`gmtime64()` — Convert Time” on page 163
- | • “`gmtime64_r()` — Convert Time (Restartable)” on page 167
- “`gmtime_r()` — Convert Time (Restartable)” on page 165
- “`localtime()` — Convert Time” on page 185
- | • “`localtime64()` — Convert Time” on page 187
- | • “`localtime64_r()` — Convert Time (Restartable)” on page 189
- “`localtime_r()` — Convert Time (Restartable)” on page 188

- “mktime() — Convert Local Time” on page 218
- | • “mktime64() — Convert Local Time” on page 220
- “strftime() — Convert Date/Time to String” on page 370
- “time() — Determine Current Time” on page 411
- “printf() — Print Formatted Characters” on page 229
- “setlocale() — Set Locale” on page 339
- | • “time64() — Determine Current Time” on page 412
- “<time.h>” on page 18

---

## asctime\_r() — Convert Time to Character String (Restartable)

### Format

```
#include <time.h>
char *asctime_r(const struct tm *tm, char *buf);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

### Description

This function is the restartable version of the `asctime()` function.

- | The `asctime_r()` function converts time, stored as a structure pointed to by *tm*, to a character string. You
- | can obtain the *tm* value from a call to `gmtime_r()`, `gmtime64_r()`, `localtime_r()`, or `localtime64_r()`.

The string result that `asctime_r()` produces contains exactly 26 characters and has the format:

```
"%.3s %.3s%3d %.2d:%.2d:%.2d %d\n"
```

The following are examples of the string returned:

```
Sat Jul 16 02:03:55 1994\n\0
or
Sat Jul 16  2:03:55 1994\n\0
```

The `asctime_r()` 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 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 time and date functions begin at 00:00:00 Universal Time, January 1, 1970.

### Return Value

The `asctime_r()` function returns a pointer to the resulting character string. If the function is unsuccessful, it returns `NULL`.

### Example that uses `asctime_r()`

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;
    char mybuf[50];

/* Get the time in seconds */
    time(&ltime);
/* Convert it to the structure tm */
    newtime = localtime_r(&ltime());
/* Print the local time as a string */
    printf("The current date and time are %s",
           asctime_r(newtime, mybuf));
}

/***** Output should be similar to *****/
The current date and time are Fri Sep 16 132951 1994
*/

```

### Related Information

- “asctime() — Convert Time to Character String” on page 39
- “ctime() — Convert Time to Character String” on page 71
- | • “ctime64() — Convert Time to Character String” on page 73
- | • “ctime64\_r() — Convert Time to Character String (Restartable)” on page 76
- “ctime\_r() — Convert Time to Character String (Restartable)” on page 74
- “gmtime() — Convert Time” on page 161
- | • “gmtime64() — Convert Time” on page 163
- | • “gmtime64\_r() — Convert Time (Restartable)” on page 167
- “gmtime\_r() — Convert Time (Restartable)” on page 165
- “localtime() — Convert Time” on page 185
- | • “localtime64() — Convert Time” on page 187
- | • “localtime64\_r() — Convert Time (Restartable)” on page 189
- “localtime\_r() — Convert Time (Restartable)” on page 188
- “mktime() — Convert Local Time” on page 218
- | • “mktime64() — Convert Local Time” on page 220
- “strftime() — Convert Date/Time to String” on page 370
- “time() — Determine Current Time” on page 411
- “printf() — Print Formatted Characters” on page 229
- “<time.h>” on page 18

---

## asin() — Calculate Arcsine

### Format

```

#include <math.h>
double asin(double x);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `asin()` function calculates the arcsine of  $x$ , in the range  $-\pi/2$  to  $\pi/2$  radians.

### Return Value

The `asin()` function returns the arcsine of  $x$ . The value of  $x$  must be between -1 and 1. If  $x$  is less than -1 or greater than 1, the `asin()` function sets `errno` to `EDOM`, and returns a value of 0.

### Example that uses `asin()`

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: %lf too large for asin\n", x );
    else if ( x < MIN )
        printf( "Error: %lf too small for asin\n", x );
    else
    {
        y = asin( x );
        printf( "asin( %lf ) = %lf\n", x, y );
    }
}

/***** Output should be similar to *****/
Enter x
asin( 0.200000 ) = 0.201358
*/
```

### Related Information

- “`acos()` — Calculate Arccosine” on page 38
- “`atan()` – `atan2()` — Calculate Arctangent” on page 44
- “`cos()` — Calculate Cosine” on page 64
- “`cosh()` — Calculate Hyperbolic Cosine” on page 65
- “`sin()` — Calculate Sine” on page 348
- “`sinh()` — Calculate Hyperbolic Sine” on page 349
- “`tan()` — Calculate Tangent” on page 409
- “`tanh()` — Calculate Hyperbolic Tangent” on page 410
- “`<math.h>`” on page 8

---

## **assert() — Verify Condition**

### Format

```
#include <assert.h>
void assert(int expression);
```

**Language Level:** ANSI

**Threadsafe:** No.

## Description

The `assert()` function prints a diagnostic message to `stderr` and aborts the program if *expression* is false (zero). The diagnostic message has the format:

Assertion failed: *expression*, file *filename*, line *line-number*.

The `assert()` function takes no action if the *expression* is true (nonzero).

Use the `assert()` function to identify program logic errors. Choose an *expression* that holds true only if the program is operating as you intend. After you have debugged the program, you can use the special no-debug identifier `NDEBUG` to remove the `assert()` calls from the program. If you define `NDEBUG` to any value with a `#define` directive, the C preprocessor expands all `assert` calls to void expressions. If you use `NDEBUG`, you must define it before you include `<assert.h>` in the program.

## Return Value

There is no return value.

**Note:** The `assert()` function is defined as a macro. Do not use the `#undef` directive with `assert()`.

## Example that uses `assert()`

In this example, the `assert()` function tests *string* for a null string and an empty string, and verifies that *length* is positive before processing these arguments.

```
#include <stdio.h>
#include <assert.h>

void analyze (char *, int);

int main(void)
{
    char *string = "ABC";
    int length = 3;

    analyze(string, length);
    printf("The string %s is not null or empty, "
           "and has length %d \n", string, 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 should be similar to *****/
The string ABC is not null or empty, and has length 3
```

## Related Information

- “`abort()` — Stop a Program” on page 36
- “`<assert.h>`” on page 3

---

## `atan()` – `atan2()` — Calculate Arctangent

### Format

```
#include <math.h>
double atan(double x);
double atan2(double y, double x);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

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

### Return Value

The `atan()` function returns a value in the range  $-\pi/2$  to  $\pi/2$  radians. The `atan2()` function returns a value in the range  $-\pi$  to  $\pi$  radians. If both arguments of the `atan2()` function are zero, the function sets `errno` to `EDOM`, and returns a value of 0.

### Example that uses `atan()`

This example calculates arctangents using the `atan()` and `atan2()` functions.

```
#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( %lf ) = %lf/n", c, a);
    printf("atan2( %lf, %lf ) = %lf/n", c, d, b);
}

/***** Output should be similar to *****/
atan( 0.450000 ) = 0.422854
atan2( 0.450000, 0.230000 ) = 1.098299
*****/
```

### Related Information

- “`acos()` — Calculate Arccosine” on page 38
- “`asin()` — Calculate Arcsine” on page 42
- “`cos()` — Calculate Cosine” on page 64
- “`cosh()` — Calculate Hyperbolic Cosine” on page 65
- “`sin()` — Calculate Sine” on page 348
- “`sinh()` — Calculate Hyperbolic Sine” on page 349
- “`tan()` — Calculate Tangent” on page 409
- “`tanh()` — Calculate Hyperbolic Tangent” on page 410
- “`<math.h>`” on page 8

---

## **atexit() — Record Program Ending Function**

### Format

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `atexit()` function records the function, pointed to by *func*, that the system calls at normal program end. For portability, you should use the `atexit()` function to register a maximum of 32 functions. The functions are processed in a last-in, first-out order. The `atexit()` function cannot be called from the OPM default activation group. Most functions can be used with the `atexit` function; however, if the `exit` function is used the `atexit` function will fail.

### Return Value

The `atexit()` function returns 0 if it is successful, and nonzero if it fails.

### Example that uses `atexit()`

This example uses the `atexit()` function to call `goodbye()` at program end.

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

int main(void)
{
    void goodbye(void);
    int rc;

    rc = atexit(goodbye);
    if (rc != 0)
        perror("Error in atexit");
    exit(0);
}

void goodbye(void)
/* This function is called at normal program end */
{
    printf("The function goodbye was called at program end\n");
}

/****** Output should be similar to: *****/

The function goodbye was called at program end
*/
```

### Related Information

- “`exit()` — End Program” on page 88
- “`signal()` — Handle Interrupt Signals” on page 346
- “`<stdlib.h>`” on page 17

---

## atof() — Convert Character String to Float

### Format

```
#include <stdlib.h>
double atof(const char *string);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

## Description

The `atof()` function converts a character string to a double-precision floating-point value.

The input *string* is a sequence of characters that can be interpreted as a numeric value of the specified return type. The function stops reading the input string at the first character that it cannot recognize as part of a number. This character can be the null character that ends the string.

The `atof()` function expects a *string* in the following form:



The white space consists of the same characters for which the `isspace()` function is true, such as spaces and tabs. The `atof()` function ignores leading white-space characters.

For the `atof()` function, *digits* is one or more decimal digits; if no digits appear before the decimal point, at least one digit must appear after the decimal point. The decimal digits can precede an exponent, introduced by the letter `e` or `E`. The exponent is a decimal integer, which might be signed.

The `atof()` function will not fail if a character other than a digit follows an `E` or if `e` is read in as an exponent. For example, `100elf` will be converted to the floating-point value `100.0`. The accuracy is up to 17 significant character digits.

## Return Value

The `atof()` function returns a double value that is produced by interpreting the input characters as a number. The return value is 0 if the function cannot convert the input to a value of that type. In case of overflow, the function sets `errno` to `ERANGE` and returns the value `-HUGE_VAL` or `+HUGE_VAL`.

## Example that uses `atof()`

This example shows how to convert numbers that are stored as strings to numeric values.

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

int main(void)
{
    double x;
    char *s;

    s = "-2309.12E-15";
    x = atof(s);    /* x = -2309.12E-15 */

    printf("x = %.4e\n",x);
}
```

/\*\*\*\*\* Output should be similar to: \*\*\*\*\*/

```
x = -2.3091e-12
*/
```

### Related Information

- “atoi() — Convert Character String to Integer”
- “atol() — atoll() — Convert Character String to Long or Long Long Integer” on page 49
- “strtol() — strtoll() — Convert Character String to Long and Long Long Integer” on page 400
- “strtod() — strtodf() — strtold — Convert Character String to Double, Float, and Long Double” on page 392
- “strtod32() — strtod64() — strtod128() — Convert Character String to Decimal Floating-Point” on page 395
- “<stdlib.h>” on page 17

---

## atoi() — Convert Character String to Integer

### Format

```
#include <stdlib.h>
int atoi(const char *string);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

- Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `atoi()` function converts a character string to an integer value. The input *string* is a sequence of characters that can be interpreted as a numeric value of the specified return type. The function stops reading the input string at the first character that it cannot recognize as part of a number. This character can be the null character that ends the string.

The `atoi()` function does not recognize decimal points or exponents. The string argument for this function has the form:



where *whitespace* consists of the same characters for which the `isspace()` function is true, such as spaces and tabs. The `atoi()` function ignores leading white-space characters. The value *digits* represents one or more decimal digits.

### Return Value

The `atoi()` function returns an `int` value that is produced by interpreting the input characters as a number. The return value is 0 if the function cannot convert the input to a value of that type. The return value is undefined in the case of an overflow.

### Example that uses `atoi()`

This example shows how to convert numbers that are stored as strings to numeric values.

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

int main(void)
{
    int i;
    char *s;

    s = "-9885";
    i = atoi(s);    /* i = -9885 */

    printf("i = %d\n",i);
}

/***** Output should be similar to: *****/

i = -9885
*/
```

### Related Information

- “`atof()` — Convert Character String to Float” on page 46
- “`atol()` — `atoll()` — Convert Character String to Long or Long Long Integer”
- “`strtod()` — `strtof()` — `strtold()` — Convert Character String to Double, Float, and Long Double” on page 392
- “`strtod32()` — `strtod64()` — `strtod128()` — Convert Character String to Decimal Floating-Point” on page 395
- “`strtol()` — `strtoll()` — Convert Character String to Long and Long Long Integer” on page 400
- “`<stdlib.h>`” on page 17

---

## atol() — atoll() — Convert Character String to Long or Long Long Integer

### Format (`atol()`)

```
#include <stdlib.h>
long int atol(const char *string);
```

### Format (`atoll()`)

```
#include <stdlib.h>
long long int atoll(const char *string);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Locale Sensitive:** The behavior of these functions might be affected by the `LC_CTYPE` category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `atol()` function converts a character string to a long value. The `atoll()` function converts a character string to a long long value.

The input *string* is a sequence of characters that can be interpreted as a numeric value of the specified return type. The function stops reading the input string at the first character that it cannot recognize as part of a number. This character can be the null character that ends the string.

The `atol()` and `atoll()` functions do not recognize decimal points or exponents. The *string* argument for this function has the form:



where *whitespace* consists of the same characters for which the `isspace()` function is true, such as spaces and tabs. The `atol()` and `atoll()` functions ignore leading white-space characters. The value *digits* represents one or more decimal digits.

### Return Value

The `atol()` and `atoll()` functions return a long or a long long value that is produced by interpreting the input characters as a number. The return value is 0L if the function cannot convert the input to a value of that type. The return value is undefined in case of overflow.

### Example that uses `atol()`

This example shows how to convert numbers that are stored as strings to numeric values.

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

int main(void)
{
    long l;
    char *s;

    s = "98854 dollars";
    l = atol(s);    /* l = 98854 */

    printf("l = %.ld\n",l);
}

/***** Output should be similar to: *****/

l = 98854
*/
```

### Related Information

- “`atof()` — Convert Character String to Float” on page 46
- “`atoi()` — Convert Character String to Integer” on page 48
- “`strtod()` — `strtof()` — `strtold()` — Convert Character String to Double, Float, and Long Double” on page 392
- “`strtod32()` — `strtod64()` — `strtod128()` — Convert Character String to Decimal Floating-Point” on page 395
- “`strtol()` — `strtoll()` — Convert Character String to Long and Long Long Integer” on page 400
- “`<stdlib.h>`” on page 17

---

## Bessel Functions

### Format

```
#include <math.h>
double j0(double x);
double j1(double x);
```

```
double jn(int n, double x);
double y0(double x);
double y1(double x);
double yn(int n, double x);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

### Description

Bessel functions solve certain types of differential equations. The `j0()`, `j1()`, and `jn()` functions are Bessel functions of the first kind for orders 0, 1, and  $n$ , respectively. The `y0()`, `y1()`, and `yn()` functions are Bessel functions of the second kind for orders 0, 1, and  $n$ , respectively.

The argument  $x$  must be positive. The argument  $n$  should be greater than or equal to zero. If  $n$  is less than zero, it will be a negative exponent.

### Return Value

For `j0()`, `j1()`, `y0()`, or `y1()`, if the absolute value of  $x$  is too large, the function sets `errno` to `ERANGE`, and returns 0. For `y0()`, `y1()`, or `yn()`, if  $x$  is negative, the function sets `errno` to `EDOM` and returns the value `-HUGE_VAL`. For `y0()`, `y1()`, or `yn()`, if  $x$  causes overflow, the function sets `errno` to `ERANGE` and returns the value `-HUGE_VAL`.

### Example that uses Bessel Functions

This example computes  $y$  to be the order 0 Bessel function of the first kind for  $x$ . It also computes  $z$  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("y = %lf\n", y);
    printf("z = %lf\n", z);
}
/***** Output should be similar to: *****/

y = -0.366022
z = -0.087482
*****/
```

### Related Information

- “`erf()` – `erfc()` — Calculate Error Functions” on page 87
- “`gamma()` — Gamma Function” on page 150
- “`<math.h>`” on page 8

---

## bsearch() — Search Arrays

### Format

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
             size_t num, size_t size,
             int (*compare)(const void *key, const void *element));
```

**Language Level:** ANSI

**Threadsafe:** Yes.

## Description

The `bsearch()` function performs a binary search of an array of *num* elements, each of *size* bytes. The array must be sorted in ascending order by the function pointed to by *compare*. The *base* is a pointer to the base of the array to search, and *key* is the value being sought.

The *compare* argument is a pointer to a function you must supply that compares two items and returns a value specifying their relationship. The first item in the argument list of the `compare()` function is the pointer to the value of the item that is being searched for. The second item in the argument list of the `compare()` function is a pointer to the array *element* being compared with the *key*. The `compare()` function must compare the *key* value with the array element and then return one of the following values:

Value	Meaning
Less than 0	<i>key</i> less than <i>element</i>
0	<i>key</i> identical to <i>element</i>
Greater than 0	<i>key</i> greater than <i>element</i>

## Return Value

The `bsearch()` function returns a pointer to *key* in the array to which *base* points. If two keys are equal, the element that *key* will point to is unspecified. If the `bsearch()` function cannot find the *key*, it returns NULL.

## Example that uses `bsearch()`

This example performs a binary search on the `argv` array of pointers to the program parameters 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 `compare1()` and `compare2()` functions compare the values pointed to by *arg1* and *arg2* and return the result to the `bsearch()` function.

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

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

main(int argc, char *argv[])
{
    char **result;      /* This program performs a binary      */
    char *key = "PATH"; /* search on the argv array of pointers */
    int i;             /* to the program parameters. It first */
                    /* removes the program name from argv  */
                    /* then sorts the array alphabetically */
    argv++;           /* before calling bsearch.             */
    argc--;

    qsort((char *)argv, argc, sizeof(char *), compare1);

    result = (char**)bsearch(&key, (char *)argv, argc, sizeof(char *), compare2);
    if (result != NULL) {
```

```

        printf("result =<%s>\n",*result);
    }
    else printf("result is null\n");
}

/*This function compares the values pointed to by arg1 */
/*and arg2 and returns the result to qsort. arg1 and */
/*arg2 are both pointers to elements of the argv array. */

int compare1(const void *arg1, const void *arg2)
{
    return (strcmp(*(char **)arg1, *(char **)arg2));
}

/*This function compares the values pointed to by arg1 */
/*and arg2 and returns the result to bsearch */
/*arg1 is a pointer to the key value, arg2 points to */
/*the element of argv that is being compared to the key */
/*value. */

int compare2(const void *arg1, const void *arg2)
{
    return (strcmp(*(char **)arg1, *(char **)arg2));
}
/***** Output should be similar to: *****/

result = <PATH>

***** When the input on the i5/OS command line is *****

CALL BSEARCH PARM(WHERE IS PATH IN THIS PHRASE?')

*/

```

### Related Information

- “qsort() — Sort Array” on page 245
- “<stdlib.h>” on page 17

---

## btowc() — Convert Single Byte to Wide Character

### Format

```

#include <stdio.h>
#include <wchar.h>
wint_t btowc(int c);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. The behavior might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `btowc()` function converts the single byte value `c` to the wide-character representation of `c`. If `c` does not constitute a valid (1-byte) multibyte character in the initial shift state, the `btowc()` function returns `WEOF`.

## Return Value

The `btowc()` function returns `WEOF` if `c` has the value `EOF`, or if (unsigned char) `c` does not constitute a valid (1-byte) multibyte character in the initial shift state. Otherwise, it returns the wide-character representation of that character.

If a conversion error occurs, `errno` might be set to `ECONVERT`.

## Example that uses `btowc()`

This example scans various types of data.

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <local.h>

#define UPPER_LIMIT 0xFF

int main(void)
{
    int wc;
    int ch;
    if (NULL == setlocale(LC_ALL, "/QSYS.LIB/EN_US.LOCALE")) {
        printf("Locale could not be loaded\n");
        exit(1);
    }
    for (ch = 0; ch <= UPPER_LIMIT; ++ch) {
        wc = btowc(ch);
        if (wc==WEOF) {
            printf("%#04x is not a one-byte multibyte character\n", ch);
        } else {
            printf("%#04x has wide character representation: %#06x\n", ch, wc);
        }
    }
    wc = btowc(EOF);
    if (wc==WEOF) {
        printf("The character is EOF.\n", ch);
    } else {
        printf("EOF has wide character representation: %#06x\n", wc);
    }
    return 0;
}
/*****
    If the locale is bound to SBCS, the output should be similar to:
    0000 has wide character representation: 000000
    0x01 has wide character representation: 0x0001
    ...
    0xfe has wide character representation: 0x00fe
    0xff has wide character representation: 0x00ff
    The character is EOF.
*****/
```

## Related Information

- “`mblen()` — Determine Length of a Multibyte Character” on page 197
- “`mbtowc()` — Convert Multibyte Character to a Wide Character” on page 211
- “`mbrtowc()` — Convert a Multibyte Character to a Wide Character (Restartable)” on page 201

- “mbsrtowcs() — Convert a Multibyte String to a Wide Character String (Restartable)” on page 206
- “setlocale() — Set Locale” on page 339
- “wcrctomb() — Convert a Wide Character to a Multibyte Character (Restartable)” on page 446
- “wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)” on page 473
- “<stdio.h>” on page 15
- “<wchar.h>” on page 18

---

## **`_C_Get_Ssn_Handle()` — Handle to C Session**

### **Format**

```
#include <stdio.h>
```

```
_SSN_HANDLE_T _C_Get_Ssn_Handle (void)
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

### **Description**

Returns a handle to the C session for use with Dynamic Screen Manager (DSM) APIs.

### **Return Value**

The `_C_Get_Ssn_Handle()` function returns a handle to the C session. If an error occurs, `_SSN_HANDLE_T` is set to zero. See the APIs topic in the iInformation Center for more information about using the `_C_Get_Ssn_Handle()` function with DSM APIs.

---

## **`calloc()` — Reserve and Initialize Storage**

### **Format**

```
#include <stdlib.h>
void *calloc(size_t num, size_t size);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### **Description**

The `calloc()` function 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.

### **Return Value**

The `calloc()` function returns a pointer to the reserved space. The storage space to which the return value points is suitably aligned for storage of any type of object. To get a pointer to a type, use a type cast on the return value. The return value is NULL if there is not enough storage, or if *num* or *size* is 0.

### **Notes:**

1. All heap storage is associated with the activation group of the calling routine. As such, storage should be allocated and deallocated within the same activation group. You cannot allocate heap storage within one activation group and deallocate that storage from a different activation group. For more information about activation groups, see the *ILE Concepts* manual.

2. To use **Teraspace** storage instead of heap storage without changing the C source code, specify the `TERASPACE(*YES *TSIFC)` parameter on the compiler command. This maps the `calloc()` library function to `_C_TS_calloc()`, its Teraspace storage counterpart. The maximum amount of Teraspace storage that can be allocated by each call to `_C_TS_calloc()` is 2GB - 224, or 2147483424 bytes.

For more information about Teraspace, see the *ILE Concepts* manual.

3. If the `_C_Quickpool_Init()` function has been called in the current activation group, then the storage is retrieved using Quick Pool memory management. See `_C_Quickpool_Init()` for more information.

### Example that uses `calloc()`

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.

```
#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 )
    {
        for ( i = 0; i < num; ++i )           /* put values in arr */
            *index++ = i;                   /* using pointer no */

        for ( i = 0; i < num; ++i )           /* print the array out */
            printf( "array[%i ] = %i\n", i, array[i] );
    }
    else
    { /* out of storage */
        perror( "Out of storage" );
        abort();
    }
}
/***** Output should be similar to: *****/

Enter the size of the array
array[ 0 ] = 0
array[ 1 ] = 1
array[ 2 ] = 2
*/
```

### Related Information

- “`_C_Quickpool_Debug()` — Modify Quick Pool Memory Management Characteristics” on page 66
- “`_C_Quickpool_Init()` — Initialize Quick Pool Memory Management” on page 68
- “`_C_Quickpool_Report()` — Generate Quick Pool Memory Management Report” on page 70
- “`free()` — Release Storage Blocks” on page 128
- “`malloc()` — Reserve Storage Block” on page 195

- | • “realloc() — Change Reserved Storage Block Size” on page 264
- | • “<stdlib.h>” on page 17

---

## catclose() — Close Message Catalog

### Format

```
#include <n1_types.h>
int catclose (n1_catd catd);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

| **Locale Sensitive:** This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command.

| **Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified on the compilation command.

### Description

The `catclose()` function closes the previously opened message catalog that is identified by *catd*.

### Return Value

If the close is performed successfully, 0 is returned. Otherwise, -1 is returned indicating failure, which might happen if *catd* is not a valid message catalog descriptor.

The value of `errno` can be set to:

#### EBADF

The catalog descriptor is not valid.

#### EINTR

The function was interrupted by a signal.

**Example that uses `catclose()`**

```

#include <stdio.h>
#include <nl_types.h>
#include <locale.h>

/* Name of the message catalog is "/qsys.lib/mylib.lib/msgs.usrsrc" */

int main(void) {

    nl_catd msg_file;
    char * my_msg;
    char * my_locale;

    setlocale(LC_ALL, NULL);
    msg_file = catopen("/qsys.lib/mylib.lib/msgs.usrsrc", 0);

    if (msg_file != CATD_ERR) {

        my_msg = catgets(msg_file, 1, 2, "oops");

        printf("%s\n", my_msg);

        catclose(msg_file);
    }
}

```

### Related Information

- “catopen() — Open Message Catalog” on page 59
- “catgets() — Retrieve a Message from a Message Catalog”

---

## catgets() — Retrieve a Message from a Message Catalog

### Format

```

#include <nl_types.h>
char *catgets(nl_catd catd, int set_id, int msg_id, char *s);

```

**Language Level:** XPG4

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the  
| current locale. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation  
| command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified  
| on the compilation command.

### Description

The catgets() function retrieves message *msg\_id*, in set *set\_id* from the message catalog that is identified by *catd*. *catd* is a message catalog descriptor that is returned by a previous call to catopen(). The *s* argument points to a default message which will be returned by catgets() if the identified message cannot be retrieved.

### Return Value

| If the message is retrieved successfully, then catgets() returns a pointer to the message string that is  
| contained in the message catalog. The CCSID of the retrieved message is determined by the flags  
| specified in the *oflag* parameter on the previous call to the catopen() function, when the message catalog  
| file was opened.

- | • If the NL\_CAT\_JOB\_MODE flag was specified, then the retrieved message is in the CCSID of the job.
  - | • If the NL\_CAT\_CTYPE\_MODE flag was specified, then the retrieved message is in the CCSID of the LC\_CTYPE category of the current locale.
  - | • If neither flag was specified, the CCSID of the retrieved message matches the CCSID of the message catalog file.
- | If the message is retrieved unsuccessfully, then a pointer to the default string *s* is returned.

The value of `errno` can be set to the following:

**EBADF**

The catalog descriptor is not valid.

**ECONVERT**

A conversion error occurred.

**EINTR**

The function was interrupted by a signal.

**Example that uses `catgets()`**

```
#include <stdio.h>
#include <nl_types.h>
#include <locale.h>

/* Name of the message catalog is "/qsys.lib/mylib.lib/msgs.usrsrc" */

int main(void) {
    nl_catd msg_file;
    char * my_msg;
    char * my_locale;

    setlocale(LC_ALL, NULL);
    msg_file = catopen("/qsys.lib/mylib.lib/msgs.usrsrc", 0);

    if (msg_file != CATD_ERR) {
        my_msg = catgets(msg_file, 1, 2, "oops");

        printf("%s\n", my_msg);

        catclose(msg_file);
    }
}
```

**Related Information**

- “`catclose()` — Close Message Catalog” on page 57
- “`catopen()` — Open Message Catalog”

**catopen() — Open Message Catalog**

**Format**

```
#include <nl_types.h>
nl_catd catopen(const char *name, int oflag);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_MESSAGES category of the  
| current locale. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation  
| command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified  
| on the compilation command.

## Description

The `catopen()` function opens a message catalog, which must be done before a message can be retrieved. The `NLSPATH` environment variable and the `LC_MESSAGES` category are used to find the specified message catalog if no slash (/) characters are found in the name. If the name contains one or more slash (/) characters, then the name is interpreted as a path name of the catalog to open.

| If there is no `NLSPATH` environment variable, or if a message catalog cannot be found in the path  
| specified by `NLSPATH`, then a default path is used. The default path might be affected by the setting of  
| the `LANG` environment variable; if the `NL_CAT_LOCALE` flag is set in the *oflag* parameter or if the  
| `LANG` environment variable is not set, the default path might be affected by the `LC_MESSAGES` locale  
| category.

| Three values can be specified for the *oflag* parameter: `NL_CAT_LOCALE`, `NL_CAT_JOB_MODE`, and  
| `NL_CAT_CTYPE_MODE`. `NL_CAT_JOB_MODE` and `NL_CAT_CTYPE_MODE` are mutually exclusive. If  
| the `NL_CAT_JOB_MODE` and `NL_CAT_CTYPE_MODE` flags are both set in the *oflag* parameter, the  
| `catopen()` function will fail with a return value of `CATD_ERR`.

| If you want the catalog messages to be converted to the job CCSID before they are returned by the  
| `catgets()` function, set the parameter to `NL_CAT_JOB_MODE`. If you want the catalog messages to be  
| converted to the `LC_CTYPE` CCSID before they are returned by `catgets()`, set the parameter to  
| `NL_CAT_CTYPE_MODE`. If you do not set the parameter to `NL_CAT_JOB_MODE` or  
| `NL_CAT_CTYPE_MODE`, the messages are returned without conversion and are in the CCSID of the  
| message file.

The message catalog descriptor will remain valid until it is closed by a call to `catclose()`. If the `LC_MESSAGES` locale category is changed, it might invalidate existing open message catalogs.

| **Note:** The name of the message catalog must be a valid integrated file system file name.

## Return Value

If the message catalog is opened successfully, then a valid catalog descriptor is returned. If `catopen()` is unsuccessful, then it returns `CATD_ERR ((nl_catd)-1)`.

The `catopen()` function might fail under the following conditions, and the value of `errno` can be set to:

### EACCES

Insufficient authority to read the message catalog specified, or to search the component of the path prefix of the message catalog specified.

### ECONVERT

A conversion error occurred.

### EMFILE

`NL_MAXOPEN` message catalogs are currently open.

### ENAMETOOLONG

The length of the path name of the message catalog exceeds `PATH_MAX`, or a path name component is longer than `NAME_MAX`.

## ENFILE

Too many files are currently open in the system.

## ENOENT

The message catalog does not exist, or the name argument points to an empty string.

### Example that uses `catopen()`

```
#include <stdio.h>
#include <nl_types.h>
#include <locale.h>

/* Name of the message catalog is "/qsys.lib/mylib.lib/messages.usrsrc" */

int main(void) {
    nl_catd msg_file;
    char * my_msg;
    char * my_locale;

    setlocale(LC_ALL, NULL);
    msg_file = catopen("/qsys.lib/mylib.lib/messages.usrsrc", 0);

    if (msg_file != CATD_ERR) {
        my_msg = catgets(msg_file, 1, 2, "oops");

        printf("%s\n", my_msg);

        catclose(msg_file);
    }
}
```

### Related Information

- “`catclose()` — Close Message Catalog” on page 57
- “`catgets()` — Retrieve a Message from a Message Catalog” on page 58

---

## **ceil()** — Find Integer $\geq$ Argument

### Format

```
#include <math.h>
double ceil(double x);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `ceil()` function computes the smallest integer that is greater than or equal to  $x$ .

### Return Value

The `ceil()` function returns the integer as a double value.

### Example that uses `ceil()`

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 = %.2f ; z = %.2f\n", y, z);
}
/***** Output should be similar to: *****/

y = 2.00 ; z = -1.00
*****/

```

### Related Information

- “floor() — Find Integer <=Argument” on page 107
- “fmod() — Calculate Floating-Point Remainder” on page 108
- “<math.h>” on page 8

---

## clearerr() — Reset Error Indicators

### Format

```

#include <stdio.h>
void clearerr (FILE *stream);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `clearerr()` function resets the error indicator and end-of-file indicator for the specified *stream*. Once set, the indicators for a specified stream remain set until your program calls the `clearerr()` function or the `rewind()` function. The `fseek()` function also clears the end-of-file indicator. The ILE C/C++ runtime environment does not automatically clear error or end of file indicators.

### Return Value

There is no return value.

The value of `errno` can be set to:

#### Value Meaning

##### EBADF

The file pointer or descriptor is not valid.

##### ENOTOPEN

The file is not open.

##### ESTDIN

`stdin` cannot be opened.

##### EIOERROR

A non-recoverable I/O error occurred.

##### EIORECERR

A recoverable I/O error occurred.

### Example that uses `clearerr()`

This example reads a data stream, and then checks that a read error has not occurred.

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

FILE *stream;
int c;

int main(void)
{
    if ((stream = fopen("mylib/myfile", "r")) != NULL)
    {
        if ((c=getc(stream)) == EOF)
        {
            if (ferror(stream))
            {
                perror("Read error");
                clearerr(stream);
            }
        }
    }
    else
        exit(0);
}
```

### Related Information

- “`feof()` — Test End-of-File Indicator” on page 95
- “`ferror()` — Test for Read/Write Errors” on page 95
- “`fseek()` — `fseeko()` — Reposition File Position” on page 134
- “`perror()` — Print Error Message” on page 227
- “`rewind()` — Adjust Current File Position” on page 276
- “`strerror()` — Set Pointer to Runtime Error Message” on page 367
- “`<stdio.h>`” on page 15

---

## clock() — Determine Processor Time

### Format

```
#include <time.h>
clock_t clock(void);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `clock()` function returns an approximation of the processor time used by the program since the beginning of an implementation-defined time-period that is related to the process invocation. To obtain the time in seconds, divide the value that is returned by `clock()` by the value of the macro `CLOCKS_PER_SEC`.

### Return Value

If the value of the processor time is not available or cannot be represented, the `clock()` function returns the value `(clock_t)-1`.

To measure the time spent in a program, call `clock()` at the start of the program, and subtract its return value from the value returned by subsequent calls to `clock()`. On other platforms, you can not always rely on the `clock()` function because calls to the `system()` function might reset the clock.

### Example that uses `clock()`

This example prints the time that has elapsed since the program was called.

```
#include <time.h>
#include <stdio.h>

double time1, timedif;      /* use doubles to show small values */

int main(void)
{
    int i;

    time1 = (double) clock();      /* get initial time */
    time1 = time1 / CLOCKS_PER_SEC; /* in seconds */

    /* running the FOR loop 10000 times */
    for (i=0; i<10000; i++);

    /* call clock a second time */
    timedif = ( (double) clock() / CLOCKS_PER_SEC) - time1;
    printf("The elapsed time is %lf seconds\n", timedif);
}
```

### Related Information

- “`difftime()` — Compute Time Difference” on page 82
- “`difftime64()` — Compute Time Difference” on page 84
- “`time()` — Determine Current Time” on page 411
- “`time64()` — Determine Current Time” on page 412
- “`<time.h>`” on page 18

---

## `cos()` — Calculate Cosine

### Format

```
#include <math.h>
double cos(double x);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `cos()` function calculates the cosine of  $x$ . The value  $x$  is expressed in radians. If  $x$  is too large, a partial loss of significance in the result might occur.

### Return Value

The `cos()` function returns the cosine of  $x$ . The value of `errno` can be set to either `EDOM` or `ERANGE`.

### Example that uses `cos()`

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 should be similar to: *****/

cos( 7.200000 ) = 0.608351
*/

```

### Related Information

- “acos() — Calculate Arccosine” on page 38
- “cosh() — Calculate Hyperbolic Cosine”
- “sin() — Calculate Sine” on page 348
- “sinh() — Calculate Hyperbolic Sine” on page 349
- “tan() — Calculate Tangent” on page 409
- “tanh() — Calculate Hyperbolic Tangent” on page 410
- “<math.h>” on page 8

---

## cosh() — Calculate Hyperbolic Cosine

### Format

```

#include <math.h>
double cosh(double x);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `cosh()` function calculates the hyperbolic cosine of  $x$ . The value  $x$  is expressed in radians.

### Return Value

The `cosh()` function returns the hyperbolic cosine of  $x$ . If the result is too large, `cosh()` returns the value `HUGE_VAL` and sets `errno` to `ERANGE`.

### Example that uses `cosh()`

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 should be similar to: *****/

cosh( 7.200000 ) = 669.715755
*/

```

### Related Information

- “acos() — Calculate Arccosine” on page 38
- “cos() — Calculate Cosine” on page 64
- “sin() — Calculate Sine” on page 348
- “sinh() — Calculate Hyperbolic Sine” on page 349
- “tan() — Calculate Tangent” on page 409
- “tanh() — Calculate Hyperbolic Tangent” on page 410
- “<math.h>” on page 8

---

## **\_C\_Quickpool\_Debug() — Modify Quick Pool Memory Management Characteristics**

### Format

```

#include <stdlib.h>
_C_Quickpool_Debug_T _C_Quickpool_Debug(_C_Quickpool_Debug_T *newval);

```

**Language Level:** Extended

**Threadsafe:** Yes.

### Description

The `_C_Quickpool_Debug()` function modifies Quick Pool memory management characteristics.

The parameters for `_C_Quickpool_Debug()` are as follows:

#### **newval**

A pointer to a `_C_Quickpool_Debug_T` structure. The structure contains the following fields:

**flags** An unsigned integer value that indicates the characteristics to be modified. The flags field can contain the following values (which can be used in any combination):

#### **`__C_INIT_MALLOC`**

Initializes all allocated storage to a specified value. The value is passed in the variable argument list.

#### **`__C_INIT_FREE`**

Initializes all freed storage to a specified value. The value is passed in the variable argument list.

## **\_C\_COLLECT\_STATS**

Collects statistics on the Quick Pool memory management algorithm to use `_C_Quickpool_Report` functions.

## **malloc\_val**

A 1-byte unsigned character value that is used to initialize allocated memory. This field is in use only when the `_C_INIT_MALLOC` flag is specified.

## **free\_val**

A 1-byte unsigned character value that is used to initialize freed memory. This field is in use only when the `_C_INIT_FREE` flag is specified.

## **Return Value**

The return value is a structure that contains the `_C_Quickpool_Debug` values before the changes requested by the current function call are made. This value can be used to restore the `_C_Quickpool_Debug` values to a prior state with a later call.

## **Example that uses `_C_Quickpool_Debug()`**

The following example uses `_C_Quickpool_Debug` with the `_C_INIT_MALLOC` and `_C_INIT_FREE` flags to initialize memory on the `malloc` and `free` functions.

```
#include <stdlib.h>
#include <stdio.h>
int main(void) {
    char *p;
    char *mtest = "AAAAAAAAA";
    char *ftest = "BBBBBBBBB";
    unsigned int cell_sizes[2] = { 16, 64 };
    unsigned int cells_per_extent[2] = { 16, 16 };
    _C_Quickpool_Debug_T dbgVals = { _C_INIT_MALLOC | _C_INIT_FREE, 'A', 'B' };

    if (_C_Quickpool_Init(2, cell_sizes, cells_per_extent) {
        printf("Error initializing Quick Pool memory management.\n");
        return -1;
    }

    _C_Quickpool_Debug(dbgVals);

    if ((p = malloc(10)) == NULL) {
        printf("Error during malloc.\n");
        return -2;
    }
    if (memcmp(p, mtest, 10)) {
        printf("malloc test failed\n");
    }
    free(p);
    if (memcmp(p, ftest, 10)) {
        printf("free test failed\n");
    }
    printf("Test successful!\n");
    return 0;
}
/*****Output should be similar to:*****/
Test successful!
*****/
```

## **Related Information**

- “`_C_Quickpool_Init()` — Initialize Quick Pool Memory Management” on page 68
- “`_C_Quickpool_Report()` — Generate Quick Pool Memory Management Report” on page 70
- “`<stdlib.h>`” on page 17

---

## **`_C_Quickpool_Init()` — Initialize Quick Pool Memory Management**

### **Format**

```
#include <stdlib.h>
int _C_Quickpool_Init(unsigned int numpools, unsigned int *cell_sizes, unsigned int *num_cells);
```

**Language Level:** Extended

**Threadsafe:** Yes.

### **Description**

When the `_C_Quickpool_Init()` function is called, all subsequent calls to memory management functions (`malloc`, `calloc`, `realloc`, and `free`) in the same activation group will use the Quick Pool memory management algorithm. This algorithm can provide improved performance in some cases.

Quick Pool memory management breaks memory up into a series of pools. Each pool is broken up into a number of cells with identical sizes. The number of pools, the size of cells in each pool, and the number of cells in each pool extent is set using the `_C_Quickpool_Init()` function.

Suppose that a user wants to define four pools, each of which contains 64 cells. The first pool will have 16-byte cells; the second pool will have 256-byte cells; the third pool will have 1024-byte cells; and the fourth pool will have 2048-byte cells. When the user submits the request for storage, the memory management algorithm assigns the request to a pool first. The algorithm compares the size of storage in the request with the size of the cells in a given pool.

In this example, the first pool satisfies requests between 1 and 16 bytes in size; the second pool satisfies requests between 17 and 256 bytes in size; the third pool satisfies requests between 257 and 1024 bytes in size, and the fourth pool satisfies requests between 1025 and 2048 bytes in size. Any requests larger than the largest cell size are allocated through the old memory management algorithm.

After the pool has been assigned, the free queue for the pool is examined. Each pool has a free queue that contains cells that have been freed and have not yet been reallocated. If there is a cell on the free queue, the cell is removed from the free queue and returned; otherwise, the cell is retrieved from the current extent for the pool. An extent is a collection of cells that are allocated as one block. Initially, a pool has no extents.

When the first request comes in for a pool, an extent is allocated for the pool and the request is satisfied from that extent. Later requests are also satisfied by that extent until the extent is exhausted. When an extent is exhausted, a new extent is allocated for the pool. If the algorithm fails to retrieve a cell from the current extent, it assumes that a memory problem exists. An attempt will be made to allocate the storage using the old memory management algorithm. If the attempt is not successful, the `NULL` value is returned.

### **numpools**

The number of pools to use in the Quick Pool memory management algorithm. This parameter can have a value between 1 and 512.

### **cell\_sizes**

An array of unsigned integer values. The number of entries in the array is equal to the number specified on the `numpools` parameter. Each entry specifies the number of bytes in a cell for a given pool. These values must be multiples of 16 bytes. If a value is specified that is not a multiple of 16 bytes, the cell size is rounded up to the next larger multiple of 16 bytes. The minimum valid value is 16 bytes and the maximum valid value is 4096 bytes.

### **num\_cells**

An array of unsigned integer values. The number of entries in the array is equal to the number

| specified on the numpools parameter. Each entry specifies the number of cells in a single extent  
| for the corresponding pool. The number can be any nonzero number, but the total size of the  
| extent cannot exceed the maximum size of an extent (currently 16 MB).

| Here is the call to `_C_Quickpool_Init()` for the preceding example:

```
| unsigned int cell_sizes[4] = { 16, 256, 1024, 2048 };  
| unsigned int cells_per_extent[4] = { 64, 64, 64, 64 };  
| rc = _C_Quickpool_Init(4, /* number of pools */  
| /* cell sizes for each pool */  
| /* extent sizes for each pool */  
| cell_sizes,  
| cells_per_extent);
```

## | Return Value

| The follow list shows the return values for the `_C_Quickpool_Init` function:

- | 0 Success
- | -1 `_C_Quickpool_Init` has already been called in this activation group.
- | -2 Error allocating storage for control structures.
- | -3 An invalid number of pools was specified.
- | -4 `_C_Quickpool_Init` was called from an invalid activation group.
- | -5 An unexpected exception occurred when `_C_Quickpool_Init` was running.

## | Example that uses `_C_Quickpool_Init()`

| The following example uses `_C_Quickpool_Init` to enable Quick Pool memory allocation.

```
| #include <stdlib.h>  
| #include <stdio.h>  
| int main(void) {  
|     char *p;  
|     unsigned int cell_sizes[2] = { 16, 64 };  
|     unsigned int cells_per_extent[2] = { 16, 16 };  
|  
|     if (_C_Quickpool_Init(2, cell_sizes, cells_per_extent) {  
|         printf("Error initializing Quick Pool memory management.\n");  
|         return -1;  
|     }  
|     if ((p = malloc(10)) == NULL) {  
|         printf("Error during malloc.\n");  
|         return -2;  
|     }  
|     free(p);  
|     printf("Test successful!\n");  
|     return 0;  
| }  
| /******Output should be similar to:*****  
| Test successful!  
| *****/
```

## | Related Information

- | • “`_C_Quickpool_Debug()` — Modify Quick Pool Memory Management Characteristics” on page 66
- | • “`_C_Quickpool_Report()` — Generate Quick Pool Memory Management Report” on page 70
- | • “`<stdlib.h>`” on page 17

---

## | **\_C\_Quickpool\_Report() — Generate Quick Pool Memory Management Report**

### | **Format**

```
| #include <stdlib.h>  
| void _C_Quickpool_Report(void);
```

| **Language Level:** Extended

| **Threadsafe:** Yes.

### | **Description**

| The `_C_Quickpool_Report` function generates a spooled file that contains a snapshot of the memory used by the Quick Pool memory management algorithm in the current activation group. If the `_C_Quickpool_Init()` function has not been called in the current activation group yet or if the `_C_COLLECT_STATS` flag is not used on a call to `_C_Quickpool_Debug()`, the report will be a message that indicates no data is collected.

| If `_C_Quickpool_Init()` has been called and the `_C_COLLECT_STATS` flag is set using the `_C_Quickpool_Debug()` function, the report that is generated indicates the number of allocation attempts for each 16 bytes of memory since the time `_C_Quickpool_Debug()` was called. In addition, the report indicates the maximum number of outstanding allocations (peak allocations) that is reached for each pool. If no storage requests are made for a given range of memory, that range of memory will not be included in the report. No output is generated for allocations larger than the maximum cell size (4096 bytes).

### | **Return Value**

| There is no return value for the function.

### | **Example that uses `_C_Quickpool_Report()`**

| The following example uses `_C_Quickpool_Init` to enable Quick Pool memory allocation. It uses the `_C_COLLECT_STATS` flags to collect information. The collected information is printed using `_C_Quickpool_Report`.

```

| #include <stdlib.h>
| #include <stdio.h>
| int main(void) {
|     char *p;
|     int i;
|     unsigned int cell_sizes[2] = { 16, 64 };
|     unsigned int cells_per_extent[2] = { 16, 16 };
|     _C_Quickpool_Debug_T dbgVals = { _C_COLLECT_STATS, 'A', 'B' };
|
|     if (_C_Quickpool_Init(2, cell_sizes, cells_per_extent) {
|         printf("Error initializing Quick Pool memory management.\n");
|         return -1;
|     }
|
|     _C_Quickpool_Debug(dbgVals);
|
|     for (i = 1; i <= 64; i++) {
|         p = malloc(i);
|         free(p);
|     }
|     p = malloc(128);
|     free(p);
|     _C_Quickpool_Report();
|     return 0;
| }
|
| /*****Spooled File Output should be similar to:*****/
| Pool 1 (16 bytes, 1 peak allocations):
| 1-16 bytes: 16 allocations
| Pool 2 (64 bytes, 1 peak allocations):
| 17-32 bytes: 16 allocations
| 33-48 bytes: 16 allocations
| 49-64 bytes: 16 allocations
| Remaining allocations smaller than the largest cell size (4096 bytes):
| 113-128 bytes: 1 allocations
| *****/

```

### Related Information

- | • “\_C\_Quickpool\_Debug() — Modify Quick Pool Memory Management Characteristics” on page 66
- | • “\_C\_Quickpool\_Init() — Initialize Quick Pool Memory Management” on page 68
- | • “<stdlib.h>” on page 17

---

## ctime() — Convert Time to Character String

### Format

```

#include <time.h>
char *ctime(const time_t *time);

```

**Language Level:** ANSI

**Threadsafe:** No. Use `ctime_r()` instead.

- | **Locale Sensitive:** The behavior of this function might be affected by the LC\_TOD category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `ctime()` function converts the time value pointed to by *time* to local time in the form of a character string. A time value is usually obtained by a call to the `time()` function.

The string result that is produced 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
```

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 with a zero. The new-line character (`\n`) and the null character (`\0`) occupy the last two positions of the string.

## Return Value

The `ctime()` function returns a pointer to the character string result. If the function is unsuccessful, it returns `NULL`. A call to the `ctime()` function is equivalent to:

```
asctime(localtime(&anytime))
```

**Note:** The `asctime()` and `ctime()` functions, and other time functions can use a common, statically allocated buffer to hold the return string. Each call to one of these functions might destroy the result of the previous call. The `asctime_r()`, `ctime_r()`, `gmtime_r()`, and `localtime_r()` functions do not use a common, statically allocated buffer to hold the return string. These functions can be used in place of `asctime()`, `ctime()`, `gmtime()`, and `localtime()` if reentrancy is desired.

## Example that uses `ctime()`

This example polls the system clock using `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));
}
```

## Related Information

- “`asctime()` — Convert Time to Character String” on page 39
- “`asctime_r()` — Convert Time to Character String (Restartable)” on page 41
- “`ctime_r()` — Convert Time to Character String (Restartable)” on page 74
- | • “`ctime64()` — Convert Time to Character String” on page 73
- | • “`ctime64_r()` — Convert Time to Character String (Restartable)” on page 76
- “`gmtime()` — Convert Time” on page 161
- | • “`gmtime64()` — Convert Time” on page 163
- | • “`gmtime64_r()` — Convert Time (Restartable)” on page 167
- “`gmtime_r()` — Convert Time (Restartable)” on page 165
- “`localtime()` — Convert Time” on page 185
- | • “`localtime64()` — Convert Time” on page 187
- | • “`localtime64_r()` — Convert Time (Restartable)” on page 189
- “`localtime_r()` — Convert Time (Restartable)” on page 188
- “`mktime()` — Convert Local Time” on page 218
- | • “`mktime64()` — Convert Local Time” on page 220

- “setlocale() — Set Locale” on page 339
- “strftime() — Convert Date/Time to String” on page 370
- “time() — Determine Current Time” on page 411
- “time64() — Determine Current Time” on page 412
- “printf() — Print Formatted Characters” on page 229
- “<time.h>” on page 18

---

## ctime64() — Convert Time to Character String

### Format

```
#include <time.h>
char *ctime64(const time64_t *time);
```

**Language Level:** ILE C Extension

**Threadsafe:** No. Use `ctime64_r()` instead.

**Locale Sensitive:** The behavior of this function might be affected by the `LC_TOD` category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `ctime64()` function converts the time value pointed to by *time* to local time in the form of a character string. A time value is usually obtained by a call to the `time64()` function.

The string result that is produced by the `ctime64()` function 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
```

The `ctime64()` function uses a 24-hour clock format. The month and day abbreviations used are retrieved from the locale. All fields have a constant width. Dates with only 1 digit are preceded with a zero. The new-line character (`\n`) and the null character (`\0`) occupy the last two positions of the string.

### Return Value

The `ctime64()` function returns a pointer to the character string result. If the function is unsuccessful, it returns `NULL`. A call to the `ctime64()` function is equivalent to:

```
asctime(localtime64(&anytime))
```

**Note:** The `asctime()` and `ctime64()` functions, and other time functions can use a common, statically allocated buffer to hold the return string. Each call to one of these functions might destroy the result of the previous call. The `asctime_r()`, `ctime64_r()`, `gmtime64_r()`, and `localtime64_r()` functions do not use a common, statically allocated buffer to hold the return string. These functions can be used in place of `asctime()`, `ctime64()`, `gmtime64()`, and `localtime64()`, if reentrancy is desired.

### Example that uses ctime64()

This example polls the system clock using `time64()`. It then prints a message that gives the current date and time.

```

| #include <time.h>
| #include <stdio.h>
|
| int main(void)
| {
|     time64_t ltime;
|
|     time64(&ltime);
|
|     printf("the time is %s", ctime64(&ltime));
| }
|

```

### Related Information

- | • “asctime() — Convert Time to Character String” on page 39
- | • “asctime\_r() — Convert Time to Character String (Restartable)” on page 41
- | • “ctime() — Convert Time to Character String” on page 71
- | • “ctime64() — Convert Time to Character String” on page 73
- | • “ctime64\_r() — Convert Time to Character String (Restartable)” on page 76
- | • “gmtime() — Convert Time” on page 161
- | • “gmtime64() — Convert Time” on page 163
- | • “gmtime64\_r() — Convert Time (Restartable)” on page 167
- | • “gmtime\_r() — Convert Time (Restartable)” on page 165
- | • “localtime() — Convert Time” on page 185
- | • “localtime64() — Convert Time” on page 187
- | • “localtime64\_r() — Convert Time (Restartable)” on page 189
- | • “localtime\_r() — Convert Time (Restartable)” on page 188
- | • “mktime() — Convert Local Time” on page 218
- | • “mktime64() — Convert Local Time” on page 220
- | • “setlocale() — Set Locale” on page 339
- | • “strftime() — Convert Date/Time to String” on page 370
- | • “time() — Determine Current Time” on page 411
- | • “time64() — Determine Current Time” on page 412
- | • “printf() — Print Formatted Characters” on page 229
- | • “<time.h>” on page 18

---

## ctime\_r() — Convert Time to Character String (Restartable)

### Format

```

#include <time.h>
char *ctime_r(const time_t *time, char *buf);

```

**Language Level:** XPG4

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_TOD category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

This function is the restartable version of the `ctime()` function.

The `ctime_r()` function converts the time value pointed to by *time* to local time in the form of a character string. A time value is usually obtained by a call to the `time()` function.

The string result that is produced by the `ctime_r()` function 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
```

The `ctime_r()` 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 with a zero. The new-line character (`\n`) and the null character (`\0`) occupy the last two positions of the string.

### Return Value

The `ctime_r()` function returns a pointer to the character string result. If the function is unsuccessful, it returns `NULL`. A call to `ctime_r()` is equivalent to:

```
asctime_r(localtime_r(&anytime, buf2), buf)
```

where `buf` is a pointer to `char`.

### Example that uses `ctime_r()`

This example polls the system clock using `ctime_r()`. It then prints a message giving the current date and time.

```
#include <time.h>
#include <stdio.h>

int main(void)
{
    time_t ltime;
    char buf[50];

    time(&ltime);
    printf("the time is %s", ctime_r(&ltime, buf));
}
```

### Related Information

- “`asctime()` — Convert Time to Character String” on page 39
- “`asctime_r()` — Convert Time to Character String (Restartable)” on page 41
- “`ctime()` — Convert Time to Character String” on page 71
- | • “`ctime64()` — Convert Time to Character String” on page 73
- | • “`ctime64_r()` — Convert Time to Character String (Restartable)” on page 76
- “`gmtime()` — Convert Time” on page 161
- | • “`gmtime64()` — Convert Time” on page 163
- | • “`gmtime64_r()` — Convert Time (Restartable)” on page 167
- “`gmtime_r()` — Convert Time (Restartable)” on page 165
- “`localtime()` — Convert Time” on page 185
- | • “`localtime64()` — Convert Time” on page 187
- | • “`localtime64_r()` — Convert Time (Restartable)” on page 189
- “`localtime_r()` — Convert Time (Restartable)” on page 188

- “mktime() — Convert Local Time” on page 218
- “mktime64() — Convert Local Time” on page 220
- “strftime() — Convert Date/Time to String” on page 370
- “time() — Determine Current Time” on page 411
- “time64() — Determine Current Time” on page 412
- “<time.h>” on page 18

---

## | **ctime64\_r() — Convert Time to Character String (Restartable)**

### | **Format**

```
| #include <time.h>
| char *ctime64_r(const time64_t *time, char *buf);
```

| **Language Level:** ILE C Extension

| **Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_TOD category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### | **Description**

| This function is the restartable version of the `ctime64()` function.

| The `ctime64()` function converts the time value pointed to by *time* to local time in the form of a character string. A *time* value is usually obtained by a call to the `time64()` function.

| The string result that is produced by the `ctime64_r()` function 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
```

| The `ctime64_r()` function uses a 24-hour clock format. The month and day abbreviation used are retrieved from the locale. All fields have a constant width. Dates with only 1 digit are preceded with a zero. The new-line character (`\n`) and the null character (`\0`) occupy the last two positions of the string.

### | **Return Value**

| The `ctime64_r()` function returns a pointer to the character string result. If the function is unsuccessful, it returns `NULL`. A call to the `ctime64_r()` function is equivalent to:

```
| asctime_r(localtime64_r(&anytime, buf2), buf)
```

### | **Example that uses ctime64\_r()**

| This example polls the system clock using `time64()`. It then prints a message, giving the current date and time.

```

| #include <time.h>
| #include <stdio.h>
|
| int main(void)
| {
|     time64_t ltime;
|     char buf[50];
|
|     time64(&ltime);
|     printf("the time is %s", ctime64_r(&ltime, buf));
| }
|

```

### Related Information

- | • “asctime() — Convert Time to Character String” on page 39
- | • “asctime\_r() — Convert Time to Character String (Restartable)” on page 41
- | • “ctime() — Convert Time to Character String” on page 71
- | • “ctime64() — Convert Time to Character String” on page 73
- | • “ctime\_r() — Convert Time to Character String (Restartable)” on page 74
- | • “gmtime() — Convert Time” on page 161
- | • “gmtime64() — Convert Time” on page 163
- | • “gmtime64\_r() — Convert Time (Restartable)” on page 167
- | • “gmtime\_r() — Convert Time (Restartable)” on page 165
- | • “localtime() — Convert Time” on page 185
- | • “localtime64() — Convert Time” on page 187
- | • “localtime64\_r() — Convert Time (Restartable)” on page 189
- | • “localtime\_r() — Convert Time (Restartable)” on page 188
- | • “mktime() — Convert Local Time” on page 218
- | • “mktime64() — Convert Local Time” on page 220
- | • “strftime() — Convert Date/Time to String” on page 370
- | • “time() — Determine Current Time” on page 411
- | • “time64() — Determine Current Time” on page 412
- | • “<time.h>” on page 18

---

## **`_C_TS_malloc_debug()` — Determine amount of teraspace memory used (with optional dumps and verification)**

### Format

```

#include <mallocinfo.h>
int _C_TS_malloc_debug(unsigned int dump_level, unsigned int verify_level,
                      struct _C_mallinfo_t *output_record, size_t sizeofoutput);

```

**Language Level:** Extended

**Threadsafe:** Yes.

### Description

The `_C_TS_malloc_debug()` function determines the amount of teraspace memory used and returns the information within the given `output_record` structure. If the given `dump_level` parameter is greater than 0, it also dumps the internal memory structures used to `stdout`. If the given `verify_level` parameter is greater than 0, it also performs verification checks for the internal memory structures. If a verification fails, a message is generated to `stdout` indicating the failure. If both the `dump_level` and `verify_level` parameters are 0, this function provides the same behavior as the `_C_TS_malloc_info` function.

The following macros are defined within the <mallocinfo.h> include file to be specified for the `dump_level` parameter:

<code>_C_NO_DUMPS</code>	No information is dumped
<code>_C_DUMP_TOTALS</code>	Overall totals and totals for each chunk are printed
<code>_C_DUMP_CHUNKS</code>	Additional information about each chunk is printed
<code>_C_DUMP_NODES</code>	Additional information for all nodes within each chunk is printed
<code>_C_DUMP_TREE</code>	Additional information for the cartesian tree used to track free nodes is printed
<code>_C_DUMP_ALL</code>	All available information is printed

The following macros are defined within the <mallocinfo.h> include file to be specified for the `verify_level` parameter:

<code>_C_NO_CHECKS</code>	No verification checks are performed
<code>_C_CHECK_TOTALS</code>	Totals are verified for correctness
<code>_C_CHECK_CHUNKS</code>	Additional verifications are performed for each chunk
<code>_C_CHECK_NODES</code>	Additional verifications are performed for all nodes within each chunk
<code>_C_CHECK_TREE</code>	Additional verifications are performed for the cartesian tree used to track free nodes
<code>_C_CHECK_ALL</code>	All verifications are performed
<code>_C_CHECK_ALL_AND_ABORT</code>	All verifications are performed, and if any verification fails, the <code>abort()</code> function is called

**Note:** This function is for low-level debug of teraspace memory usage within an application.

### Return Value

If successful, the function returns 0. If an error occurs, the function returns a negative value.

**Example that uses `_C_TS_malloc_debug()`**

This example prints the information returned from `_C_TS_malloc_debug()` to `stdout`. This program is compiled with `TERASPACE(*YES *TSIFC)`.

```

#include <stdio.h>
#include <stdlib.h>
#include <mallocinfo.h>

int main (void)
{
    _C_mallinfo_t info;
    int rc;
    void *m;

    /* Allocate a small chunk of memory */
    m = malloc(500);

    rc = _C_TS_malloc_debug(_C_DUMP_TOTALS,
                           _C_NO_CHECKS,
                           &info, sizeof(info));

    if (rc == 0) {
        Printf("_C_TS_malloc_debug successful\n");
    }
    else {
        printf("_C_TS_malloc_debug failed (rc = %d)\n", rc);
    }

    free(m);
}

```

```

/*****

```

The output should be similar to:

```

total_bytes      = 524288
allocated_bytes  = 688
unallocated_bytes = 523600
allocated_blocks = 1
unallocated_blocks = 1
requested_bytes  = 500
pad_bytes        = 12
overhead_bytes   = 176
Number of memory chunks = 1
Total bytes      = 524288
Total allocated bytes = 688
Total unallocated bytes = 523600
Total allocated blocks = 1
Total unallocated blocks = 1
Total requested bytes = 500
Total pad bytes    = 12
Total overhead bytes = 176
_C_TS_malloc_debug successful

```

```

*****

```

### Related Information

- “\_C\_TS\_malloc\_info() — Determine amount of teraspace memory used”
- “calloc() — Reserve and Initialize Storage” on page 55
- “free() — Release Storage Blocks” on page 128
- “malloc() — Reserve Storage Block” on page 195
- “realloc() — Change Reserved Storage Block Size” on page 264
- “<mallocinfo.h>” on page 8

---

## **\_C\_TS\_malloc\_info() — Determine amount of teraspace memory used**

### Format

```
#include <mallocinfo.h>
int _C_TS_malloc_info(struct _C_mallinfo_t *output_record, size_t sizeofoutput);
```

**Language Level:** Extended

**Threadsafe:** Yes.

### Description

The `_C_TS_malloc_info()` function determines the amount of teraspace memory used and returns the information within the given `output_record` structure.

**Note:** This function is for low-level debug of teraspace memory usage within an application.

### Return Value

If successful, the function returns 0. If an error occurs, the function returns a negative value.

### Example that uses `_C_TS_malloc_info()`

This example prints the information returned from `_C_TS_malloc_info()` to `stdout`. This program is compiled with `TERASPACE(*YES *TSIFC)`.

```

#include <stdio.h>
#include <stdlib.h>
#include <mallocinfo.h>

int main (void)
{
    _C_mallinfo_t info;
    int          rc;
    void         *m;

    /* Allocate a small chunk of memory */
    m = malloc(500);

    rc = _C_TS_malloc_info(&info, sizeof(info));

    if (rc == 0) {
        printf("Total bytes          = %llu\n",
              info.total_bytes);
        printf("Total allocated bytes   = %llu\n",
              info.allocated_bytes);
        printf("Total unallocated bytes = %llu\n",
              info.unallocated_bytes);
        printf("Total allocated blocks   = %llu\n",
              info.allocated_blocks);
        printf("Total unallocated blocks = %llu\n",
              info.unallocated_blocks);
        printf("Total requested bytes    = %llu\n",
              info.requested_bytes);
        printf("Total pad bytes          = %llu\n",
              info.pad_bytes);
        printf("Total overhead bytes     = %llu\n",
              info.overhead_bytes);
    }
    else {
        printf("_C_TS_malloc_info failed (rc = %d)\n", rc);
    }

    free(m);
}

```

```

/*****

```

The output should be similar to:

```

Total bytes          = 524288
Total allocated bytes = 688
Total unallocated bytes = 523600
Total allocated blocks = 1
Total unallocated blocks = 1
Total requested bytes = 500
Total pad bytes       = 12
Total overhead bytes  = 176

```

```

*****

```

## Related Information

- “\_C\_TS\_malloc\_debug() — Determine amount of teraspace memory used (with optional dumps and verification)” on page 77
- “calloc() — Reserve and Initialize Storage” on page 55
- “free() — Release Storage Blocks” on page 128
- “malloc() — Reserve Storage Block” on page 195
- “realloc() — Change Reserved Storage Block Size” on page 264
- “<mallocinfo.h>” on page 8

---

## difftime() — Compute Time Difference

### Format

```
#include <time.h>
double difftime(time_t time2, time_t time1);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `difftime()` function computes the difference in seconds between *time2* and *time1*.

### Return Value

The `difftime()` function returns the elapsed time in seconds from *time1* to *time2* as a double precision number. Type `time_t` is defined in `<time.h>`.

### Example that uses `difftime()`

This example shows a timing application that uses `difftime()`. The example calculates how long, on average, it takes to find the prime numbers from 2 to 10 000.

```

#include <time.h>
#include <stdio.h>

#define RUNS 1000
#define SIZE 10000

int mark[SIZE];

int main(void)
{
    time_t start, finish;
    int i, loop, n, num;

    time(&start);

    /* This loop finds the prime numbers between 2 and SIZE */
    for (loop = 0; loop < RUNS; ++loop)
    {
        for (n = 0; n < SIZE; ++n)
            mark [n] = 0;
        /* This loops marks all the composite numbers with -1 */
        for (num = 0, n = 2; n < SIZE; ++n)
            if ( ! mark[n])
            {
                for (i = 2 * n; i < SIZE; i += n)
                    mark[i] = -1;
                ++num;
            }
    }
    time(&finish);
    printf("Program takes an average of %f seconds "
           "to find %d primes.\n",
           difftime(finish,start)/RUNS, num);
}

/***** Output should be similar: *****/

The program takes an average of 0.106000 seconds to find 1229 primes.
*/

```

## Related Information

- “asctime() — Convert Time to Character String” on page 39
- “asctime\_r() — Convert Time to Character String (Restartable)” on page 41
- “ctime() — Convert Time to Character String” on page 71
- | • “ctime64() — Convert Time to Character String” on page 73
- | • “ctime64\_r() — Convert Time to Character String (Restartable)” on page 76
- “ctime\_r() — Convert Time to Character String (Restartable)” on page 74
- “difftime64() — Compute Time Difference” on page 84
- “gmtime() — Convert Time” on page 161
- | • “gmtime64() — Convert Time” on page 163
- | • “gmtime64\_r() — Convert Time (Restartable)” on page 167
- “gmtime\_r() — Convert Time (Restartable)” on page 165
- “localtime() — Convert Time” on page 185
- | • “localtime64() — Convert Time” on page 187
- | • “localtime64\_r() — Convert Time (Restartable)” on page 189
- “localtime\_r() — Convert Time (Restartable)” on page 188
- “mktime() — Convert Local Time” on page 218
- | • “mktime64() — Convert Local Time” on page 220

- “strftime() — Convert Date/Time to String” on page 370
- “time() — Determine Current Time” on page 411
- “time64() — Determine Current Time” on page 412
- “<time.h>” on page 18

---

## | **difftime64() — Compute Time Difference**

### | **Format**

```
| #include <time.h>
| double difftime64(time64_t time2, time64_t time1);
```

| **Language Level:** ILE C Extension

| **Threadsafe:** Yes.

### | **Description**

| The `difftime64()` function computes the difference in seconds between *time2* and *time1*.

### | **Return Value**

| The `difftime64()` function returns the elapsed time in seconds from *time1* to *time2* as a double precision number. Type `time64_t` is defined in `<time.h>`.

### | **Example that uses difftime64()**

| This example shows a timing application that uses `difftime64()`. The example calculates how long, on average, it takes to find the prime numbers from 2 to 10 000.

```

| #include <time.h>
| #include <stdio.h>
|
| #define RUNS 1000
| #define SIZE 10000
|
| int mark[SIZE];
|
| int main(void)
| {
|     time64_t start, finish;
|     int i, loop, n, num;
|
|     time64(&start);
|
|     /* This loop finds the prime numbers between 2 and SIZE */
|     for (loop = 0; loop < RUNS; ++loop)
|     {
|         for (n = 0; n < SIZE; ++n)
|             mark [n] = 0;
|         /* This loops marks all the composite numbers with -1 */
|         for (num = 0, n = 2; n < SIZE; ++n)
|             if ( ! mark[n])
|             {
|                 for (i = 2 * n; i < SIZE; i += n)
|                     mark[i] = -1;
|                 ++num;
|             }
|     }
|     time64(&finish);
|     printf("Program takes an average of %f seconds "
|           "to find %d primes.\n",
|           difftime64(finish,start)/RUNS, num);
| }
|
| /***** Output should be similar: *****/
|
| The program takes an average of 0.106000 seconds to find 1229 primes.
| */

```

## Related Information

- | • “asctime() — Convert Time to Character String” on page 39
- | • “asctime\_r() — Convert Time to Character String (Restartable)” on page 41
- | • “ctime() — Convert Time to Character String” on page 71
- | • “ctime64() — Convert Time to Character String” on page 73
- | • “ctime64\_r() — Convert Time to Character String (Restartable)” on page 76
- | • “ctime\_r() — Convert Time to Character String (Restartable)” on page 74
- | • “difftime() — Compute Time Difference” on page 82
- | • “gmtime() — Convert Time” on page 161
- | • “gmtime64() — Convert Time” on page 163
- | • “gmtime64\_r() — Convert Time (Restartable)” on page 167
- | • “gmtime\_r() — Convert Time (Restartable)” on page 165
- | • “localtime() — Convert Time” on page 185
- | • “localtime64() — Convert Time” on page 187
- | • “localtime64\_r() — Convert Time (Restartable)” on page 189
- | • “localtime\_r() — Convert Time (Restartable)” on page 188
- | • “mktime() — Convert Local Time” on page 218
- | • “mktime64() — Convert Local Time” on page 220

- | • “strftime() — Convert Date/Time to String” on page 370
- | • “time() — Determine Current Time” on page 411
- | • “time64() — Determine Current Time” on page 412
- | • “<time.h>” on page 18

---

## div() — Calculate Quotient and Remainder

### Format

```
#include <stdlib.h>
div_t div(int numerator, int denominator);
```

**Language Level:** ANSI

**Threadsafe:** Yes. However, only the function version is threadsafe. The macro version is NOT threadsafe.

### Description

The `div()` function calculates the quotient and remainder of the division of *numerator* by *denominator*.

### Return Value

The `div()` function returns a structure of type `div_t`, containing both the quotient `int quot` and the remainder `int rem`. If the return value cannot be represented, its value is undefined. If *denominator* is 0, an exception will be raised.

### Example that uses `div()`

This example uses `div()` to calculate the quotients and remainders for a set of two dividends and two divisors.

```

#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    int num[2] = {45,-45};
    int den[2] = {7,-7};
    div_t ans; /* div_t is a struct type containing two ints:
                'quot' stores quotient; 'rem' stores remainder */
    short i,j;

    printf("Results of division:\n");
    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
            {
                ans = div(num[i],den[j]);
                printf("Dividend: %6d Divisor: %6d", num[i], den[j]);
                printf(" Quotient: %6d Remainder: %6d\n", ans.quot, ans.rem);
            }
}

/***** Output should be similar to: *****/

```

```

Results of 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

- “ldiv() — lldiv() — Perform Long and Long Long Division” on page 179
- “<stdlib.h>” on page 17

## erf() – erfc() — Calculate Error Functions

### Format

```

#include <math.h>
double erf(double x);
double erfc(double x);

```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

### Description

The erf() function calculates the error function of:

$$2\pi^{-1/2} \int_0^x e^{-t^2} dt$$

The erfc() function computes the value of 1.0 - erf(x). The erfc() function is used in place of erf() for large values of x.

### Return Value

The `erf()` function returns a double value that represents the error function. The `erfc()` function returns a double value representing  $1.0 - \text{erf}$ .

### Example that uses `erf()`

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: %lf\n", value);

    value = erfc(largex);        /* value = 2.088488e-45 */
    printf("Error value for 10.0: %le\n", value);
}

/***** Output should be similar to: *****/

Error value for 0.1: 0.112463
Error value for 10.0: 2.088488e-45
*/
```

### Related Information

- “Bessel Functions” on page 50
- “`gamma()` — Gamma Function” on page 150
- “`<math.h>`” on page 8

---

## `exit()` — End Program

### Format

```
#include <stdlib.h>
void exit(int status);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `exit()` function returns control to the host environment from the program. It first calls all functions that are registered with the `atexit()` function, in reverse order; that is, the last one that is registered is the first one called. It deletes all buffers and closes all open files before ending the program.

The argument *status* can have a value from 0 to 255 inclusive, or be one of the macros `EXIT_SUCCESS` or `EXIT_FAILURE`. A *status* value of `EXIT_SUCCESS` or 0 indicates a normal exit; otherwise, another status value is returned.

**Note:** When compiled with `SYSIFCOPT(*ASYNC SIGNAL)`, `exit()` cannot be called in a signal handler.

### Return Value

The `exit()` function returns both control and the value of *status* to the operating system.

### Example that uses `exit()`

This example ends the program after deleting buffers and closing any open files if it cannot open the file `myfile`.

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

FILE *stream;

int main(void)
{
    if ((stream = fopen("mylib/myfile", "r")) == NULL)
    {
        perror("Could not open data file");
        exit(EXIT_FAILURE);
    }
}
```

### Related Information

- “`abort()` — Stop a Program” on page 36
- “`atexit()` — Record Program Ending Function” on page 45
- “`signal()` — Handle Interrupt Signals” on page 346
- “`<stdlib.h>`” on page 17

---

## `exp()` — Calculate Exponential Function

### Format

```
#include <math.h>
double exp(double x);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `exp()` function calculates the exponential value of a floating-point argument  $x$  ( $e^x$ , where  $e$  equals 2.718281828...).

### Return Value

If an overflow occurs, the `exp()` function returns `HUGE_VAL`. If an underflow occurs, it returns 0. Both overflow and underflow set `errno` to `ERANGE`. The value of `errno` can also be set to `EDOM`.

### Example that uses `exp()`

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( %lf ) = %lf\n", x, y);
}

/***** Output should be similar to: *****/

exp( 5.000000 ) = 148.413159
*/

```

### Related Information

- “log() — Calculate Natural Logarithm” on page 191
- “log10() — Calculate Base 10 Logarithm” on page 191
- “<math.h>” on page 8

---

## fabs() — Calculate Floating-Point Absolute Value

### Format

```

#include <math.h>
double fabs(double x);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `fabs()` function calculates the absolute value of the floating-point argument  $x$ .

### Return Value

The `fabs()` function returns the absolute value. There is no error return value.

### Example that uses `fabs()`

This example calculates  $y$  as the absolute value of  $x$ :

```

#include <math.h>
#include <stdio.h>

int main(void)
{
    double x, y;

    x = -5.6798;
    y = fabs(x);

    printf("fabs( %lf ) = %lf\n", x, y);
}

/***** Output should be similar to: *****/

fabs( -5.679800 ) = 5.679800
*/

```

### Related Information

- “abs() — Calculate Integer Absolute Value” on page 37
- “labs() — llabs() — Calculate Absolute Value of Long and Long Long Integer” on page 177
- “<math.h>” on page 8

---

## fclose() — Close Stream

### Format

```

#include <stdio.h>
int fclose(FILE *stream);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `fclose()` function closes a stream pointed to by *stream*. This function deletes all buffers that are associated with the stream before closing it. When it closes the stream, the function releases any buffers that the system reserved. When a binary stream is closed, the last record in the file is padded with null characters (`\0`) to the end of the record.

### Return Value

The `fclose()` function returns 0 if it successfully closes the stream, or EOF if any errors were detected.

The value of `errno` can be set to:

### Value Meaning

#### ENOTOPEN

The file is not open.

#### EIOERROR

A non-recoverable I/O error occurred.

#### EIORECERR

A recoverable I/O error occurred.

#### ESCANFAILURE

The file was marked with a scan failure.

**Note:** The storage pointed to by the FILE pointer is freed by the `fclose()` function. After the use of the `fclose()` function, any attempt to use the FILE pointer is not valid.

### Example that uses `fclose()`

This example opens a file `myfile` for reading as a stream; then it closes this file.

```
#include <stdio.h>

#define NUM_ALPHA 26

int main(void)
{
    FILE *stream;
    char buffer[NUM_ALPHA];

    if (( stream = fopen("mylib/myfile", "r")) != NULL )
    {
        fread( buffer, sizeof( char ), NUM_ALPHA, stream );
        printf( "buffer = %s\n", buffer );
    }

    if (fclose(stream)) /* Close the stream. */
        perror("fclose error");
    else printf("File mylib/myfile closed successfully.\n");
}
```

### Related Information

- “`fflush()` — Write Buffer to File” on page 96
- “`fopen()` — Open Files” on page 109
- “`freopen()` — Redirect Open Files” on page 130
- “`<stdio.h>`” on page 15

---

## `fdopen()` — Associates Stream With File Descriptor

### Format

```
#include <stdio.h>
FILE *fdopen(int handle, char *type);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

| **Integrated File System Interface:** This function is not available when `SYSIFCOPT(*NOIFSIO)` is specified on the compilation command.

### Description

| The `fdopen()` function associates an input or output stream with the file that is identified by *handle*. The *type* variable is a character string specifying the type of access that is requested for the stream. The variable contains one positional parameter that is followed by optional keyword parameters.

| The possible values for the positional parameters are:

#### Mode Description

**r** Create a stream to read a text file. The file pointer is set to the beginning of the file.

**w** Create a stream to write to a text file. The file pointer is set to the beginning of the file.

- a** Create a stream to write, in append mode, at the end of the text file. The file pointer is set to the end of the file.
- r+** Create a stream for reading and writing a text file. The file pointer is set to the beginning of the file.
- w+** Create a stream for reading and writing a text file. The file pointer is set to the beginning of the file.
- a+** Create a stream for reading or writing, in append mode, at the end of the text file. The file pointer is set to the end of the file.
- rb** Create a stream to read a binary file. The file pointer is set to the beginning of the file.
- wb** Create a stream to write to a binary file. The file pointer is set to the beginning of the file.
- ab** Create a stream to write to a binary file in append mode. The file pointer is set to the end of the file.
- r+b or rb+**  
Create a stream for reading and writing a binary file. The file pointer is set to the beginning of the file.
- w+b or wb+**  
Create a stream for reading and writing a binary file. The file pointer is set to the beginning of the file.
- a+b or ab+**  
Create a stream for reading and writing to a binary file in append mode. The file pointer is set to the end of the file.

**Note:** Use the `w`, `w+`, `wb`, `wb+`, and `w+b` modes with care; they can destroy existing files.

The specified *type* must be compatible with the access method you used to open the file. If the file was opened with the `O_APPEND` flag, the stream mode must be `a`, `a+`, `ab`, `a+b`, or `ab+`. To use the `fdopen()` function you need a file descriptor. To get a descriptor use the POSIX function `open()`. The `O_APPEND` flag is a mode for `open()`. Modes for `open()` are defined in `QSYSINC/H/FCNTL`. For further information see the APIs topic in the i5/OS Information Center.

- | The keyword parameters allowed for `fdopen()` are the same as those documented in “`fopen()` — Open
- | Files” on page 109 that are for the integrated file system.

If `fdopen()` returns `NULL`, use `close()` to close the file. If `fdopen()` is successful, you must use `fclose()` to close the stream and file.

### Return Value

The `fdopen()` function returns a pointer to a file structure that can be used to access the open file. A `NULL` pointer return value indicates an error.

### Example that uses `fdopen()`

This example opens the file `sample.dat` and associates a stream with the file using `fdopen()`. It then reads from the stream into the buffer.

```

/* compile with SYSIFCOPT(*IFSIO) */
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>

int main(void)
{
    long length;
    int fh;
    char buffer[20];
    FILE *fp;

    printf("\nCreating sample.dat.\n");
    if ((fp= fopen("/sample.dat", "w")) == NULL) {
        perror(" File was not created: ");
        exit(1);
    }
    fputs("Sample Program", fp);
    fclose(fp);

    memset(buffer, '\0', 20);                /* Initialize buffer*/

    if (-1 == (fh = open("/sample.dat", O_RDWR|O_APPEND))) {
        perror("Unable to open sample.dat");
        exit(1);
    }
    if (NULL == (fp = fdopen(fh, "r"))) {
        perror("fdopen failed");
        close(fh);
        exit(1);
    }
    if (14 != fread(buffer, 1, 14, fp)) {
        perror("fread failed");
        fclose(fp);
        exit(1);
    }
    printf("Successfully read from the stream the following:\n%s.\n", buffer);
    fclose(fp);
    return 1;

    /*****
    * The output should be:
    *
    * Creating sample.dat.
    * Successfully read from the stream the following:
    * Sample Program.
    */
}

```

## Related Information

- “fclose() — Close Stream” on page 91
- “fopen() — Open Files” on page 109
- “fseek() — fseeko() — Reposition File Position” on page 134
- “fsetpos() — Set File Position” on page 136
- “rewind() — Adjust Current File Position” on page 276
- “<stdio.h>” on page 15
- open API in the APIs topic in the i5/OS Information Center.
- close API in the APIs topic in the i5/OS Information Center.

---

## feof() — Test End-of-File Indicator

### Format

```
#include <stdio.h>
int feof(FILE *stream);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `feof()` function indicates whether the end-of-file flag is set for the given *stream*. The end-of-file flag is set by several functions to indicate the end of the file. The end-of-file flag is cleared by calling the `rewind()`, `fsetpos()`, `fseek()`, or `clearerr()` functions for this stream.

### Return Value

The `feof()` function returns a nonzero value if and only if the EOF flag is set; otherwise, it returns 0.

### Example that uses feof()

This example scans the input stream until it reads an end-of-file character.

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

int main(void)
{
    char string[100];
    FILE *stream;
    memset(string, 0, sizeof(string));
    stream = fopen("qcpple/qacsrc(feof)", "r");

    fscanf(stream, "%s", string);
    while (!feof(stream))
    {
        printf("%s\n", string);
        memset(string, 0, sizeof(string));
        fscanf(stream, "%s", string);
    }
}
```

### Related Information

- “`clearerr()` — Reset Error Indicators” on page 62
- “`ferror()` — Test for Read/Write Errors”
- “`fseek()` — `fseeko()` — Reposition File Position” on page 134
- “`fsetpos()` — Set File Position” on page 136
- “`perror()` — Print Error Message” on page 227
- “`rewind()` — Adjust Current File Position” on page 276
- “`<stdio.h>`” on page 15

---

## ferror() — Test for Read/Write Errors

### Format

```
#include <stdio.h>
int ferror(FILE *stream);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `ferror()` function tests for an error in reading from or writing to the given *stream*. If an error occurs, the error indicator for the *stream* remains set until you close *stream*, call the `rewind()` function, or call the `clearerr()` function.

### Return Value

The `ferror()` function returns a nonzero value to indicate an error on the given *stream*. A return value of 0 means that no error has occurred.

### Example that uses `ferror()`

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("mylib/myfile", "w");

    fprintf(stream, "%s\n", string);
    if (ferror(stream))
    {
        printf("write error\n");
        clearerr(stream);
    }
    if (fclose(stream))
        perror("fclose error");
}
```

### Related Information

- “`clearerr()` — Reset Error Indicators” on page 62
- “`feof()` — Test End-of-File Indicator” on page 95
- “`fopen()` — Open Files” on page 109
- “`perror()` — Print Error Message” on page 227
- “`strerror()` — Set Pointer to Runtime Error Message” on page 367
- “`<stdio.h>`” on page 15

---

## **fflush() — Write Buffer to File**

### Format

```
#include <stdio.h>
int fflush(FILE *stream);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

## Description

The `fflush()` function causes the system to empty the buffer that is associated with the specified output *stream*, if possible. If the *stream* is open for input, the `fflush()` function undoes the effect of any `ungetc()` function. The *stream* remains open after the call.

If *stream* is `NULL`, the system flushes all open streams.

**Note:** The system automatically deletes buffers when you close the stream, or when a program ends normally without closing the stream.

## Return Value

The `fflush()` function returns the value 0 if it successfully deletes the buffer. It returns EOF if an error occurs.

The value of `errno` can be set to:

### Value Meaning

#### ENOTOPEN

The file is not open.

#### ERECIO

The file is opened for record I/O.

#### ESTDIN

`stdin` cannot be opened.

#### EIOERROR

A non-recoverable I/O error occurred.

#### EIORECERR

A recoverable I/O error occurred.

The `fflush()` function is not supported for files that are opened with `type=record`.

## Example that uses `fflush()`

This example deletes a stream buffer.

```
#include <stdio.h>

int main(void)
{
    FILE *stream;
    int ch;
    unsigned int result = 0;

    stream = fopen("mylib/myfile", "r");
    while ((ch = getc(stream)) != EOF && isdigit(ch))
        result = result * 10 + ch - '0';
    if (ch != EOF)
        ungetc(ch, stream);

    fflush(stream);          /* fflush undoes the effect of ungetc function
*/
    printf("The result is: %d\n", result);
    if ((ch = getc(stream)) != EOF)
        printf("The character is: %c\n", ch);
}
```

## Related Information

- “fclose() — Close Stream” on page 91
- “fopen() — Open Files” on page 109
- “setbuf() — Control Buffering” on page 336
- “ungetc() — Push Character onto Input Stream” on page 420
- “<stdio.h>” on page 15

---

## fgetc() — Read a Character

### Format

```
#include <stdio.h>
int fgetc(FILE *stream);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `fgetc()` function reads a single unsigned character from the input *stream* at the current position and increases the associated file pointer, if any, so that it points to the next character.

**Note:** The `fgetc()` function is identical to `getc()`, but it is always defined as a function call; it is never replaced by a macro.

### Return Value

The `fgetc()` function returns the character that is read as an integer. An EOF return value indicates an error or an end-of-file condition. Use the `feof()` or the `ferror()` function to determine whether the EOF value indicates an error or the end of the file.

The value of `errno` can be set to:

#### Value Meaning

##### EBADF

The file pointer or descriptor is not valid.

##### ECONVERT

A conversion error occurred.

##### ENOTREAD

The file is not open for read operations.

##### EGETANDPUT

An read operation that was not allowed occurred after a write operation.

##### ERECIO

The file is open for record I/O.

##### ESTDIN

`stdin` cannot be opened.

##### EIOERROR

A non-recoverable I/O error occurred.

##### EIORECERR

A recoverable I/O error occurred.

The `fgetc()` function is not supported for files that are opened with `type=record`.

### Example that uses `fgetc()`

This example gathers a line of input from a stream.

```
#include <stdio.h>

#define MAX_LEN 80

int main(void)
{
    FILE *stream;
    char buffer[MAX_LEN + 1];
    int i, ch;

    stream = fopen("mylib/myfile", "r");

    for (i = 0; (i < (sizeof(buffer)-1) &&
        ((ch = fgetc(stream)) != EOF) && (ch != '\n')); i++)
        buffer[i] = ch;

    buffer[i] = '\0';

    if (fclose(stream))
        perror("fclose error");

    printf("line: %s\n", buffer);
}

/*****
    If FILENAME contains: one two three
    The output should be:
    line: one two three
*****/
```

### Related Information

- “`feof()` — Test End-of-File Indicator” on page 95
- “`ferror()` — Test for Read/Write Errors” on page 95
- “`fgetc()` — Read Wide Character from Stream” on page 102
- “`fputc()` — Write Character” on page 118
- “`getc()` – `getchar()` — Read a Character” on page 152
- “`getwc()` — Read Wide Character from Stream” on page 157
- “`getwchar()` — Get Wide Character from `stdin`” on page 159
- “`<stdio.h>`” on page 15

---

## **fgetpos()** — Get File Position

### Format

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

**Language Level:** ANSI

**Threadsafe:** YES

### Description

The `fgetpos()` function stores the current position of the file pointer that is 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*.

### Return Value

The `fgetpos()` function returns 0 if successful; on error, it returns nonzero and sets `errno` to a nonzero value.

The value of `errno` can be set to:

#### Value Meaning

##### EBADF

The file pointer or descriptor is not valid.

##### EBADSEEK

Bad offset for a seek operation.

##### ENODEV

Operation was attempted on a wrong device.

##### ENOTOPEN

The file is not open.

##### ERECIO

The file is open for record I/O.

##### ESTDERR

`stderr` cannot be opened.

##### ESTDIN

`stdin` cannot be opened.

##### ESTDOUT

`stdout` cannot be opened.

##### EIOERROR

A non-recoverable I/O error occurred.

##### EIORECERR

A recoverable I/O error occurred.

The `fgetpos()` function is not supported for files that are opened with `type=record`.

### Example that uses `fgetpos()`

This example opens the file `myfile` for reading and stores the current file pointer position into the variable *pos*.

```

#include <stdio.h>

FILE *stream;

int main(void)
{
    int retcode;
    fpos_t pos;

    stream = fopen("mylib/myfile", "rb");

    /* The value returned by fgetpos can be used by fsetpos */
    /* to set the file pointer if 'retcode' is 0          */

    if ((retcode = fgetpos(stream, Point-of-Sale)) == 0)
        printf("Current position of file pointer found\n");
    fclose(stream);
}

```

### Related Information

- “fseek() — fseeko() — Reposition File Position” on page 134
- “fsetpos() — Set File Position” on page 136
- “ftell() — ftello() — Get Current Position” on page 138
- “<stdio.h>” on page 15

---

## fgets() — Read a String

### Format

```

#include <stdio.h>
char *fgets (char *string, int n, FILE *stream);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `fgets()` function reads characters from the current *stream* position up to and including the first new-line character (`\n`), up to the end of the stream, or until the number of characters read is equal to *n*-1, 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. If *n* is equal to 1, the *string* is empty.

### Return Value

The `fgets()` function returns a pointer to the *string* buffer if successful. A NULL return value indicates an error or an end-of-file condition. Use the `feof()` or `ferror()` functions to determine whether the NULL value indicates an error or the end of the file. In either case, the value of the string is unchanged.

The `fgets()` function is not supported for files that are opened with `type=record`.

The value of `errno` can be set to:

### Value Meaning

#### EBADF

The file pointer or descriptor is not valid.

**ECONVERT**

A conversion error occurred.

**ENOTREAD**

The file is not open for read operations.

**EGETANDPUT**

An read operation that was not allowed occurred after a write operation.

**ERECIO**

The file is open for record I/O.

**ESTDIN**

stdin cannot be opened.

**EIOERROR**

A non-recoverable I/O error occurred.

**EIORECERR**

A recoverable I/O error occurred.

**Example that uses fgets()**

This example gets a line of input from a data stream. The example 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("mylib/myfile","rb");

    if ((result = fgets(line,MAX_LEN,stream)) != NULL)
        printf("The string is %s\n", result);

    if (fclose(stream))
        perror("fclose error");
}
```

**Related Information**

- “feof() — Test End-of-File Indicator” on page 95
- “ferror() — Test for Read/Write Errors” on page 95
- “fgetws() — Read Wide-Character String from Stream” on page 104
- “fputs() — Write String” on page 121
- “gets() — Read a Line” on page 156
- “puts() — Write a String” on page 241
- “<stdio.h>” on page 15

---

## **fgetwc() — Read Wide Character from Stream**

**Format**

```
#include <wchar.h>
#include <stdio.h>
wint_t fgetwc(FILE *stream);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the  
| current locale. The behavior might also be affected by the LC\_UNI\_CTYPE category of the current locale  
| if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation  
| command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation  
| command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified  
| on the compilation command.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `fgetwc()` reads 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).

Using non-wide-character functions with `fgetwc()` on the same stream results in undefined behavior. After calling `fgetwc()`, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless EOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling `fgetwc()`.

| **Note:** If the current locale is changed between subsequent read operations on the same stream, undefined  
| results can occur.

### Return Value

The `fgetwc()` function 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 the `fgetwc()` function returns WEOF. If an encoding error occurs (an error converting the multibyte character into a wide character), the `fgetwc()` function sets `errno` to `EILSEQ` and returns WEOF.

Use the `ferror()` and `feof()` functions to distinguish between a read error and an EOF. 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.

The value of `errno` can be set to:

#### Value Meaning

##### EBADF

The file pointer or descriptor is not valid.

##### ENOTREAD

The file is not open for read operations.

##### EGETANDPUT

An read operation that was not allowed occurred after a write operation.

##### ERECIO

The file is open for record I/O.

**ESTDIN**

stdin cannot be opened.

**EIOERROR**

A non-recoverable I/O error occurred.

**EIORECERR**

A recoverable I/O error occurred.

**EILSEQ**

An invalid multibyte character sequence was encountered.

**ECONVERT**

A conversion error occurred.

**Example that uses fgetwc()**

This example opens a file, reads in each wide character, and prints out the characters.

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
    FILE *stream;
    wint_t wc;

    if (NULL == (stream = fopen("fgetwc.dat", "r"))) {
        printf("Unable to open: \"fgetwc.dat\"\n");
        exit(1);
    }

    errno = 0;
    while (WEOF != (wc = fgetwc(stream)))
        printf("wc = %lc\n", wc);

    if (EILSEQ == errno) {
        printf("An invalid wide character was encountered.\n");
        exit(1);
    }
    fclose(stream);
    return 0;
}
* * * End of File * * *
```

**Related Information**

- “fgetc() — Read a Character” on page 98
- “fputwc() — Write Wide Character” on page 122
- “fgetws() — Read Wide-Character String from Stream”
- “getc() – getchar() — Read a Character” on page 152
- “getwc() — Read Wide Character from Stream” on page 157
- “getwchar() — Get Wide Character from stdin” on page 159
- “<stdio.h>” on page 15
- “<wchar.h>” on page 18

---

## **fgetws() — Read Wide-Character String from Stream**

**Format**

```
#include <wchar.h>
#include <stdio.h>
wchar_t *fgetws(wchar_t *wcs, int n, FILE *stream);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the  
| current locale. The behavior might also be affected by the LC\_UNI\_CTYPE category of the current locale  
| if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation  
| command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation  
| command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified  
| on the compilation command.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `fgetws()` function reads at most one less than the number of wide characters specified by *n* from the stream pointed to by *stream*. The `fgetws()` function stops reading characters after WEOF, or after it reads a new-line wide character (which is retained). It adds a null wide character immediately after the last wide character read into the array. The `fgetws()` function advances the file position unless there is an error. If an error occurs, the file position is undefined.

Using non-wide-character functions with the `fgetws()` function on the same stream results in undefined behavior. After calling the `fgetws()` function, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless WEOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling the `fgetws()` function.

| **Note:** If the current locale is changed between subsequent read operations on the same stream, undefined  
| results can occur.

### Return Value

If successful, the `fgetws()` function returns a pointer to the wide-character string *wcs*. If WEOF is encountered before any wide characters have been read into *wcs*, the contents of *wcs* remain unchanged and the `fgetws()` function returns a null pointer. If WEOF is reached after data has already been read into the string buffer, the `fgetws()` function returns a pointer to the string buffer to indicate success. A subsequent call would return NULL because WEOF would be reached without any data being read.

If a read error occurs, the contents of *wcs* are indeterminate, and the `fgetws()` function returns NULL. If an encoding error occurs (in converting a wide character to a multibyte character), the `fgetws()` function sets `errno` to `EILSEQ` and returns NULL.

If *n* equals 1, the *wcs* buffer has only room for the ending null character, and nothing is read from the stream. (Such an operation is still considered a read operation, so it cannot immediately follow a write operation unless the buffer is flushed or the stream pointer repositioned first.) If *n* is greater than 1, the `fgetws()` function fails only if an I/O error occurs, or if WEOF is reached before data is read from the stream.

Use the `ferror()` and `feof()` functions to distinguish between a read error and a WEOF. A WEOF error 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 WEOF indicator.

For information about `errno` values for `fgetws()`, see “`fgetwc() — Read Wide Character from Stream`” on page 102.

### Example that uses `fgetws()`

This example opens a file, reads in the file contents, then prints the file contents.

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
    FILE    *stream;
    wchar_t  wcs[100];

    if (NULL == (stream = fopen("fgetws.dat", "r"))) {
        printf("Unable to open: \"fgetws.dat\"\n");
        exit(1);
    }

    errno = 0;
    if (NULL == fgetws(wcs, 100, stream)) {
        if (EILSEQ == errno) {
            printf("An invalid wide character was encountered.\n");
            exit(1);
        }
        else if (feof(stream))
            printf("End of file reached.\n");
        else
            perror("Read error.\n");
    }
    printf("wcs = \"%ls\"\n", wcs);
    fclose(stream);
    return 0;

    /*****
    Assuming the file fgetws.dat contains:

    This test string should not return -1

    The output should be similar to:

    wcs = "This test string should not return -1"
    *****/
}
```

### Related Information

- “`fgetc() — Read a Character`” on page 98
- “`fgets() — Read a String`” on page 101
- “`fgetwc() — Read Wide Character from Stream`” on page 102
- “`fputws() — Write Wide-Character String`” on page 124
- “`<stdio.h>`” on page 15
- “`<wchar.h>`” on page 18

---

## **fileno() — Determine File Handle**

### Format

```
#include <stdio.h>
int fileno(FILE *stream);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

| **Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified  
| on the compilation command.

### Description

The `fileno()` function determines the file handle that is currently associated with *stream*.

### Return Value

If the environment variable `QIBM_USE_DESCRIPTOR_STDIO` is set to Yes, the `fileno()` function returns 0 for `stdin`, 1 for `stdout`, and 2 for `stderr`.

With `QIBM_USE_DESCRIPTOR_STDIO` set to No, the ILE C session files `stdin`, `stdout`, and `stderr` do not have a file descriptor associated with them. The `fileno()` function will return a value of -1 in this case.

The value of `errno` can be set to `EBADF`.

### Example that uses `fileno()`

This example determines the file handle of the `stderr` data stream.

```
/* Compile with SYSIFCOPT(*IFSIO)          */
#include <stdio.h>

int main (void)
{
    FILE *fp;
    int result;

    fp = fopen ("stderr","w");

    result = fileno(fp);
    printf("The file handle associated with stderr is %d.\n", result);
    return 0;

    /*****
     * The output should be:
     *
     * The file handle associated with stderr is -1.
     *****/
}
```

### Related Information

- “`fopen()` — Open Files” on page 109
- “`freopen()` — Redirect Open Files” on page 130
- “`<stdio.h>`” on page 15

---

## **floor()** — Find Integer $\leq$ Argument

### Format

```
#include <math.h>
double floor(double x);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `floor()` function calculates the largest integer that is less than or equal to  $x$ .

### Return Value

The `floor()` function returns the floating-point result as a double value.

The result of `floor()` cannot have a range error.

### Example that uses `floor()`

This example assigns  $y$  value of the largest integer less than or equal to 2.8 and  $z$  the value of the largest integer 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 ) = %lf\n", y);
    printf("floor( -2.8 ) = %lf\n", z);
}
/***** Output should be similar to: *****/

floor( 2.8 ) = 2.000000
floor( -2.8 ) = -3.000000
*/
```

### Related Information

- “`ceil()` — Find Integer  $\geq$ Argument” on page 61
- “`fmod()` — Calculate Floating-Point Remainder”
- “`<math.h>`” on page 8

---

## **fmod() — Calculate Floating-Point Remainder**

### Format

```
#include <math.h>
double fmod(double x, double y);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `fmod()` function 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  $x$ .

### Return Value

The `fmod()` function returns the floating-point remainder of  $x/y$ . If  $y$  is zero or if  $x/y$  causes an overflow, `fmod()` returns 0. The value of `errno` can be set to `EDOM`.

### Example that uses `fmod()`

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( %lf, %lf) = %lf\n", x, y, z);
}

/***** Output should be similar to: *****/

fmod( -10.000000, 3.000000) = -1.000000
*/
```

### Related Information

- “`ceil()` — Find Integer  $\geq$ Argument” on page 61
- “`fabs()` — Calculate Floating-Point Absolute Value” on page 90
- “`floor()` — Find Integer  $\leq$ Argument” on page 107
- “`<math.h>`” on page 8

---

## `fopen()` — Open Files

### Format

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `fopen()` function opens the file that is specified by *filename*. The *mode* parameter is a character string specifying the type of access that is requested for the file. The *mode* variable contains one positional parameter followed by optional keyword parameters.

| **Note:** When the program is compiled with `SYSIFCOPT(*IFSIO)` or `SYSIFCOPT(*IFS64IO)`, and `fopen()`  
| creates a file in the integrated file system, the owner of the file, the owner’s group, and public is  
| given read, write, and execute authority to the file.

The possible values for the positional parameters are:

Mode	Description
<b>r</b>	Open a text file for reading. The file must exist.
<b>w</b>	Create a text file for writing. If the given file exists, its contents are destroyed unless it is a logical file.
<b>a</b>	Open a text file in append mode for writing at the end of the file. The <code>fopen()</code> function creates the file if it does not exist and is not a logical file.
<b>r+</b>	Open a text file for both reading and writing. The file must exist.
<b>w+</b>	Create a text file for both reading and writing. If the given file exists, its contents are cleared unless it is a logical file.
<b>a+</b>	Open a text file in append mode for reading or updating at the end of the file. The <code>fopen()</code> function creates the file if it does not exist.
<b>rb</b>	Open a binary file for reading. The file must exist.
<b>wb</b>	Create an empty binary file for writing. If the file exists, its contents are cleared unless it is a logical file.
<b>ab</b>	Open a binary file in append mode for writing at the end of the file. The <code>fopen</code> function creates the file if it does not exist.
<b>r+b or rb+</b>	Open a binary file for both reading and writing. The file must exist.
<b>w+b or wb+</b>	Create an empty binary file for both reading and writing. If the file exists, its contents will be cleared unless it is a logical file.
<b>a+b or ab+</b>	Open a binary file in append mode for writing at the end of the file. The <code>fopen()</code> function creates the file if it does not exist.

**Notes:**

1. The `fopen()` function is not supported for files that are opened with the attributes `type=record` and `ab+`, `rb+`, or `wb+`
2. 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 that are organized into lines. Each line ends with a new-line character, except possibly the last line, depending on the compiler. The system can insert or convert control characters in an output text stream. The `fopen()` function mode "a" and "a+" can not be used for the QSYS.LIB file system. There are implementation restrictions when using the QSYS.LIB file system for text files in all modes. Seeking beyond the start of files cannot be relied on to work with streams opened in text mode.

**Note:** When you use `fopen()` to create a file in the QSYS.LIB file system, specifying a library name of \*LIBL or blank causes the file to be created in QTEMP library.

If a text file does not exist, you can create one using the following command:

**CRTSRCPF FILE(MYLIB/MYFILE) RCDLEN(LRECL) MBR(MYMBR) SYSTEM(\*FILETYPE)**

**Note:** Data output to a text stream might not compare as equal to the same data on input. The QSYS.LIB file system treats database files as a directory of members. The database file must exist before a member can be dynamically created when using the `fopen()` function.

| See Large file support in the Integrated file system topic in the i5/OS Information Center for the current  
| file system limit of the integrated file system. For files in the integrated file system that are larger than 2  
| GB, you need to allow your application programs access to 64-bit C runtime functions. You can use the  
| following methods to allow your program access:

- Specify `SYSIFCOPT(*IFS64IO)` on a compilation command, which causes the native C compiler to define `_IFS64_IO_`. This causes the macros `_LARGE_FILES` and `_LARGE_FILE_API` to be defined.
- Define the macro `_LARGE_FILES`, either in the program source or by specifying `DEFINE('_LARGE_FILES')` on a compilation command. The existing C runtime functions and the relevant data types in the code will all be automatically mapped or redefined to their 64-bit versions.
- Define the macro `_LARGE_FILE_API`, either in the program source or by specifying `DEFINE('_LARGE_FILE_API')` on a compilation command. This makes visible the set of of new 64-bit C runtime functions and data types. The application must explicitly specify the name of the C runtime functions, both existing version and 64-bit version, to use.

The 64-bit C runtime functions include the following: `int fgetpos64()`, `FILE *fopen64()`, `FILE *freopen64()`, `FILE *wfopen64()`, `int fsetpos64(FILE *, const fpost64_t *)`, `FILE *tmpfile64()`, `int fseeko(FILE *, off_t, int)`, `int fseeko64(FILE *, off64_t, int)`, `off_t ftello(FILE *)`, and `off64_t ftello64()`.

*Binary files* contain a series of characters. For binary files, the system does not translate control characters on input or output.

If a binary file does not exist, you can create one using the following command:

```
CRTPF FILE(MYLIB/MYFILE) RCDLEN(LRECL) MBR(MYMBR) MAXMBRS(*NOMAX)  
SYSTEM(*FILETYPE)
```

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 the `fseek()` function or the `rewind()` function, the write functions move the file pointer back to the end of the file before they carry out any 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 the `fseek()`, `fsetpos()`, `rewind()`, or `fflush()`. Output can immediately follow input if the end-of-file was detected.

### Keyword parameters for non-Integrated File System

**blksize=***value*

Specifies the maximum length, in bytes, of a physical block of records.

**lrecl=***value*

Specifies the length, in bytes, for fixed-length records and the maximum length for variable-length records.

**recfm=***value*

*value* can be:

**F** fixed-length, deblocked records

**FB** fixed-length, blocked records

**V** variable-length, deblocked records

**VB** variable-length, blocked records

**VBS** variable-length, blocked, spanned records for tape files

**VS** variable-length, deblocked, spanned records for tape files

- D** variable-length, deblocked, unspanned records for ASCII D format for tape files
- DB** variable-length, blocked, unspanned records for ASCII D format for tape files
- U** undefined format for tape files
- FA** fixed-length that uses first character forms control data for printer files

**Note:** If the file is created using CTLCHAR(\*FCFC), the first character form control will be used. If it is created using CTLCHAR(\*NONE), the first character form control will not be used.

**commit=***value*

*value* can be:

**N** This parameter identifies that this file is not opened under commitment control. This is the default.

**Y** This parameter identifies that this file is opened under commitment control.

**ccsid=***value*

If a CCSID that is not supported by the i5/OS operating system is specified, it is ignored by data management.

| When LOCALETYPE(\*LOCALEUTF) is specified on the compilation command, the default value  
 | is the LC\_CTYPE CCSID value, which is determined by your current locale setting. See  
 | "setlocale() — Set Locale" on page 339 for further information about locale settings. When  
 | LOCALETYPE(\*LOCALEUTF) is not specified on the compilation command, the default value is  
 | the job CCSID value. See "File CCSID" on page 525 for further information about file CCSID  
 | values.

**arrseq=***value*

*value* can be:

**N** This parameter identifies that this file is processed in the way it was created. This is the default.

**Y** This parameter identifies that this file is processed in arrival sequence.

**indicators=***value*

*value* can be:

**N** This parameter identifies that indicators in display, ICF, or printer files are stored in the file buffer. This is the default.

**Y** This parameter identifies that indicators in display, ICF, or printer files are stored in a separate indicator area, not in the file buffer. A file buffer is the area the system uses to transfer data to and from the user program and the operating system when writing and reading. You must store indicators in a separate indicator area when processing ICF files.

**type=***value*

*value* can be:

**memory** This parameter identifies this file as a memory file that is available only from C programs. This is the default.

**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, the fopen() function fails. Read and write operations are done with the fread() function and the fwrite() functions.

**Keyword parameters for Integrated File System only**

**type=***value*

*value* can be:

**record** The file is opened for sequential record I/O. (File has to be opened as binary stream.)

**ccsid=***value*

*ccsid* is converted to a code page value. The default is to use the job CCSID value as the code page. The CCSID and codepage option cannot both be specified. The CCSID option provides compatibility with i5/OS and Data management based stream I/O.

**Note:** Mixed data (the data contains both single and double-byte characters) is not supported for a file data processing mode of text. Mixed data is supported for a file processing mode of binary.

If you specify the *ccsid* keyword, you cannot specify the *o\_ccsid* keyword or the *codepage* keyword.

Because of the possible expansion or contraction of converted data, making assumptions about data size and the current file offset is dangerous. For example, a file might have a physical size of 100 bytes, but after an application has read 100 bytes from the file, the current file offset might be only 50. In order to read the whole file, the application might have to read 200 bytes or more, depending on the CCSIDs involved. Therefore, file positioning functions, such as `ftell()`, `fseek()`, `fgetpos()`, and `fsetpos()`, might not work. These functions might fail with error ENOTSUP. Read functions also will not work if buffering is on, as it is by default. To turn buffering off, use the `setvbuf` function with the `_IONBF` keyword.

The `fopen()` function might fail with the ECONVERT error when all of the following three conditions occur:

- The file data processing mode is text.
- The code page is not specified.
- The CCSID of the job is 'mixed-data' (the data contains both single-byte and double-byte characters).

**o\_ccsid=***value*

When `LOCALETYPE(*LOCALEUTF)` is specified on the compilation command, the default value is the `LC_CTYPE` CCSID value, which is determined by your current locale setting. See “`setlocale() — Set Locale`” on page 339 for further information about locale settings. When `LOCALETYPE(*LOCALEUTF)` is not specified on the compilation command, the default value is the job CCSID value. See “`File CCSID`” on page 525 for further information about file CCSID values.

This parameter is similar to the *ccsid* parameter, except that the value specified is not converted to a code page. Also, mixed data is supported. If the file is created, it is tagged with the specified CCSID. If the file already exists, data will be converted from the CCSID of the file to the specified CCSID on read operations. On write operations, the data is assumed to be in the specified CCSID, and is converted to the CCSID of the file.

Because of the possible expansion or contraction of converted data, making assumptions about data size and the current file offset is dangerous. For example, a file might have a physical size of 100 bytes, but after an application has read 100 bytes from the file, the current file offset might be only 50. In order to read the whole file, the application might have to read 200 bytes or more, depending on the CCSIDs involved. Therefore, file positioning functions such as `ftell()`, `fseek()`, `fgetpos()`, and `fsetpos()` will not work. These functions will fail with ENOTSUP. Read functions also will not work if buffering is on, as it is by default. To turn buffering off, use the `setvbuf` function with the `_IONBF` keyword.

Example that uses *o\_ccsid*

```
/* Create a file that is tagged with CCSID 37 */
if ((fp = fopen("/MYFILE" , "w, o_ccsid=37")) == NULL) {
    printf("Failed to open file with o_ccsid=37\n");
}
```

```

fclose(fp);

/* Now reopen the file with CCSID 13488, because your application
wants to deal with the data in UNICODE */

if ((fp = fopen("/MYFILE" , "r+", o_ccsid=13488)) == NULL) {
    printf("Failed to open file with o_ccsid=13488\n");
}
/* Turn buffering off because read functions do not work when
buffering is on */

if (setbuf(fp, NULL, _IONBF, 0) != 0){
    printf("Unable to turn buffering off\n");
}
/* Because you opened with o_ccsid = 13488, you must provide
all input data as unicode.
If this program is compiled with LOCALETYPE(*LOCALEUCS2),
L constraints will be unicode. */

funcreturn = fputws(L"ABC", fp); /* Write a unicode ABC to the file. */

if (funcreturn < 0) {
    printf("Error with 'fputws' on line %d\n", __LINE__);
}
/* Because the file was tagged with CCSID 37, the unicode ABC was
converted to EBCDIC ABC when it was written to the file. */

```

#### **codepage=value**

The code page that is specified by *value* is used.

If you specify the codepage keyword, you cannot specify the ccsid keyword or the o\_ccsid keyword.

If the file to be opened does not exist, and the open mode specifies that the file should be created, the file is created and tagged with the calculated code page. If the file already exists, the data read from the file is converted from the files code page to the calculated code page during the read operation. Data written to the file is assumed to be in the calculated code page and is converted to the code page of the file during the write operation.

#### **crln=value**

*value* can be:

**Y** The line terminator to be used is carriage return [CR], new line [NL] combination. When data is read, all carriage returns [CR] are stripped for string functions. When data is written to a file, carriage returns [CR] are added before each new line [NL] character. Line terminator processing only occurs when a file is open with text mode. This is the default.

**N** The line terminator to be used is new line [NL] only.

The keyword parameters are not case sensitive and should be separated by a comma.

The fopen() function generally fails if parameters are mismatched.

#### **Return Value**

The fopen() function returns a pointer to a **FILE** structure type that can be used to access the open file.

**Note:** To use stream files (type = record) with record I/O functions, you must cast the **FILE** pointer to an **RFILE** pointer.

A NULL pointer return value indicates an error.

The value of errno can be set to:

**Value Meaning**

**EBADMODE**

The file mode that is specified is not valid.

**EBADNAME**

The file name that is specified is not valid.

**ECONEVRT**

Conversion error.

**ENOENT**

No file or library.

**ENOMEM**

Storage allocation request failed.

**ENOTOPEN**

The file is not open.

**EIOERROR**

A non-recoverable I/O error occurred.

**EIORECERR**

A recoverable I/O error occurred.

**ESCANFAILURE**

The file was marked with a scan failure.

If the mode string passed to `fopen()` is correct, `fopen()` will not set `errno` to `EBADMODE`, regardless of the file type.

If the mode string that is passed to `fopen()` is not valid, `fopen()` will set `errno` to `EBADMODE`, regardless of the file type.

If the mode string passed to `fopen()` is correct, but is invalid to that specific type of file, `fopen()` will set `errno` to `ENOTOPEN`, `EIOERROR`, or `EIORECERR`, regardless of the file type.

**Example that uses `fopen()`**

This example attempts to open a file for reading.

```

#include <stdio.h>
#define MAX_LEN 60

int main(void)
{
    FILE *stream;
    fpos_t pos;
    char line1[MAX_LEN];
    char line2[MAX_LEN];
    char *result;
    char ch;
    int num;

    /* The following call opens a text file for reading.  */
    if ((stream = fopen("mylib/myfile", "r")) == NULL)
        printf("Could not open data file\n");
    else if ((result = fgets(line1,MAX_LEN,stream)) != NULL)
    {
        printf("The string read from myfile: %s\n", result);
        fclose(stream);
    }

    /* The following call opens a fixed record length file */
    /* for reading and writing.                               */
    if ((stream = fopen("mylib/myfile2", "rb+", lrecl=80, \
        blksize=240, recfm=f")) == NULL)
        printf("Could not open data file\n");
    else {
        fgetpos(stream, Point-of-Sale);
        if (!fread(line2,sizeof(line2),1,stream))
            perror("fread error");
        else printf("1st record read from myfile2: %s\n", line2);

        fsetpos(stream, Point-of-Sale);    /* Reset pointer to start of file */
        fputs(result, stream);             /* The line read from myfile is   */
                                           /* written to myfile2.                */
        fclose(stream);
    }
}

```

## Related Information

- “fclose() — Close Stream” on page 91
- “fflush() — Write Buffer to File” on page 96
- “fread() — Read Items” on page 126
- “freopen() — Redirect Open Files” on page 130
- “fseek() — fseeko() — Reposition File Position” on page 134
- “fsetpos() — Set File Position” on page 136
- “fwrite() — Write Items” on page 146
- “rewind() — Adjust Current File Position” on page 276
- “w fopen() — Open Files” on page 498
- “<stdio.h>” on page 15
- open() API in the APIs in the i5/OS Information Center.

---

## fprintf() — Write Formatted Data to a Stream

### Format

```

#include <stdio.h>
int fprintf(FILE *stream, const char *format-string, argument-list);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. The behavior might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `fprintf()` function formats and writes a series of characters and values to the output *stream*. The `fprintf()` function converts each entry in *argument-list*, if any, and writes to the stream according to the corresponding format specification in the *format-string*.

The *format-string* has the same form and function as the *format-string* argument for the `printf()` function.

### Return Value

The `fprintf()` function returns the number of bytes that are printed or a negative value if an output error occurs.

For information about `errno` values for `fprintf()`, see “`printf()` — Print Formatted Characters” on page 229.

### Example that uses `fprintf()`

This example sends a line of asterisks for each integer in the array `count` to the file `myfile`. The number of asterisks that are printed on each line corresponds to an integer in the array.

```

#include <stdio.h>

int count [10] = {1, 5, 8, 3, 0, 3, 5, 6, 8, 10};

int main(void)
{
    int i,j;
    FILE *stream;

    stream = fopen("mylib/myfile", "w");
    /* Open the stream for writing */
    for (i=0; i < sizeof(count) / sizeof(count[0]); i++)
    {
        for (j = 0; j < count[i]; j++)
            fprintf(stream,"*");
        /* Print asterisk          */
        fprintf(stream,"\n");
        /* Move to the next line    */
    }
    fclose (stream);
}

/***** Output should be similar to: *****/

*
****
*****
***

***
****
*****
*****
*****
*/

```

### Related Information

- “fscanf() — Read Formatted Data” on page 132
- “fwprintf() — Format Data as Wide Characters and Write to a Stream” on page 143
- “printf() — Print Formatted Characters” on page 229
- “sprintf() — Print Formatted Data to Buffer” on page 352
- “vfprintf() — Print Argument Data to Stream” on page 425
- “vprintf() — Print Argument Data” on page 432
- “vsprintf() — Print Argument Data to Buffer” on page 436
- “<stdio.h>” on page 15

---

## fputc() — Write Character

### Format

```

#include <stdio.h>
int fputc(int c, FILE *stream);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `fputc()` function converts `c` to an unsigned char and then writes `c` to the output *stream* at the current position and advances the file position appropriately. If the stream is opened with one of the append modes, the character is appended to the end of the stream.

The `fputc()` function is identical to `putc()`; it always is defined as a function call; it is never replaced by a macro.

### Return Value

The `fputc()` function returns the character that is written. A return value of EOF indicates an error.

The value of `errno` can be set to:

#### Value Meaning

##### ECONVERT

A conversion error occurred.

##### ENOTWRITE

The file is not open for write operations.

##### EPUTANDGET

A write operation that was not permitted occurred after a read operation.

##### ERECIO

The file is open for record I/O.

##### ESTDERR

`stderr` cannot be opened.

##### ESTDOUT

`stdout` cannot be opened.

##### EIOERROR

A non-recoverable I/O error occurred.

##### EIOECERR

A recoverable I/O error occurred.

The `fputc()` function is not supported for files that are opened with `type=record`.

### Example that uses `fputc()`

This example writes the contents of `buffer` to a file that is called *myfile*.

**Note:** 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] = "abcdefghijklmnopqrstuvwxy";

    if (( stream = fopen("mylib/myfile", "w")) != NULL )
    {
        /* Put buffer into file */
        for ( i = 0; ( i < sizeof(buffer) ) &&
              ((ch = fputc( buffer[i], stream)) != EOF ); ++i );
        fclose( stream );
    }
    else
        perror( "Error opening myfile" );
}

```

### Related Information

- “fgetc() — Read a Character” on page 98
- “putc() – putchar() — Write a Character” on page 239
- “<stdio.h>” on page 15

---

## **\_fputchar - Write Character**

### Format

```

#include <stdio.h>
int _fputchar(int c);

```

**Language Level:** Extension

**Threadsafe:** Yes.

### Description

`_fputchar` writes the single character `c` to the `stdout` stream at the current position. It is equivalent to the following `fputc` call:

```
fputc(c, stdout);
```

For portability, use the ANSI/ISO `fputc` function instead of `_fputchar`.

### Return Value

`_fputchar` returns the character written. A return value of `EOF` indicates that a write error has occurred. Use `ferror` and `feof` to tell whether this is an error condition or the end of the file.

For information about `errno` values for `_fputchar`, see “`fputc() — Write Character`” on page 118.

### Example that uses `_fputchar()`

This example writes the contents of `buffer` to `stdout`:

```

#include <stdio.h>
int main(void)
{
    char buffer[80];
    int i,ch = 1;
    for (i = 0; i < 80; i++)
        buffer[i] = 'c';
    for (i = 0; (i < 80) && (ch != EOF); i++)
        ch = fputc(buffer[i]);
    printf("\n");
    return 0;
}

```

The output should be similar to:

```
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
```

### Related Information:

- “getc() – getchar() — Read a Character” on page 152
- “fputc() — Write Character” on page 118
- “putc() – putchar() — Write a Character” on page 239
- “<stdio.h>” on page 15

## fputs() — Write String

### Format

```

#include <stdio.h>
int fputs(const char *string, FILE *stream);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `fputs()` function copies *string* to the output *stream* at the current position. It does not copy the null character (`\0`) at the end of the string.

### Return Value

The `fputs()` function returns EOF if an error occurs; otherwise, it returns a non-negative value.

The `fputs()` function is not supported for files that are opened with `type=record`.

For information about `errno` values for `fputs()`, see “fputc() — Write Character” on page 118.

### Example that uses fputs()

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("mylib/myfile", "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 */
            perror( "fputs failed" );
    }
    else
        perror( "Error opening myfile" );
}

```

### Related Information

- “fgets() — Read a String” on page 101
- “fputws() — Write Wide-Character String” on page 124
- “gets() — Read a Line” on page 156
- “puts() — Write a String” on page 241
- “<stdio.h>” on page 15

---

## fputc() — Write Wide Character

### Format

```

#include <wchar.h>
#include <stdio.h>
wint_t fputc(wint_t wc, FILE *stream);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. It might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified on the compilation command.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `fputc()` function writes the wide character *wc* to the output stream pointed to by *stream* at the current position. It also advances the file position indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the stream.

Using non-wide-character functions with the `fputc()` function on the same stream will result in undefined behavior. After calling the `fputc()` function, delete the buffer or reposition the stream pointer before calling a read function for the stream. After reading from the stream, delete the buffer or reposition the stream pointer before calling the `fputc()` function, unless EOF has been reached.

| **Note:** If the current locale is changed between subsequent operations on the same stream, undefined  
| results can occur.

### **Return Value**

The `fputc()` function returns the wide character that is written. If a write error occurs, the error indicator for the stream is set, and the `fputc()` function returns WEOF. If an encoding error occurs during conversion from wide character to a multibyte character, `fputc()` sets `errno` to `EILSEQ` and returns WEOF.

For information about `errno` values for `putc()`, see “`putc()` — Write Character” on page 118.

### **Example that uses `fputc()`**

This example opens a file and uses the `fputc()` function to write wide characters to the file.

```

#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
    FILE    *stream;
    wchar_t *wcs = L"A character string.";
    int     i;

    if (NULL == (stream = fopen("fputwc.out", "w")))
    {
        printf("Unable to open: \"fputwc.out\".\n");
        exit(1);
    }

    for (i = 0; wcs[i] != L'\0'; i++) {
        errno = 0;
        if (WEOF == fputwc(wcs[i], stream)) {
            printf("Unable to fputwc() the wide character.\n"
                "wcs[%d] = 0x%.4lx\n", i, wcs[i]);
            if (EILSEQ == errno)
                printf("An invalid wide character was encountered.\n");
            exit(1);
        }
    }
    fclose(stream);
    return 0;

    /*****
    The output file fputwc.out should contain:

    A character string.
    *****/
}

```

### Related Information

- “fgetwc() — Read Wide Character from Stream” on page 102
- “fputc() — Write Character” on page 118
- “fputwc() — Write Wide Character” on page 122
- “putc() – putchar() — Write a Character” on page 239
- “putwchar() — Write Wide Character to stdout” on page 244
- “putwc() — Write Wide Character” on page 242
- “<stdio.h>” on page 15
- “<wchar.h>” on page 18

---

## fputws() — Write Wide-Character String

### Format

```

#include <wchar.h>
#include <stdio.h>
int fputws(const wchar_t *wcs, FILE *stream);

```

**Language Level:** XPG4

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the  
| current locale. It might also be affected by the LC\_UNI\_CTYPE category of the current locale if

| LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation  
| command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation  
| command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified  
| on the compilation command.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `fputws()` function writes the wide-character string *wcs* to a *stream*. It does not write the ending null wide characters.

Using non-wide-character functions with the `fputws()` function on the same stream will result in undefined behavior. After calling the `fputws()` function, flush the buffer or reposition the stream pointer before calling a read function for the stream. After a read operation, flush the buffer or reposition the stream pointer before calling the `fputws()` function, unless EOF has been reached.

| **Note:** If the current locale is changed between subsequent operations on the same stream, undefined  
| results can occur.

### Return Value

The `fputws()` function returns a non-negative value if successful. If a write error occurs, the error indicator for the stream is set, and the `fputws()` function returns -1. If an encoding error occurs in converting the wide characters to multibyte characters, the `fputws()` function sets `errno` to `EILSEQ` and returns -1.

For information about `errno` values for `fputws()`, see “`fputc()` — Write Character” on page 118.

### Example that uses `fputws()`

This example opens a file and writes a wide-character string to the file using the `fgetws()` function.

```

#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 return -1";

    if (NULL == (stream = fopen("fputws.out", "w"))) {
        printf("Unable to open: \"fputws.out\".\n");
        exit(1);
    }

    errno = 0;
    if (EOF == fputws(wcs, stream)) {
        printf("Unable to complete fputws() function.\n");
        if (EILSEQ == errno)
            printf("An invalid wide character was encountered.\n");
        exit(1);
    }
    fclose(stream);
    return 0;

    /*****
    The output file fputws.out should contain:

    This test string should not return -1
    *****/
}

```

## Related Information

- “fgetws() — Read Wide-Character String from Stream” on page 104
- “fputs() — Write String” on page 121
- “fputwc() — Write Wide Character” on page 122
- “puts() — Write a String” on page 241
- “<stdio.h>” on page 15
- “<wchar.h>” on page 18

---

## fread() — Read Items

### Format

```

#include <stdio.h>
size_t fread(void *buffer, size_t size, size_t count, FILE *stream);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `fread()` function reads up to *count* items of *size* length from the input *stream* and stores them in the given *buffer*. The position in the file increases by the number of bytes read.

### Return Value

The `fread()` function returns the number of full items successfully read, which can be less than *count* if an error occurs, or if the end-of-file is met before reaching *count*. If *size* or *count* is 0, the `fread()` function returns zero, and the contents of the array and the state of the stream remain unchanged.

The value of `errno` can be set to:

**Value Meaning**

**EGETANDPUT**

A read operation that was not permitted occurred after a write operation.

**ENOREC**

Record is not found.

**ENOTREAD**

The file is not open for read operations.

**ERECIO**

The file is open for record I/O.

**ESTDIN**

`stdin` cannot be opened.

**ETRUNC**

Truncation occurred on the operation.

**EIOERROR**

A non-recoverable I/O error occurred.

**EIORECERR**

A recoverable I/O error occurred.

Use the `ferror()` and `feof()` functions to distinguish between a read error and an end-of-file.

When using `fread()` for record input, set *size* to 1 and *count* to the maximum expected length of the record, to obtain the number of bytes. If you do not know the record length, you should set *size* to 1 and *count* to a large value. You can read only one record at a time when using record I/O.

**Example that uses `fread()`**

This example attempts to read `NUM_ALPHA` characters from the file *myfile*. 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];

    if (( stream = fopen("mylib/myfile", "r"))!= NULL )
    {
        memset(buffer, 0, sizeof(buffer));
        num = fread( buffer, sizeof( char ), NUM_ALPHA, stream );
        if ( num ) { /* fread success */
            printf( "Number of characters has been read = %i\n", num );
            printf( "buffer = %s\n", buffer );
            fclose( stream );
        }
        else { /* fread failed */
            if ( ferror(stream) ) /* possibility 1 */
                perror( "Error reading myfile" );
            else if ( feof(stream) ) /* possibility 2 */
                perror( "EOF found" );
        }
    }
    else
        perror( "Error opening myfile" );
}

```

### Related Information

- “feof() — Test End-of-File Indicator” on page 95
- “ferror() — Test for Read/Write Errors” on page 95
- “fopen() — Open Files” on page 109
- “fwrite() — Write Items” on page 146
- “<stdio.h>” on page 15

---

## free() — Release Storage Blocks

### Format

```

#include <stdlib.h>
void free(void *ptr);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The free() function frees a block of storage. The *ptr* argument points to a block that is previously reserved with a call to the calloc(), malloc(), realloc(), \_C\_TS\_calloc(), \_C\_TS\_malloc(), \_C\_TS\_realloc(), or \_C\_TS\_malloc64() functions. The number of bytes freed is the number of bytes specified when you reserved (or reallocated, in the case of the realloc() function) the block of storage. If *ptr* is NULL, free() simply returns.

## Notes:

1. All heap storage is associated with the activation group of the calling routine. As such, storage should be allocated and deallocated within the same activation group. It is not valid to allocate heap storage within one activation group and deallocate that storage from a different activation group. For more information about activation groups, see the *ILE Concepts* manual.
2. Attempting to free a block of storage not allocated with `calloc()`, `malloc()`, or `realloc()` (or previously freed storage) can affect the subsequent reserving of storage and lead to undefined results. Storage that is allocated with the ILE bindable API `CEEGETST` can be freed with `free()`.

To use **Teraspace** storage instead of heap storage without changing the C source code, specify the `TERASPACE(*YES *TSIFC)` parameter on the `CRTCMOD` compiler command. This maps the `free()` library function to `_C_TS_free()`, its Teraspace storage counterpart.

**Note:** Whenever an invalid pointer is passed to the `free()` function, a C2M1212 message is signalled. Usually an MCH message associated with the C2M1212 message appears immediately prior to the C2M1212 message in the job log. The MCH message usually has additional information that can be used to debug the problem.

If a C2M1212 message is signalled and the data area `QGPL/QC2M1212` exists, the program stack at the point that the C2M1212 message is signalled is spooled to a spool file. You can create the `QGPL/QC2M1212` data area using the `CRTDTAARA` (Create Data Area) command. You can specify any type and length for the data area.

Following are some of the problems that can cause a C2M1212 message to be signalled:

- A pointer that was never set to point to storage reserved by the `malloc()` function is passed to the `free()` function.
- A pointer was set to point to storage reserved by the `malloc()` function, was subsequently modified, and then is passed to the `free()` function.
- A pointer was set to point to storage reserved by the `malloc()` function, was passed to the `free()` function, and then is passed to the `free()` function..

When a C2M1212 message is generated, the hexadecimal value of the pointer passed to the `free()` function is included as part of the message description. This hexadecimal value can provide clues as to the origin of the problem. The `malloc()` function returns only pointers that end in hexadecimal 0. Any pointer that does not end in hexadecimal 0 was either never set to point to storage reserved by the `malloc()` function or was modified since it was set to point to storage reserved by the `malloc()` function. If the pointer ends in hexadecimal 0, then the cause of the C2M1212 message is uncertain, and the program code that calls `free()` should be examined.

## Return Value

There is no return value.

### Example that uses `free()`

This example uses the `calloc()` function to allocate storage for `x` array elements, and then calls the `free()` function 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 = calloc( num, sizeof( long ))) != NULL )
{
    for ( i = 0; i < num; ++i )          /* put values in array */
        *index++ = i;                  /* using pointer notation */

    free( array );                      /* deallocates array */
}
else
{ /* Out of storage */
    perror( "Error: out of storage" );
    abort();
}
}

```

#### | Related Information

- | • “calloc() — Reserve and Initialize Storage” on page 55
- | • “\_C\_Quickpool\_Debug() — Modify Quick Pool Memory Management Characteristics” on page 66
- | • “\_C\_Quickpool\_Init() — Initialize Quick Pool Memory Management” on page 68
- | • “\_C\_Quickpool\_Report() — Generate Quick Pool Memory Management Report” on page 70
- | • “malloc() — Reserve Storage Block” on page 195
- | • “realloc() — Change Reserved Storage Block Size” on page 264
- | • “<stdlib.h>” on page 17

---

## freopen() — Redirect Open Files

### Format

```

#include <stdio.h>
FILE *freopen(const char *filename, const char *mode, FILE *stream);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `freopen()` function closes the file that is currently associated with *stream* and reassigns *stream* to the file that is specified by *filename*. The `freopen()` function opens the new file associated with *stream* with the given *mode*, which is a character string specifying the type of access requested for the file. You can also use the `freopen()` function to redirect the standard stream files `stdin`, `stdout`, and `stderr` to files that you specify.

For database files, if *filename* is an empty string, the `freopen()` function closes and reopens the stream to the new open mode, rather than reassigning it to a new file or device. You can use the `freopen()` function with no file name specified to change the mode of a standard stream from text to binary without redirecting the stream, for example:

```
fp = freopen("", "rb", stdin);
```

You can use the same method to change the mode from binary back to text.

You cannot use the `freopen()` function with *filename* as an empty string in modules created with `SYSIFCOPT(*IFSIO)`.

## Return Value

The `freopen()` function returns a pointer to the newly opened stream. If an error occurs, the `freopen()` function closes the original file and returns a NULL pointer value.

The value of `errno` can be set to:

### Value Meaning

#### EBADF

The file pointer or descriptor is not valid.

#### EBADMODE

The file mode that is specified is not valid.

#### EBADNAME

The file name that is specified is not valid.

#### ENOENT

No file or library.

#### ENOTOPEN

The file is not open.

#### EIOERROR

A non-recoverable I/O error occurred.

#### EIORECERR

A recoverable I/O error occurred.

## Example that uses `freopen()`

This example closes the *stream1* data stream and reassigns its stream pointer. *stream1* and *stream2* will have the same value, but they will not necessarily have the same value as *stream*.

```
#include <stdio.h>
#define MAX_LEN 100

int main(void)
{
    FILE *stream, *stream1, *stream2;
    char line[MAX_LEN], *result;
    int i;

    stream = fopen("mylib/myfile","r");
    if ((result = fgets(line,MAX_LEN,stream)) != NULL)
        printf("The string is %s\n", result);

    /* Change all spaces in the line to '*'. */
    for (i=0; i<=sizeof(line); i++)
        if (line[i] == ' ')
            line[i] = '*';

    stream1 = stream;
    stream2 = freopen("", "w+", stream1);
    fputs( line, stream2 );
    fclose( stream2);
}
```

## Related Information

- “`fclose()` — Close Stream” on page 91
- “`fopen()` — Open Files” on page 109
- “`<stdio.h>`” on page 15

---

## frexp() — Separate Floating-Point Value

### Format

```
#include <math.h>
double frexp(double x, int *expPtr);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `frexp()` function breaks down the floating-point value  $x$  into a term  $m$  for the mantissa and another term  $n$  for the exponent. It is done such that  $x = m \cdot 2^n$ , and the absolute value of  $m$  is greater than or equal to 0.5 and less than 1.0 or equal to 0. The `frexp()` function stores the integer exponent  $n$  at the location to which `expPtr` points.

### Return Value

The `frexp()` function returns the mantissa term  $m$ . If  $x$  is 0, `frexp()` returns 0 for both the mantissa and exponent. The mantissa has the same sign as the argument  $x$ . The result of the `frexp()` function cannot have a range error.

### Example that uses `frexp()`

This example separates the floating-point value of  $x$ , 16.4, into its mantissa 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 mantissa is %lf and the exponent is %d\n", m, n);
}

/***** Output should be similar to: *****/

The mantissa is 0.512500 and the exponent is 5
*/
```

### Related Information

- “`ldexp()` — Multiply by a Power of Two” on page 178
- “`modf()` — Separate Floating-Point Value” on page 222
- “`<math.h>`” on page 8

---

## fscanf() — Read Formatted Data

### Format

```
#include <stdio.h>
int fscanf (FILE *stream, const char *format-string, argument-list);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. The behavior might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `fscanf()` function reads data from the current position of the specified *stream* into the locations that are given by the entries in *argument-list*, if any. Each entry in *argument-list* must be a pointer to a variable with a type that corresponds to a type specifier in *format-string*.

The *format-string* controls the interpretation of the input fields and has the same form and function as the *format-string* argument for the `scanf()` function.

### Return Value

The `fscanf()` function returns the number of fields that it successfully converted and assigned. The return value does not include fields that the `fscanf()` function read but did not assign.

The return value is EOF if an input failure occurs before any conversion, or the number of input items assigned if successful.

### Example that uses `fscanf()`

This example opens the file *myfile* 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("mylib/myfile", "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);
}

/***** If myfile contains *****/
/***** abcdefghijklmnopqrstuvwxyz 343.2 *****/
/***** expected output is: *****/

string = abcdefghijklmnopqrstuvwxyz
long double = 343
char = .
float = 2.000000
*/

```

### Related Information

- “fprintf() — Write Formatted Data to a Stream” on page 116
- “fwscanf() — Read Data from Stream Using Wide Character” on page 147
- “scanf() — Read Data” on page 330
- “sscanf() — Read Data” on page 355
- “swscanf() — Read Wide Character Data” on page 407
- “wscanf() — Read Data Using Wide-Character Format String” on page 504
- “<stdio.h>” on page 15

---

## fseek() — fseeko() — Reposition File Position

### Format

```

#include <stdio.h>
int fseek(FILE *stream, long int offset, int origin);
int fseeko(FILE *stream, off_t offset, int origin);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Integrated File System Interface:** The fseeko() function is not available when SYSIFCOPT(\*NOIFSIO) is  
| specified on the compilation command.

## Description

The `fseek()` and `fseeko()` functions change the current file position that is associated with *stream* to a new location within the file. The next operation on *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 `fseeko()` function is identical to `fseek()` except that the offset argument is of type `off_t`.

The *origin* must be one of the following constants that are defined in `<stdio.h>`:

### Origin Definition

#### SEEK\_SET

Beginning of file

#### SEEK\_CUR

Current position of file pointer

#### SEEK\_END

End of file

For a binary stream, you can also change the position beyond the end of the file. An attempt to position before the beginning of the file causes an error. If successful, the `fseek()` or `fseeko()` function clears the end-of-file indicator, even when *origin* is `SEEK_END`, and undoes the effect of any preceding the `ungetc()` function on the same stream.

**Note:** For streams opened in text mode, the `fseek()` and `fseeko()` functions have limited use because some system translations (such as those between carriage-return-line-feed and new line) can produce unexpected results. The only `fseek()` and `fseeko()` operations that can be relied upon to work on streams opened in text mode are seeking with an offset of zero relative to any of the origin values, or seeking from the beginning of the file with an offset value returned from a call to the `ftell()` or `ftello()` functions. Calls to the `ftell()` and `ftello()` functions are subject to their restrictions.

### Return Value

The `fseek()` or `fseeko()` function returns 0 if it successfully moves the pointer. A nonzero return value indicates an error. On devices that cannot seek, such as terminals and printers, the return value is nonzero.

The value of `errno` can be set to:

#### Value Meaning

##### EBADF

The file pointer or descriptor is invalid.

##### EBADSEEK

Bad offset for a seek operation.

##### ENODEV

Operation was attempted on a wrong device.

##### ENOTOPEN

The file is not open.

##### ERECIO

The file is open for record I/O.

##### ESTDERR

`stderr` cannot be opened.

**ESTDIN**

stdin cannot be opened.

**ESTDOUT**

stdout cannot be opened.

**EIOERROR**

A non-recoverable I/O error occurred.

**EIORECERR**

A recoverable I/O error occurred.

The `fseek()` and `fseeko()` functions are not supported for files that are opened with `type=record`.

**Example that uses `fseek()`**

This example opens a file `myfile` for reading. After performing input operations, `fseek()` moves the file pointer to the beginning of the file.

```
#include <stdio.h>
#define MAX_LEN 10

int main(void)
{
    FILE *stream;
    char buffer[MAX_LEN + 1];
    int result;
    int i;
    char ch;

    stream = fopen("mylib/myfile", "r");
    for (i = 0; (i < (sizeof(buffer)-1) &&
        ((ch = fgetc(stream)) != EOF) && (ch != '\n')); i++)
        buffer[i] = ch;

    result = fseek(stream, 0L, SEEK_SET); /* moves the pointer to the */
                                        /* beginning of the file */

    if (result == 0)
        printf("Pointer successfully moved to the beginning of the file.\n");
    else
        printf("Failed moving pointer to the beginning of the file.\n");
}
```

**Related Information**

- “`ftell()` — `ftello()` — Get Current Position” on page 138
- “`fgetpos()` — Get File Position” on page 99
- “`fsetpos()` — Set File Position”
- “`rewind()` — Adjust Current File Position” on page 276
- “`ungetc()` — Push Character onto Input Stream” on page 420
- “`fseek()` — `fseeko()` — Reposition File Position” on page 134
- “`<stdio.h>`” on page 15

---

**fsetpos() — Set File Position****Format**

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

## Description

The `fsetpos()` function moves any file position that is associated with *stream* to a new location within the file according to the value pointed to by *pos*. The value of *pos* was obtained by a previous call to the `fgetpos()` library function.

If successful, `fsetpos()` clears the end-of-file indicator, and undoes the effect of any previous `ungetc()` function on the same stream.

After the `fsetpos()` call, the next operation on a stream in update mode can be input or output.

## Return Value

If `fsetpos()` successfully changes the current position of the file, it returns 0. A nonzero return value indicates an error.

The value of `errno` can be set to:

### Value Meaning

#### EBADF

The file pointer or descriptor is invalid.

#### EBADPOS

The position that is specified is not valid.

#### EINVAL

The value specified for the argument is not correct. You might receive this `errno` when you compile your program with `*IFSIO`, and you are working with a file in the QSYS file system. For example, `"/qsys.lib/qtemp.lib/myfile.file/mymem.mbr"`

#### ENODEV

Operation was attempted on a wrong device.

#### ENOPOS

No record at the specified position.

#### ERECIO

The file is open for record I/O.

#### ESTDERR

`stderr` cannot be opened.

#### ESTDIN

`stdin` cannot be opened.

#### ESTDOUT

`stdout` cannot be opened.

#### EIOERROR

A non-recoverable I/O error occurred.

#### EIORECERR

A recoverable I/O error occurred.

The `fsetpos()` function cannot be used for files that are opened with `type=record`. Also, the `fsetpos()` function can only support setting the position to the beginning of the file if:

- your program is compiled with `*IFSIO`, and
- you are working on a file in the QSYS file system.

## Example that uses fsetpos()

This example opens a file mylib/myfile for reading. After performing input operations, fsetpos() moves the file pointer to the beginning of the file and rereads the first byte.

```
#include <stdio.h>

FILE *stream;

int main(void)
{
    int retcode;
    fpos_t pos;
    char ptr[20]; /* existing file 'mylib/myfile' has 20 byte records */
    int i;

    /* Open file, get position of file pointer, and read first record */

    stream = fopen("mylib/myfile", "rb");
    fgetpos(stream,Point-of-Sale);
    if (!fread(ptr,sizeof(ptr),1,stream))
        perror("fread error");
    else printf("1st record: %s\n", ptr);

    /* Perform another read operation on the second record      */
    /* - the value of 'pos' changes                               */
    if (!fread(ptr,sizeof(ptr),1,stream))
        perror("fread error");
    else printf("2nd record: %s\n", ptr);

    /* Re-set pointer to start of file and re-read first record */
    fsetpos(stream,Point-of-Sale);
    if (!fread(ptr,sizeof(ptr),1,stream))
        perror("fread error");
    else printf("1st record again: %s\n", ptr);

    fclose(stream);
}
```

## Related Information

- “fgetpos() — Get File Position” on page 99
- “fseek() — fseeko() — Reposition File Position” on page 134
- “ftell() — ftello() — Get Current Position”
- “rewind() — Adjust Current File Position” on page 276
- “<stdio.h>” on page 15

---

## ftell() — ftello() — Get Current Position

### Format

```
#include <stdio.h>
long int ftell(FILE *stream);
off_t ftello(FILE *stream);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Integrated File System Interface:** The ftello() function is not available when SYSIFCOPT(\*NOIFSIO) is  
| specified on the compilation command.

## Description

The `ftell()` and `ftello()` functions find the current position of the file associated with *stream*. For a fixed-length binary file, the value that is returned is an offset relative to the beginning of the *stream*.

For file in the QSYS library system, the `ftell()` and `ftello()` functions return a relative value for fixed-format binary files and an encoded value for other file types. This encoded value must be used in calls to the `fseek()` and `fseeko()` functions to positions other than the beginning of the file.

## Return Value

The `ftell()` and `ftello()` functions return the current file position. On error, `ftell()` and `ftello()` return `-1`, cast to `long` and `off_t` respectively, and set `errno` to a nonzero value.

The value of `errno` can be set to:

### Value Meaning

#### ENODEV

Operation was attempted on a wrong device.

#### ENOTOPEN

The file is not open.

#### ENUMMBRS

The file is open for multi-member processing.

#### ENUMRECS

Too many records.

#### ERECIO

The file is open for record I/O.

#### ESTDERR

`stderr` cannot be opened.

#### ESTDIN

`stdin` cannot be opened.

#### ESTDOUT

`stdout` cannot be opened.

#### EIOERROR

A non-recoverable I/O error occurred.

#### EIOECERR

A recoverable I/O error occurred.

The `ftell()` and `ftello()` functions are not supported for files that are opened with `type=record`.

### Example that uses `ftell()`

This example opens the file `mylib/myfile` for reading. It reads enough characters to fill half of the buffer and prints out the position in the stream and the buffer.

```

#include <stdio.h>

#define NUM_ALPHA 26
#define NUM_CHAR 6

int main(void)
{
    FILE * stream;
    int i;
    char ch;

    char buffer[NUM_ALPHA];
    long position;

    if (( stream = fopen("mylib/myfile", "r") ) != NULL )
    {
        /* read into buffer */
        for ( i = 0; ( i < NUM_ALPHA/2 ) && ((buffer[i] = fgetc(stream)) != EOF ); ++i )
            if (i==NUM_CHAR-1) /* We want to be able to position the */
                                /* file pointer to the character in */
                                /* position NUM_CHAR */
                position = ftell(stream);

        buffer[i] = '\0';
    } printf("Current file position is %d\n", position);
    printf("Buffer contains: %s\n", buffer);
}

```

### Related Information

- “fseek() — fseeko() — Reposition File Position” on page 134
- “fgetpos() — Get File Position” on page 99
- “fopen() — Open Files” on page 109
- “fsetpos() — Set File Position” on page 136
- “ftell() — ftello() — Get Current Position” on page 138
- “<stdio.h>” on page 15

---

## fwide() — Determine Stream Orientation

### Format

```

#include <stdio.h>
#include <wchar.h>
int fwide(FILE *stream, int mode);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command.

| **Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified on the compilation command.

### Description

The `fwide()` function determines the orientation of the stream pointed to by `stream`. If `mode` is greater than 0, the `fwide()` function first attempts to make the stream wide oriented. If `mode` is less than 0, the `fwide()` function first attempts to make the stream byte oriented. Otherwise, `mode` is 0, and the `fwide()` function does not alter the orientation of the stream.

| **Note:** If the orientation of the stream has already been determined, the `fwide()` function does not change  
| it.

### **Return Value**

If, after the call, the stream has wide orientation, the `fwide()` function returns a value greater than 0. If the stream has byte orientation, it returns a value less than 0. If the stream has no orientation, it returns 0.

**Example that uses `fwide()`**

```

#include <stdio.h>
#include <math.h>
#include <wchar.h>

void check_orientation(FILE *stream)
{
    int rc;
    rc = fwide(stream,0);    /* check the orientation */
    if (rc<0) {
        printf("Stream has byte orientation.\n");
    } else if (rc>0) {
        printf("Stream has wide orientation.\n");
    } else {
        printf("Stream has no orientation.\n");
    }
    return;
}

int main(void)
{
    FILE *stream;
    /* Demonstrate that fwide can be used to set the orientation,
       but cannot change it once it has been set.  */
    stream = fopen("test.dat","w");
    printf("After opening the file: ");
    check_orientation(stream);
    fwide(stream, -1);      /* Make the stream byte oriented */
    printf("After fwide(stream, -1): ");
    check_orientation(stream);
    fwide(stream, 1);      /* Try to make the stream wide oriented */
    printf("After fwide(stream, 1): ");
    check_orientation(stream);
    fclose(stream);
    printf("Close the stream\n");
    /* Check that a wide character output operation sets the orientation
       as expected.  */
    stream = fopen("test.dat","w");
    printf("After opening the file: ");
    check_orientation(stream);
    fwprintf(stream, L"pi = %.5f\n", 4* atan(1.0));
    printf("After fwprintf( ): ");
    check_orientation(stream);
    fclose(stream);
    return 0;
    /*****
       The output should be similar to :
       After opening the file: Stream has no orientation.
       After fwide(stream, -1): Stream has byte orientation.
       After fwide(stream, 1): Stream has byte orientation.
       Close the stream
       After opening the file: Stream has no orientation.
       After fwprintf( ): Stream has wide orientation.
    *****/
}

```

## Related Information

- “fgetwc() — Read Wide Character from Stream” on page 102
- “fgetws() — Read Wide-Character String from Stream” on page 104
- “fputwc() — Write Wide Character” on page 122
- “fputws() — Write Wide-Character String” on page 124
- “<stdio.h>” on page 15
- “<wchar.h>” on page 18

---

## fwprintf() — Format Data as Wide Characters and Write to a Stream

### Format

```
#include <stdio.h>
#include <wchar.h>
int fwprintf(FILE *stream, const wchar_t *format, argument-list);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

- | **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale, and might also be affected by the LC\_UNI\_CTYPE and LC\_UNI\_NUMERIC categories of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.
- | **Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified on the compilation command.
- | **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `fwprintf()` function writes output to the stream pointed to by *stream*, under control of the wide string pointed to by *format*. The format string specifies how subsequent arguments are converted for output.

The `fwprintf()` function converts each entry in *argument-list* according to the corresponding wide-character format specifier in *format*.

If insufficient arguments exist for the format, the behavior is undefined. If the format is exhausted while arguments remain, the `fwprintf()` function evaluates the excess arguments, but otherwise ignores them. The `fwprintf()` function returns when it encounters the end of the format string.

The format comprises zero or more directives: ordinary wide characters (not %) and conversion specifications. Conversion specifications are processed as if they were replaced in the format string by wide-character strings. The wide-character strings are the result of fetching zero or more subsequent arguments and then converting them, if applicable, according to the corresponding conversion specifier. The `fwprintf()` function then writes the expanded wide-character format string to the output stream.

The format for the `fwprintf()` function has the same form and function as the format string for `printf()`, with the following exceptions:

- %c (without an l prefix) converts an integer argument to `wchar_t`, as if by calling the `btowc()` function.
- %s (without an l prefix) converts an array of multibyte characters to an array of `wchar_t`, as if by calling the `mbrtowc()` function. The array is written up to, but not including, the terminating null character, unless the precision specifies a shorter output.
- %ls and %S write an array of `wchar_t`. The array is written up to, but not including, the ending null character, unless the precision specifies a shorter output.
- | • Any width or precision specified for %c, %s, %ls, and %S indicates the number of characters rather than the number of bytes.

If a conversion specification is invalid, the behavior is undefined.

| If any argument is, or points to, a union or an aggregate (except for an array of char type using %s  
| conversion, an array of wchar\_t type using %ls conversion, or a pointer using %p conversion), the  
| behavior is undefined.

In no case does a nonexistent, or small field width, cause truncation of a field; if the conversion result is wider than the field width, the field is expanded to contain the conversion result.

| **Note:** When you write wide characters, the file should be opened in binary mode, or opened with the  
| `o_ccsid` or `codepage` parameters. This ensures that no conversions occur on the wide characters.

## Return Value

The `fwprintf()` function returns the number of wide characters transmitted. If an output error occurred, it returns a negative value.

## Example that uses `fwprintf()`

```
#include <stdio.h>
#include <wchar.h>
#include <locale.h>
int count [10] = {1, 5, 8, 3, 0, 3, 5, 6, 8, 10};
int main(void)
{
    int i,j;
    FILE *stream;          /* Open the stream for writing */
    if (NULL == (stream = fopen("/QSYS.LIB/LIB.LIB/WCHAR.FILE/WCHAR.MBR","wb")))
        perror("fopen error");
    for (i=0; i < sizeof(count) / sizeof(count[0]); i++)
    {
        for (j = 0; j < count[i]; j++)
            fwprintf(stream, L"*");    /* Print asterisk      */
            fwprintf(stream, L"\n");   /* Move to the next line */
    }
    fclose (stream);
}
/* The member WCHAR of file WCHAR will contain:
*
*****
*****
***
***
*****
*****
*****
*****
*****
*/
```

## Unicode example that uses `fwprintf()`

```

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
/* This program is compile LOCALETYPE(*LOCALEUCS2) and          */
/* SYSIFCOPT(*IFSIO)                                           */
int main(void)
{
    FILE *stream;
    wchar_t wc = 0x0058;    /* UNICODE X */
    char c1 = 'c';
    char *s1 = "123";
    wchar_t ws[4];
    setlocale(LC_ALL,
        "/QSYS.LIB/ĒN_US.LOCALE"); /* a CCSID 37 locale */
    ws[0] = 0x0041;        /* UNICODE A */
    ws[1] = (wchar_t)0x0042; /* UNICODE B */
    ws[2] = (wchar_t)0x0043; /* UNICODE C */
    ws[3] = (wchar_t)0x0000;

    stream = fopen("myfile.dat", "wb+");

    /* lc and ls take wide char as input and just copies then */
    /* to the file. So the file would look like this          */
    /* after the below fprintf statement:                      */
    /* 0058002000200020004100420043                          */
    /* 0020 is UNICODE blank                                  */

    fprintf(stream, L"%lc %ls",wc,ws);
    /* c and s take multibyte as input and produce UNICODE */
    /* In this case c1 and s1 are CCSID 37 characters based */
    /* on the setlocale above. So the characters are        */
    /* converted from CCSID 37 to UNICODE and will look     */
    /* like this in hex after the following fprintf          */
    /* statement: 0063002000200020003100320033             */
    /* 0063 is a UNICODE c 0031 is a UNICODE 1 and so on  */

    fprintf(stream, L"%c %s",c1,s1);

    /* Now lets try width and precision. 6ls means write */
    /* 6 wide characters so we will pad with 3 UNICODE */
    /* blanks and %.2s means write no more then 2 wide */
    /* characters. So we get an output that looks like */
    /* this: 00200020002000410042004300310032             */

    fprintf(stream, L"%6ls%.2s",ws,s1);
}

```

## Related Information

- “fprintf() — Write Formatted Data to a Stream” on page 116
- “printf() — Print Formatted Characters” on page 229
- “vfprintf() — Print Argument Data to Stream” on page 425
- “vprintf() — Print Argument Data” on page 432
- “btowc() — Convert Single Byte to Wide Character” on page 53
- “mbrtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 201
- “vfwprintf() — Format Argument Data as Wide Characters and Write to a Stream” on page 428
- “vswprintf() — Format and Write Wide Characters to Buffer” on page 439
- “wprintf() — Format Data as Wide Characters and Print” on page 503
- “<stdarg.h>” on page 13
- “<wchar.h>” on page 18

---

## **fwrite()** — Write Items

### **Format**

```
#include <stdio.h>
size_t fwrite(const void *buffer, size_t size, size_t count,
              FILE *stream);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### **Description**

The `fwrite()` function writes up to *count* items, each of *size* bytes in length, from *buffer* to the output *stream*.

### **Return Value**

The `fwrite()` function returns the number of full items successfully written, which can be fewer than *count* if an error occurs.

When using `fwrite()` for record output, set *size* to 1 and *count* to the length of the record to obtain the number of bytes written. You can only write one record at a time when using record I/O.

The value of `errno` can be set to:

#### **Value Meaning**

##### **ECONVERT**

A conversion error occurred.

##### **ENOTWRITE**

The file is not open for write operations.

**EPAD** Padding occurred on a write operation.

##### **EPUTANDGET**

An illegal write operation occurred after a read operation.

##### **ESTDERR**

`stderr` cannot be opened.

##### **ESTDIN**

`stdin` cannot be opened.

##### **ESTDOUT**

`stdout` cannot be opened.

##### **ETRUNC**

Truncation occurred on I/O operation.

##### **EIOERROR**

A non-recoverable I/O error occurred.

##### **EIOECERR**

A recoverable I/O error occurred.

### **Example that uses fwrite()**

This example writes `NUM` long integers to a stream in binary format.

```

#include <stdio.h>
#define NUM 100

int main(void)
{
    FILE *stream;
    long list[NUM];
    int numwritten;
    int i;

    stream = fopen("MYLIB/MYFILE", "w+b");

    /* assign values to list[] */
    for (i=0; i<=NUM; i++)
        list[i]=i;

    numwritten = fwrite(list, sizeof(long), NUM, stream);
    printf("Number of items successfully written = %d\n", numwritten);
}

```

### Related Information

- “fopen() — Open Files” on page 109
- “fread() — Read Items” on page 126
- “<stdio.h>” on page 15

---

## fwscanf() — Read Data from Stream Using Wide Character

### Format

```

#include <stdio.h>
#include <wchar.h>
int fwscanf(FILE *stream, const wchar_t *format, argument-list);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. It might also be affected by the LC\_UNI\_CTYPE and LC\_UNI\_NUMERIC categories of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

**Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified on the compilation command.

**Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `fwscanf()` function reads input from the stream pointed to by *stream*, under control of the wide string pointed to by *format*. The format string specifies the admissible input sequences and how they are to be converted for assignment. To receive the converted input, the `fwscanf()` function uses subsequent arguments as pointers to the objects.

Each argument in *argument-list* must point to a variable with a type that corresponds to a type specifier in *format*.

If insufficient arguments exist for the format, the behavior is undefined. If the format is exhausted while arguments remain, the `fwscanf()` function evaluates the excess arguments, but otherwise ignores them.

The format consists of zero or more directives: one or more white-space wide characters; an ordinary wide character (neither `%` nor a white-space wide character); or a conversion specification. Each conversion specification is introduced by a `%`.

The format has the same form and function as the format string for the `scanf()` function, with the following exceptions:

- `%c` (with no `l` prefix) converts one or more `wchar_t` characters (depending on precision) to multibyte characters, as if by calling `wcrtomb()`.
- `%lc` and `%C` convert one or more `wchar_t` characters (depending on precision) to an array of `wchar_t`.
- `%s` (with no `l` prefix) converts a sequence of non-white-space `wchar_t` characters to multibyte characters, as if by calling the `wcrtomb()` function. The array includes the ending null character.
- `%ls` and `%S` copy an array of `wchar_t`, including the ending null wide character, to an array of `wchar_t`.

| If the data is from `stdin`, and `stdin` has not been overridden, the data is assumed to be in the CCSID of  
| the job. The data is converted as required by the format specifications. If the file that is being read is not  
| opened with file mode `rb`, then invalid conversion can occur.

If a conversion specification is invalid, the behavior is undefined. If the `fwscanf()` function encounters end-of-file during input, conversion is ended. If end-of-file occurs before the `fwscanf()` function reads any characters matching the current directive (other than leading white space, where permitted), execution of the current directive ends with an input failure. Otherwise, unless execution of the current directive terminates with a matching failure, execution of the following directive (other than `%n`, if any) ends with an input failure.

The `fwscanf()` function leaves trailing white space (including new-line wide characters) unread, unless matched by a directive. You cannot determine the success of literal matches and suppressed assignments other than through the `%n` directive.

### Return Value

The `fwscanf()` function returns the number of input items assigned, which can be fewer than provided for, in the event of an early matching failure.

If an input failure occurs before any conversion, the `fwscanf()` function returns EOF.

### Example that uses `fwscanf()`

This example opens the file **myfile.dat** for input, and then scans this file for a string, a long integer value, a character, and a floating-point value.

```

#include <stdio.h>
#include <wchar.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");

    /* Read data from file. */

    fwscanf(stream, L"%s", &s[0]);
    fwscanf(stream, L"%ld", &l);
    fwscanf(stream, L"%c", &c);
    fwscanf(stream, L"%f", &fp);

    printf("string = %s\n", s);
    printf("long integer = %ld\n", l);
    printf("char = %c\n", c);
    printf("float = %f\n", fp);
    return 0;

    /*****
    If myfile.dat contains:
    abcdefghijklmnopqrstuvwxyz 343.2.

    The output should be:

    string = abcdefghijklmnopqrstuvwxyz
    long integer = 343
    char = .
    float = 2.000000
    *****/
}

```

### Unicode example that uses fwscanf()

This example reads a Unicode string from **unicode.dat** and prints it to the screen. The example is compiled with `LOCALETYPE(*LOCALEUCS2) SYSIFCOPT(*IFSIO)`:

```

#include <stdio.h>
#include <wchar.h>
#include <locale.h>
void main(void)
{
FILE *stream;
wchar_t buffer[20];
stream=fopen("unicode.dat","rb");

fwscanf(stream,L"%ls", buffer);
wprintf(L"The string read was :%ls\n",buffer);

fclose(stream);
}

/* If the input in unicode.dat is :
ABC
and ABC is in unicode which in hex would be 0x0041, 0x0042, 0x0043
then the output will be similar to:
The string read was :ABC
*/

```

### Related Information

- “fwscanf() — Read Formatted Data” on page 132
- “fwprintf() — Format Data as Wide Characters and Write to a Stream” on page 143
- “scanf() — Read Data” on page 330
- “swprintf() — Format and Write Wide Characters to Buffer” on page 406
- “swscanf() — Read Wide Character Data” on page 407
- “wscanf() — Read Data Using Wide-Character Format String” on page 504
- “<stdio.h>” on page 15
- “<wchar.h>” on page 18

---

## gamma() — Gamma Function

### Format

```

#include <math.h>
double gamma(double x);

```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

### Description

The gamma() function 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 argument  $x$  must be a positive real value.

### Return Value

The gamma() function returns the value of  $\ln(|G(x)|)$ . If  $x$  is a negative value, `errno` is set to `EDOM`. If the result causes an overflow, gamma() returns `HUGE_VAL` and sets `errno` to `ERANGE`.

### Example that uses gamma()

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(%4.21f) is %7.2e\n", x, g_at_x);
}

/***** Output should be similar to: *****/

The value of G(42.00) is 3.35e+49
*/
```

### Related Information

- “Bessel Functions” on page 50
- “erf() – erfc() — Calculate Error Functions” on page 87
- “<math.h>” on page 8

---

## **\_gcvt - Convert Floating-Point to String**

### Format

```
#include <stdlib.h>
char *_gcvt(double value, int ndec, char *buffer);
```

**Note:** The \_gcvt function is supported only for C++, not for C.

**Language Level:** Extension

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

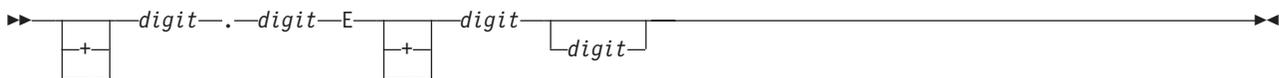
\_gcvt() converts a floating-point value to a character string pointed to by *buffer*. The *buffer* should be large enough to hold the converted value and a null character (\0) that \_gcvt() automatically adds to the end of the string. There is no provision for overflow.

\_gcvt() attempts to produce *ndec* significant digits in FORTRAN F format. Failing that, it produces *ndec* significant digits in FORTRAN E format. Trailing zeros might be suppressed in the conversion if they are insignificant.

A FORTRAN F number has the following format:



A FORTRAN E number has the following format:



| `_gcvt` also converts infinity values to the string INFINITY.

### Return Value

`_gcvt()` returns a pointer to the string of digits. If it cannot allocate memory to perform the conversion, `_gcvt()` returns an empty string and sets `errno` to `ENOMEM`.

### Example that uses `_gcvt()`

This example converts the value `-3.1415e3` to a character string and places it in the character array `buffer1`.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char buffer1[10];
    _gcvt(-3.1415e3, 7, buffer1);
    printf("The first result is %s \n", buffer1);
    return 0;
}
```

The output should be:

```
The first result is -3141.5
```

### | Related Information:

| • “`<stdlib.h>`” on page 17

---

## getc() – getchar() — Read a Character

### Format

```
#include <stdio.h>
int getc(FILE *stream);
int getchar(void);
```

**Language Level:** ANSI

**Threadsafe:** No. `#undef getc` or `#undef getchar` allows the `getc` or `getchar` function to be called instead of the macro version of these functions. The functions are threadsafe.

### Description

The `getc()` function reads a single character from the current *stream* position and advances the *stream* position to the next character. The `getchar()` function is identical to `getc(stdin)`.

The difference between the `getc()` and `fgetc()` functions is that `getc()` can be implemented so that its arguments can be evaluated multiple times. Therefore, the *stream* argument to `getc()` should not be an expression with side effects.

### Return Value

The `getc()` and `getchar()` functions return the character read. A return value of EOF indicates an error or end-of-file condition. Use `ferror()` or `feof()` to determine whether an error or an end-of-file condition occurred.

The value of `errno` can be set to:

#### **Value Meaning**

##### **EBADF**

The file pointer or descriptor is not valid.

##### **ECONVERT**

A conversion error occurred.

##### **EGETANDPUT**

An illegal read operation occurred after a write operation.

##### **EIOERROR**

A non-recoverable I/O error occurred.

##### **EIORECERR**

A recoverable I/O error occurred.

The `getc()` and `getchar()` functions are not supported in record mode.

#### **Example that uses `getc()`**

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 );
}
```

#### **Related Information**

- “`fgetc()` — Read a Character” on page 98
- “`fgetwc()` — Read Wide Character from Stream” on page 102
- “`gets()` — Read a Line” on page 156
- “`getwc()` — Read Wide Character from Stream” on page 157

- “getwchar() — Get Wide Character from stdin” on page 159
- “putc() – putchar() — Write a Character” on page 239
- “ungetc() — Push Character onto Input Stream” on page 420
- “<stdio.h>” on page 15

---

## getenv() — Search for Environment Variables

### Format

```
#include <stdlib.h>
char *getenv(const char *varname);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

- | **Job CCSID Interface:** All character data sent to this function is expected to be in the CCSID of the job.
- | All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and
- | Locales” on page 524 for more information.

### Description

The `getenv()` function searches the list of environment variables for an entry corresponding to *varname*.

### Return Value

The `getenv()` function returns a pointer to the string containing the value for the specified *varname* in the current environment. If `getenv()` cannot find the environment string, `NULL` is returned, and `errno` is set to indicate the error.

### Example that uses `getenv()`

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

/* Where the environment variable 'PATH' is set to a value. */

int main(void)
{
    char *pathvar;

    pathvar = getenv("PATH");
    printf("pathvar=%s",pathvar);
}
```

### Related Information

- “<stdlib.h>” on page 17
- “putenv() — Change/Add Environment Variables” on page 240
- Environment Variable APIs in the APIs topic in the i5/OS Information Center.

---

## \_GetExcData() — Get Exception Data

### Format

```
#include <signal.h>
void _GetExcData(_INTRPT_Hndlr_Parms_T *parms);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

- | **Job CCSID Interface:** All character data sent to this function is expected to be in the CCSID of the job.
- | All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and Locales” on page 524 for more information.

### Description

The `_GetExcData()` function returns information about the current exception from within a C signal handler. The caller of the `_GetExcData()` function must allocate enough storage for a structure of type `_INTRPT_Hndlr_Parms_T`. If the `_GetExcData()` function is called from outside a signal handler, the storage pointed to by *parms* is not updated.

This function is not available when `SYSIFCOPT(*ASYNCSIGNAL)` is specified on the compilation commands. When `SYSIFCOPT(*ASYNCSIGNAL)` is specified, a signal handler established with the `ILE C signal()` function has no way to access any exception information that might have caused the signal handler to be invoked. An extended signal handler established with the `sigaction()` function, however, does have access to this exception information. The extended signal handler has the following function prototype:

```
void func( int signo, siginfo_t *info, void *context )
```

The exception information is appended to the `siginfo_t` structure, which is then passed as the second parameter to the extended signal handler.

The `siginfo_t` structure is defined in `signal.h`. The exception-related data follows the `si_sigdata` field in the `siginfo_t` structure. You can address it from the `se_data` field of the `sigdata_t` structure.

The format of the exception data appended to the `siginfo_t` structure is defined by the `_INTRPT_Hndlr_Parms_T` structure in `except.h`.

### Return Value

There is no return value.

### Example that uses `_GetExcData()`

This example shows how exceptions from MI library functions can be monitored and handled using a signal handling function. The signal handler `my_signal_handler` is registered before the `rs1vsp()` function signals a 0x2201 exception. When a `SIGSEGV` signal is raised, the signal handler is called. If an 0x2201 exception occurred, the signal handler calls the `QUSRCRTS` API to create a space.

```

#include <signal.h>
#include <QSYSINC/MIH/RSLVSP>
#include <QSYSINC/H/QUSCRTUS>
#include <string.h>

#define CREATION_SIZE 65500

void my_signal_handler(int sig) {

    _INTRPT_Hndlr_Parms_T excp_data;
    int error_code = 0;

    /* Check the message id for exception 0x2201 */
    _GetExcData(&excp_data);

    if (!memcmp(excp_data.Msg_Id, "MCH3401", 7))
        QUSCRTUS("MYSPACE QTEMP ",
                "MYSPACE ",
                CREATION_SIZE,
                "\0",
                "*ALL ",
                "MYSPACE example for Programmer's Reference ",
                "*YES ",
                &error_code);
}

```

### Related Information

- “signal() — Handle Interrupt Signals” on page 346
- “<except.h>” on page 4

---

## gets() — Read a Line

### Format

```

#include <stdio.h>
char *gets(char *buffer);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `gets()` function reads a line from the standard input stream `stdin` and stores it in *buffer*. The line consists of all characters up to but not including the first new-line character (`\n`) or EOF. The `gets()` function then replaces the new-line character, if read, with a null character (`\0`) before returning the line.

### Return Value

If successful, the `gets()` function returns its argument. A NULL pointer return value indicates an error, or an end-of-file condition with no characters read. Use the `ferror()` function or the `feof()` function to determine which of these conditions occurred. If there is an error, the value that is stored in *buffer* is undefined. If an end-of-file condition occurs, *buffer* is not changed.

### Example that uses gets()

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("Please enter a string:\n");
    if ((result = gets(line)) != NULL)
        printf("The string is: %s\n", line);
    else if (ferror(stdin))
        perror("Error");
}

```

### Related Information

- “fgets() — Read a String” on page 101
- “fgetws() — Read Wide-Character String from Stream” on page 104
- “feof() — Test End-of-File Indicator” on page 95
- “ferror() — Test for Read/Write Errors” on page 95
- “fputs() — Write String” on page 121
- “getc() – getchar() — Read a Character” on page 152
- “puts() — Write a String” on page 241
- “<stdio.h>” on page 15

---

## getwc() — Read Wide Character from Stream

### Format

```

#include <stdio.h>
#include <wchar.h>
wint_t getwc(FILE *stream);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. It might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified on the compilation command.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `getwc()` function reads the next multibyte character from stream, converts it to a wide character, and advances the associated file position indicator for stream.

The `getwc()` function is equivalent to the `fgetwc()` function except that, if it is implemented as a macro, it can evaluate *stream more* than once. Therefore, the argument should never be an expression with side effects.

- | If the current locale is changed between subsequent read operations on the same stream, undefined
- | results can occur. Using non-wide-character functions with the `getwc()` function on the same stream
- | results in undefined behavior.

After calling the `getwc()` function, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless EOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling the `getwc()` function.

### **Return Value**

The `getwc()` function returns the next wide character from the input stream, or WEOF. If an error occurs, the `getwc()` function sets the error indicator. If the `getwc()` function encounters the end-of-file, it sets the EOF indicator. If an encoding error occurs during conversion of the multibyte character, the `getwc()` function sets `errno` to `EILSEQ`.

Use the `ferror()` or `feof()` functions to determine whether an error or an EOF condition occurred. 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.

For information about `errno` values for `getwc()`, see “`fgetwc()` — Read Wide Character from Stream” on page 102.

**Example that uses `getwc()`**

```

#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
    FILE *stream;
    wint_t wc;

    if (NULL == (stream = fopen("getwc.dat", "r"))) {
        printf("Unable to open: \"getwc.dat\"\n");
        exit(1);
    }

    errno = 0;
    while (WEOF != (wc = getwc(stream)))
        printf("wc = %lc\n", wc);

    if (EILSEQ == errno) {
        printf("An invalid wide character was encountered.\n");
        exit(1);
    }
    fclose(stream);
    return 0;

    /*****
    Assuming the file getwc.dat contains:

    Hello world!

    The output should be similar to:

    wc = H
    wc = e
    wc = l
    wc = l
    wc = o
    :
    *****/
}

```

### Related Information

- “fgetwc() — Read Wide Character from Stream” on page 102
- “getwchar() — Get Wide Character from stdin”
- “getc() – getchar() — Read a Character” on page 152
- “putwc() — Write Wide Character” on page 242
- “ungetwc() — Push Wide Character onto Input Stream” on page 422
- “<stdio.h>” on page 15
- “<wchar.h>” on page 18

---

## getwchar() — Get Wide Character from stdin

### Format

```

#include <wchar.h>
wint_t getwchar(void);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

- | **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. It might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.
- | **Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified on the compilation command.
- | **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `getwchar()` function reads the next multibyte character from `stdin`, converts it to a wide character, and advances the associated file position indicator for `stdin`. A call to the `getwchar()` function is equivalent to a call to `getwc(stdin)`.

- | If the current locale is changed between subsequent read operations on the same stream, undefined results can occur. Using non-wide-character functions with the `getwchar()` function on `stdin` results in undefined behavior.

### Return Value

The `getwchar()` function returns the next wide character from `stdin` or `WEOF`. If the `getwchar()` function encounters EOF, it sets the EOF indicator for the stream and returns `WEOF`. If a read error occurs, the error indicator for the stream is set, and the `getwchar()` function returns `WEOF`. If an encoding error occurs during the conversion of the multibyte character to a wide character, the `getwchar()` function sets `errno` to `EILSEQ` and returns `WEOF`.

Use the `ferror()` or `feof()` functions to determine whether an error or an EOF condition occurred. 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.

For information about `errno` values for `getwchar()`, see “`fgetwc()` — Read Wide Character from Stream” on page 102.

### Example that uses `getwchar()`

This example uses the `getwchar()` to read wide characters from the keyboard, then prints the wide characters.

```

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

int main(void)
{
    wint_t wc;

    errno = 0;
    while (WEOF != (wc = getwchar()))
        printf("wc = %lc\n", wc);

    if (EILSEQ == errno) {
        printf("An invalid wide character was encountered.\n");
        exit(1);
    }
    return 0;

    /*****
    Assuming you enter: abcde

    The output should be:

    wc = a
    wc = b
    wc = c
    wc = d
    wc = e
    *****/
}

```

## Related Information

- “fgetc() — Read a Character” on page 98
- “fgetwc() — Read Wide Character from Stream” on page 102
- “fgetws() — Read Wide-Character String from Stream” on page 104
- “getc() – getchar() — Read a Character” on page 152
- “getwc() — Read Wide Character from Stream” on page 157
- “ungetwc() — Push Wide Character onto Input Stream” on page 422
- “<wchar.h>” on page 18

---

## gmtime() — Convert Time

### Format

```

#include <time.h>
struct tm *gmtime(const time_t *time);

```

**Language Level:** ANSI

**Threadsafe:** No. Use `gmtime_r()` instead.

### Description

The `gmtime()` function breaks down the *time* value, in seconds, and stores it in a *tm* structure, defined in `<time.h>`. The value *time* is usually obtained by a call to the `time()` function.

The fields of the *tm* structure include:

**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.

### Return Value

The `gmtime()` function returns a pointer to the resulting `tm` structure.

### Notes:

1. The range (0-61) for `tm_sec` allows for as many as two leap seconds.
2. The `gmtime()` and `localtime()` functions can use a common, statically allocated buffer for the conversion. Each call to one of these functions might alter the result of the previous call.
3. Calendar time is the number of seconds that have elapsed since EPOCH, which is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).

### Example that uses `gmtime()`

This example uses the `gmtime()` function to adjust a `time_t` representation to a Coordinated Universal Time character string, and then converts it to a printable string using the `asctime()` function.

```
#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 should be similar to: *****/

Coordinated Universal Time is Wed Aug 18 21:01:44 1993
*/
```

### Related Information

- “`asctime()` — Convert Time to Character String” on page 39
- “`asctime_r()` — Convert Time to Character String (Restartable)” on page 41

- “`ctime()` — Convert Time to Character String” on page 71
- | • “`ctime64()` — Convert Time to Character String” on page 73
- | • “`ctime64_r()` — Convert Time to Character String (Restartable)” on page 76
- “`ctime_r()` — Convert Time to Character String (Restartable)” on page 74
- | • “`gmtime64()` — Convert Time”
- | • “`gmtime64_r()` — Convert Time (Restartable)” on page 167
- “`gmtime_r()` — Convert Time (Restartable)” on page 165
- “`localtime()` — Convert Time” on page 185
- | • “`localtime64()` — Convert Time” on page 187
- | • “`localtime64_r()` — Convert Time (Restartable)” on page 189
- “`localtime_r()` — Convert Time (Restartable)” on page 188
- “`mktime()` — Convert Local Time” on page 218
- | • “`mktime64()` — Convert Local Time” on page 220
- “`setlocale()` — Set Locale” on page 339
- “`time()` — Determine Current Time” on page 411
- | • “`time64()` — Determine Current Time” on page 412
- “`<time.h>`” on page 18

---

## | **gmtime64() — Convert Time**

### | **Format**

```
| #include <time.h>
| struct tm *gmtime64(const tim64_t *time);
```

| **Language Level:** ILE C Extension

| **Threadsafe:** No. Use `gmtime64_r()` instead.

### | **Description**

| The `gmtime64()` function breaks down the *time* value, in seconds, and stores it in a *tm* structure, defined in `<time.h>`. The value *time* is usually obtained by a call to the `time64()` function.

| The fields of the *tm* structure include:

#### | **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.

### | **Return Value**

| The `gmtime64()` function returns a pointer to the resulting `tm` structure.

### | **Notes:**

- | 1. The range (0-61) for `tm_sec` allows for as many as two leap seconds.
- | 2. The `gmtime64()` and `localtime64()` functions can use a common, statically allocated buffer for the conversion. Each call to one of these functions might alter the result of the previous call. The `asctime_r()`, `ctime64_r()`, `gmtime64_r()`, and `localtime64_r()` functions do not use a common statically allocated buffer to hold the return string. These functions can be used in place of the `asctime()`, `ctime64()`, `gmtime64()`, and `localtime64()` functions if reentrancy is desired.
- | 3. Calendar time is the number of seconds that have elapsed since EPOCH, which is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).

### | **Example that uses `gmtime64()`**

| This example uses the `gmtime64()` function to adjust a `time64_t` representation to a Universal Coordinate Time character string and then converts it to a printable string using the `asctime()` function.

```
| #include <stdio.h>  
| #include <time.h>  
|  
| int main(void)  
| {  
|     time64_t ltime;  
|  
|     time64(&ltime);  
|     printf ("Universal Coordinate Time is %s",  
|           asctime(gmtime64(&ltime)));  
| }  
|  
| /***** Output should be similar to: *****/  
|  
| Universal Coordinate Time is Wed Aug 18 21:01:44 1993  
| */
```

### | **Related Information**

- | • “`asctime()` — Convert Time to Character String” on page 39
- | • “`asctime_r()` — Convert Time to Character String (Restartable)” on page 41
- | • “`ctime()` — Convert Time to Character String” on page 71
- | • “`ctime64()` — Convert Time to Character String” on page 73
- | • “`ctime64_r()` — Convert Time to Character String (Restartable)” on page 76
- | • “`ctime_r()` — Convert Time to Character String (Restartable)” on page 74
- | • “`gmtime()` — Convert Time” on page 161
- | • “`gmtime_r()` — Convert Time (Restartable)” on page 165
- | • “`gmtime64_r()` — Convert Time (Restartable)” on page 167
- | • “`localtime()` — Convert Time” on page 185
- | • “`localtime64()` — Convert Time” on page 187
- | • “`localtime64_r()` — Convert Time (Restartable)” on page 189

- | • “localtime\_r() — Convert Time (Restartable)” on page 188
- | • “mktime() — Convert Local Time” on page 218
- | • “mktime64() — Convert Local Time” on page 220
- | • “setlocale() — Set Locale” on page 339
- | • “time() — Determine Current Time” on page 411
- | • “time64() — Determine Current Time” on page 412
- | • “<time.h>” on page 18

---

## gmtime\_r() — Convert Time (Restartable)

### Format

```
#include <time.h>
struct tm *gmtime_r(const time_t *time, struct tm *result);
```

**Language Level:** XPG4

**Threadsafe:** Yes,

### Description

This function is the restartable version of `gmtime()`.

The `gmtime_r()` function breaks down the *time* value, in seconds, and stores it in *result*. *result* is a pointer to the *tm* structure, defined in `<time.h>`. The value *time* is usually obtained by a call to the `time()` function.

The fields of the *tm* structure include:

**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.

### Return Value

The `gmtime_r()` function returns a pointer to the resulting *tm* structure.

## Notes:

1. The range (0-61) for `tm_sec` allows for as many as two leap seconds.
2. The `gmtime()` and `localtime()` functions can use a common, statically allocated buffer for the conversion. Each call to one of these functions might alter the result of the previous call. The `asctime_r()`, `ctime_r()`, `gmtime_r()`, and `localtime_r()` functions do not use a common, statically allocated buffer to hold the return string. These functions can be used in place of the `asctime()`, `ctime()`, `gmtime()`, and `localtime()` functions if reentrancy is desired.
3. Calendar time is the number of seconds that have elapsed since EPOCH, which is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).

## Example that uses `gmtime_r()`

This example uses the `gmtime_r()` function to adjust a `time_t` representation to a Coordinated Universal Time character string, and then converts it to a printable string using the `asctime_r()` function.

```
#include <stdio.h>
#include <time.h>

int main(void)
{
    time_t ltime;
    struct tm mytime;
    char buf[50];

    time(&ltime)
    printf ("Coordinated Universal Time is %s\n",
           asctime_r(gmtime_r(&ltime, &mytime), buf));
}

/***** Output should be similar to: *****/

Coordinated Universal Time is Wed Aug 18 21:01:44 1993
*/
```

## Related Information

- “`asctime()` — Convert Time to Character String” on page 39
- “`asctime_r()` — Convert Time to Character String (Restartable)” on page 41
- “`ctime()` — Convert Time to Character String” on page 71
- | • “`ctime64()` — Convert Time to Character String” on page 73
- | • “`ctime64_r()` — Convert Time to Character String (Restartable)” on page 76
- “`ctime_r()` — Convert Time to Character String (Restartable)” on page 74
- “`gmtime()` — Convert Time” on page 161
- | • “`gmtime64()` — Convert Time” on page 163
- | • “`gmtime64_r()` — Convert Time (Restartable)” on page 167
- “`localtime()` — Convert Time” on page 185
- | • “`localtime64()` — Convert Time” on page 187
- | • “`localtime64_r()` — Convert Time (Restartable)” on page 189
- “`localtime_r()` — Convert Time (Restartable)” on page 188
- “`mktime()` — Convert Local Time” on page 218
- | • “`mktime64()` — Convert Local Time” on page 220
- “`time()` — Determine Current Time” on page 411
- | • “`time64()` — Determine Current Time” on page 412
- “`<time.h>`” on page 18

---

## gmtime64\_r() — Convert Time (Restartable)

### Format

```
#include <time.h>
struct tm *gmtime64_r(const time64_t *time, struct tm *result);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

### Description

This function is the restartable version of `gmtime64()`.

The `gmtime64_r()` function breaks down the *time* value, in seconds, and stores it in *result*. *result* is a pointer to the *tm* structure, defined in `<time.h>`. The value *time* is usually obtained by a call to the `time64()` function.

The fields of the *tm* structure include:

**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.

### Return Value

The `gmtime64_r()` function returns a pointer to the resulting *tm* structure.

### Notes:

1. The range (0-61) for `tm_sec` allows for as many as two leap seconds.
2. The `gmtime64()` and `localtime64()` functions might use a common, statically allocated buffer for the conversion. Each call to one of these functions might alter the result of the previous call. The `asctime_r()`, `ctime64_r()`, `gmtime64_r()`, and `localtime64_r()` functions do not use a common, statically allocated buffer to hold the return string. These functions can be used in place of the `asctime()`, `ctime64()`, `gmtime64()`, and `localtime64()` functions if reentrancy is desired.

3. Calendar time is the number of seconds that have elapsed since EPOCH, which is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).

#### Example that uses `gmtime64_r()`

This example uses the `gmtime64_r()` function to adjust a `time64_t` representation to a Universal Coordinate Time character string and then converts it to a printable string using the `asctime_r()` function.

```
#include <stdio.h>
#include <time.h>

int main(void)
{
    time64_t ltime;
    struct tm mytime;
    char buf[50];

    time64(&ltime)
    printf ("Universal Coordinate Time is %s",
           asctime_r(gmtime64_r(&ltime, &mytime), buf));
}

/***** Output should be similar to: *****/
Universal Coordinate Time is Wed Aug 18 21:01:44 1993
*/
```

#### Related Information

- “`asctime()` — Convert Time to Character String” on page 39
- “`asctime_r()` — Convert Time to Character String (Restartable)” on page 41
- “`ctime()` — Convert Time to Character String” on page 71
- “`ctime64()` — Convert Time to Character String” on page 73
- “`ctime64_r()` — Convert Time to Character String (Restartable)” on page 76
- “`ctime_r()` — Convert Time to Character String (Restartable)” on page 74
- “`gmtime()` — Convert Time” on page 161
- “`gmtime64()` — Convert Time” on page 163
- “`gmtime_r()` — Convert Time (Restartable)” on page 165
- “`localtime()` — Convert Time” on page 185
- “`localtime64()` — Convert Time” on page 187
- “`localtime64_r()` — Convert Time (Restartable)” on page 189
- “`localtime_r()` — Convert Time (Restartable)” on page 188
- “`mktime()` — Convert Local Time” on page 218
- “`mktime64()` — Convert Local Time” on page 220
- “`time()` — Determine Current Time” on page 411
- “`time64()` — Determine Current Time” on page 412
- “`<time.h>`” on page 18

---

## hypot() — Calculate Hypotenuse

### Format

```
#include <math.h>
double hypot(double side1, double side2);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

### Description

The `hypot()` function calculates the length of the hypotenuse of a right-angled triangle based on the lengths of two sides *side1* and *side2*. A call to the `hypot()` function is equivalent to:

```
sqrt(side1 * side1 + side2 * side2);
```

### Return Value

The `hypot()` function returns the length of the hypotenuse. If an overflow results, `hypot()` sets `errno` to `ERANGE` and returns the value `HUGE_VAL`. If an underflow results, `hypot()` sets `errno` to `ERANGE` and returns zero. The value of `errno` can also be set to `EDOM`.

### Example that uses `hypot()`

This example calculates the hypotenuse of a right-angled triangle with sides of 3.0 and 4.0.

```
#include <math.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 should be similar to: *****/

The hypotenuse of the triangle with sides 3.000000 and 4.000000 is 5.000000
*/
```

### Related Information

- “`sqrt()` — Calculate Square Root” on page 353
- “`<math.h>`” on page 8

---

## isalnum() - isxdigit() — Test Integer Value

### Format

```
#include <ctype.h>
int isalnum(int c);
/* Test for upper- or lowercase letters, or decimal digit */
int isalpha(int c);
/* Test for alphabetic character */
int iscntrl(int c);
/* Test for any control character */
int isdigit(int c);
/* Test for decimal digit */
int isgraph(int c);
/* Test for printable character excluding space */
```

```

int islower(int c);
/* Test for lowercase */
int isprint(int c);
/* Test for printable character including space */
int ispunct(int c);
/* Test for any nonalphanumeric printable character */
/* excluding space */
int isspace(int c);
/* Test for whitespace character */
int isupper(int c);
/* Test for uppercase */
int isxdigit(int c);
/* Test for hexadecimal digit */

```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of these functions might be affected by the LC\_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

| The <ctype.h> functions listed test a character with an integer value.

### Return Value

These functions return a nonzero value if the integer satisfies the test condition, or a zero value if it does not. The integer variable *c* must be representable as an unsigned char.

**Note:** EOF is a valid input value.

### Example that uses <ctype.h> functions

This example analyzes all characters between code 0x0 and code UPPER\_LIMIT, printing A for alphabetic characters, AN for alphanumerics, U for uppercase, L for lowercase, D for digits, X for hexadecimal digits, S for spaces, PU for punctuation, PR for printable characters, G for graphics characters, and C for control characters. This example prints the code if printable.

The output of this example is a 256-line table showing the characters from 0 to 255 that possess 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("%3s ", isalnum(ch) ? "AN" : " ");
        printf("%2s ", isalpha(ch) ? "A" : " ");
        printf("%2s", iscntrl(ch) ? "C" : " ");
        printf("%2s", isdigit(ch) ? "D" : " ");
        printf("%2s", isgraph(ch) ? "G" : " ");
        printf("%2s", islower(ch) ? "L" : " ");
        printf(" %c", isprint(ch) ? ch : ' ');
        printf("%3s", ispunct(ch) ? "PU" : " ");
        printf("%2s", isspace(ch) ? "S" : " ");
        printf("%3s", isprint(ch) ? "PR" : " ");
        printf("%2s", isupper(ch) ? "U" : " ");
        printf("%2s", isxdigit(ch) ? "X" : " ");

        putchar('\n');
    }
}

```

### Related Information

- “tolower() – toupper() — Convert Character Case” on page 416
- “isblank() — Test for Blank or Tab Character” on page 172
- “<ctype.h>” on page 3

---

## isascii() — Test for Character Representable as ASCII Value

### Format

```

#include <ctype.h>
int isascii(int c);

```

**Language Level:** XPG4

**Threadsafe:** Yes.

**Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `isascii()` function tests if a given character, in the current locale, can be represented as a valid 7-bit US-ASCII character.

### Return Value

The `isascii()` function returns nonzero if `c`, in the current locale, can be represented as a character in the 7-bit US-ASCII character set. Otherwise, it returns 0.

**Example that uses `isascii()`**

This example tests the integers from 0x7c to 0x82, and prints the corresponding character if the integer can be represented as a character in the 7-bit US-ASCII character set.

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    int ch;

    for (ch = 0x7c; ch <= 0x82; ch++) {
        printf("%#04x  ", ch);
        if (isascii(ch))
            printf("The character is %c\n", ch);
        else
            printf("Cannot be represented by an ASCII character\n");
    }
    return 0;
}

/*****
    The output should be:

0x7c  The character is @
0x7d  The character is '
0x7e  The character is =
0x7f  The character is "
0x80  Cannot be represented by an ASCII character
0x81  The character is a
0x82  The character is b

*****/
```

### Related Information

- “isalnum() - isxdigit() — Test Integer Value” on page 169
- “iswalnum() to iswxdigit() — Test Wide Integer Value” on page 173
- “toascii() — Convert Character to Character Representable by ASCII” on page 415
- “tolower() - toupper() — Convert Character Case” on page 416
- “towlower() - towupper() — Convert Wide Character Case” on page 418
- “<ctype.h>” on page 3

---

## isblank() — Test for Blank or Tab Character

### Format

```
#include <ctype.h>
int isblank(int c);
```

**Note:** The `isblank()` function is supported only for C++, not for C.

**Language Level:** Extended

**Threadsafe:** Yes.

### Description

The `isblank()` function tests if a character is either the EBCDIC space or EBCDIC tab character.

### Return Value

The `isblank()` function returns nonzero if `c` is either the EBCDIC space character or the EBCDIC tab character, otherwise it returns 0.

Example that uses `isblank()`

This example tests several characters using `isblank()`.

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    char *buf = "a b\tc";
    int i;

    for (i = 0; i < 5; i++) {
        if (isblank(buf[i]))
            printf("Character %d is not a blank.\n", i);
        else
            printf("Character %d is a blank\n", i);
    }
    return 0;
}
```

```
/******
   The output should be
```

```
Character 0 is not a blank.
Character 1 is a blank.
Character 2 is not a blank.
Character 3 is a blank.
Character 4 is not a blank.
```

```
*****/
```

### Related Information

- “`isalnum()` - `isxdigit()` — Test Integer Value” on page 169
- “`iswalnum()` to `iswxdigit()` — Test Wide Integer Value”
- “`isascii()` — Test for Character Representable as ASCII Value” on page 171
- “`tolower()` - `toupper()` — Convert Character Case” on page 416
- “`towlower()` - `towupper()` — Convert Wide Character Case” on page 418
- “`<ctype.h>`” on page 3

---

## `iswalnum()` to `iswxdigit()` — Test Wide Integer Value

### 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);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of these functions might be affected by the LC\_CTYPE category of the  
| current locale if LOCALETYPE(\*LOCALE) is specified on the compilation command. The behavior of  
| these functions might be affected by the LC\_UNI\_CTYPE category of the current locale if either the  
| LOCALETYPE(\*LOCALEUCS2) option or the LOCALETYPE(\*LOCALEUTF) option is specified on the  
| compilation command. These functions are not available when LOCALETYPE(\*CLD) is specified on the  
| compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

## Description

The functions listed above, which are all declared in <wctype.h>, test a given wide integer value.

The value of *wc* must be a wide-character code corresponding to a valid character in the current locale, or must equal the value of the macro WEOF. If the argument has any other value, the behavior is undefined.

Here are descriptions of each function in this group.

| **iswalnum()**  
|       Test for a wide alphanumeric character.

| **iswalpha()**  
|       Test for a wide alphabetic character, as defined in the alpha class of the current locale.

| **iswcntrl()**  
|       Test for a wide control character, as defined in the cntrl class of the current locale.

| **iswdigit()**  
|       Test for a wide decimal-digit character: 0 through 9, as defined in the digit class of the current  
|       locale.

| **iswgraph()**  
|       Test for a wide printing character, not a space, as defined in the graph class of the current locale.

| **iswlower()**  
|       Test for a wide lowercase character, as defined in the lower class of the current locale or for  
|       which none of the iswcntrl(), iswdigit(), iswspace() functions are true.

| **iswprint()**  
|       Test for any wide printing character, as defined in the print class of the current locale.

| **iswpunct()**  
|       Test for a wide nonalphanumeric, nonspace character, as defined in the punct class of the current  
|       locale.

| **iswspace()**  
|       Test for a wide whitespace character, as defined in the space class of the current locale.

| **iswupper()**  
|       Test for a wide uppercase character, as defined in the upper class 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 class of the current locale.

## 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 char. WEOF is a valid input value.

## Example

```
#include <stdio.h>
#include <wctype.h>

int main(void)
{
    int wc;

    for (wc=0; wc <= 0xFF; wc++) {
        printf("%3d", wc);
        printf(" %4x ", wc);
        printf("%3s", iswalnum(wc) ? "AN" : " ");
        printf("%2s", iswalph(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 19

---

## iswctype() — Test for Character Property

### Format

```
#include <wctype.h>
int iswctype(wint_t wc, wctype_t wc_prop);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

- | **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale if LOCALETYPE(\*LOCALE) is specified on the compilation command. The behavior of this function might be affected by the LC\_UNI\_CTYPE category of the current locale if either the LOCALETYPE(\*LOCALEUCS2) option or the LOCALETYPE(\*LOCALEUTF) option is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.
- | **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `iswctype()` function 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 characters that corresponds to a multibyte character, the behavior is undefined. If the value of `wc_prop` is incorrect (that is, it is not obtained by a previous call to the `wctype()` function, or `wc_prop` has been invalidated by a subsequent call to the `setlocale()` function), the behavior is undefined.

### Return Value

The `iswctype()` function returns true if the value of the wide character `wc` has the property `wc_prop`.

The following strings, `alnum` through to `xdigit` are reserved for the standard character classes. The functions are shown as follows with their equivalent `isw*()` function:

```
iswctype(wc, wctype("alnum"));      /* is equivalent to */      iswalnum(wc);
iswctype(wc, wctype("alpha"));      /* is equivalent to */      iswalpha(wc);
iswctype(wc, wctype("cntrl"));      /* is equivalent to */      iswcntrl(wc);
iswctype(wc, wctype("digit"));      /* is equivalent to */      iswdigit(wc);
iswctype(wc, wctype("graph"));      /* is equivalent to */      iswgraph(wc);
iswctype(wc, wctype("lower"));      /* is equivalent to */      iswlower(wc);
iswctype(wc, wctype("print"));      /* is equivalent to */      iswprint(wc);
iswctype(wc, wctype("punct"));      /* is equivalent to */      iswpunct(wc);
iswctype(wc, wctype("space"));      /* is equivalent to */      iswspace(wc);
iswctype(wc, wctype("upper"));      /* is equivalent to */      iswupper(wc);
iswctype(wc, wctype("xdigit"));     /* is equivalent to */      iswxdigit(wc);
```

### Example that uses `iswctype()`

```
#include <stdio.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()` — Get Handle for Character Property Classification” on page 495
- “`iswalnum()` to `iswxdigit()` — Test Wide Integer Value” on page 173
- “`<wctype.h>`” on page 19

---

## **`_itoa` - Convert Integer to String**

### Format

```
#include <stdlib.h>
char *_itoa(int value, char *string, int radix);
```

**Note:** The `_itoa` function is supported only for C++, not for C.

**Language Level:** Extension

**Threadsafe:** Yes.

## Description

`_itoa()` converts the digits of the given *value* to a character string that ends with a null character and stores the result in *string*. The *radix* argument specifies the base of *value*; it must be in the range 2 to 36. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (-).

**Note:** The space reserved for *string* must be large enough to hold the returned string. The function can return up to 33 bytes including the null character (`\0`).

## Return Value

`_itoa` returns a pointer to *string*. There is no error return value.

When the string argument is NULL or the *radix* is outside the range 2 to 36, `errno` will be set to `EINVAL`.

## Example that uses `_itoa()`

This example converts the integer value -255 to a decimal, a binary, and a hex number, storing its character representation in the array *buffer*.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char buffer[35];
    char *p;
    p = _itoa(-255, buffer, 10);
    printf("The result of _itoa(-255) with radix of 10 is %s\n", p);
    p = _itoa(-255, buffer, 2);
    printf("The result of _itoa(-255) with radix of 2\n    is %s\n", p);
    p = _itoa(-255, buffer, 16);
    printf("The result of _itoa(-255) with radix of 16 is %s\n", p);
    return 0;
}
```

The output should be:

```
The result of _itoa(-255) with radix of 10 is -255
The result of _itoa(-255) with radix of 2
    is 1111111111111111111111111111111100000001
The result of _itoa(-255) with radix of 16 is fffff01
```

## Related Information:

- “`_gcvt` - Convert Floating-Point to String” on page 151
- “`_itoa` - Convert Integer to String” on page 176
- “`_ltoa` - Convert Long Integer to String” on page 192
- “`_ultoa` - Convert Unsigned Long Integer to String” on page 419
- “`<stdlib.h>`” on page 17

---

## `labs()` — `llabs()` — Calculate Absolute Value of Long and Long Long Integer

### Format (`labs()`)

```
#include <stdlib.h>
long int labs(long int n);
```

### Format (`llabs()`)

```
#include <stdlib.h>
long long int llabs(long long int i);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `labs()` function produces the absolute value of its long integer argument  $n$ . The result might be undefined when the argument is equal to `LONG_MIN`, the smallest available long integer. The value `LONG_MIN` is defined in the `<limits.h>` include file.

The `llabs()` function returns the absolute value of its long long integer operand. The result might be undefined when the argument is equal to `LONG_LONG_MIN`, the smallest available long integer. The value `LONG_LONG_MIN` is defined in the `<limits.h>` include file.

### Return Value

The `labs()` function returns the absolute value of  $n$ . There is no error return value.

The `llabs()` function returns the absolute value of  $i$ . There is no error return value.

### Example that uses `labs()`

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 should be similar to: *****/

The absolute value of -41567 is 41567
*/
```

### Related Information

- “`abs()` — Calculate Integer Absolute Value” on page 37
- “`fabs()` — Calculate Floating-Point Absolute Value” on page 90
- “`<limits.h>`” on page 7

---

## **ldexp() — Multiply by a Power of Two**

### Format

```
#include <math.h>
double ldexp(double x, int exp);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `ldexp()` function calculates the value of  $x * (2^{exp})$ .

### Return Value

The `ldexp()` function returns the value of  $x * (2^{exp})$ . If an overflow results, the function returns `+HUGE_VAL` for a large result or `-HUGE_VAL` for a small result, and sets `errno` to `ERANGE`.

### Example that uses `ldexp()`

This example computes  $y$  as 1.5 times 2 to the fifth power ( $1.5 * 2^5$ ):

```
#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 should be similar to: *****/
1.500000 times 2 to the power of 5 is 48.000000
*/
```

### Related Information

- “`exp()` — Calculate Exponential Function” on page 89
- “`frexp()` — Separate Floating-Point Value” on page 132
- “`modf()` — Separate Floating-Point Value” on page 222
- “`<math.h>`” on page 8

---

## **ldiv() — lldiv() — Perform Long and Long Long Division**

### Format (`ldiv()`)

```
#include <stdlib.h>
ldiv_t ldiv(long int numerator, long int denominator);
```

### Format (`lldiv()`)

```
#include <stdlib.h>
lldiv_t lldiv(long long int numerator, long long int denominator);
```

**Language Level:** ANSI

**Threadsafe:** Yes. However, only the function version is threadsafe. The macro version is NOT threadsafe.

### Description

The `ldiv()` function calculates the quotient and remainder of the division of *numerator* by *denominator*.

### Return Value

The `ldiv()` function 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 return value is undefined. If *denominator* is 0, an exception is raised.

The `lldiv()` subroutine computes the quotient and remainder of the *numerator* parameter by the *denominator* parameter.

The `lldiv()` subroutine returns a structure of type `lldiv_t`, containing both the quotient and the remainder. The structure is defined as:

```
struct lldiv_t
{
    long long int quot; /* quotient */
    long long int rem; /* remainder */
};
```

If the division is inexact, the sign of the resulting quotient is that of the algebraic quotient, and magnitude of the resulting quotient is the largest long long integer less than the magnitude of the algebraic quotient. If the result cannot be represented (for example, if the *denominator* is 0), the behavior is undefined.

### Example that uses `ldiv()`

This example uses `ldiv()` 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 ints:
                '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);
        }
}
```

/\*\*\*\*\*\* Expected 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

- “`div()` — Calculate Quotient and Remainder” on page 86
- “`<stdlib.h>`” on page 17

---

## localeconv() — Retrieve Information from the Environment

### Format

```
#include <locale.h>
struct lconv *localeconv(void);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

- | **Locale Sensitive:** The behavior of this function might be affected by the LC\_NUMERIC and LC\_MONETARY categories of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

- | The `localeconv()` sets the components of a structure having type `struct lconv` to values appropriate for the current locale. The structure might be overwritten by another call to `localeconv()`, or by calling the `setlocale()` function.

The structure contains the following elements (defaults shown are for the C locale):

Element	Purpose of Element	Default
<code>char *decimal_point</code>	Decimal-point character used to format non-monetary quantities.	"."
<code>char *thousands_sep</code>	Character used to separate groups of digits to the left of the decimal-point character in formatted non-monetary quantities.	""
<code>char *grouping</code>	String indicating the size of each group of digits in formatted non-monetary quantities. Each character in the string specifies the number of digits in a group. The initial character represents the size of the group immediately to the left of the decimal delimiter. The characters following this define succeeding groups to the left of the previous group. If the last character is not <code>UCHAR_MAX</code> , the grouping is repeated using the last character as the size. If the last character is <code>UCHAR_MAX</code> , grouping is only performed for the groups already in the string (no repetition). See Table 1 on page 183 for an example of how grouping works.	""
<code>char *int_curr_symbol</code>	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.	""
<code>char *currency_symbol</code>	Local currency symbol of the current locale.	""
<code>char *mon_decimal_point</code>	Decimal-point character used to format monetary quantities.	""
<code>char *mon_thousands_sep</code>	Separator for digits in formatted monetary quantities.	""

Element	Purpose of Element	Default
char *mon_grouping	String indicating the size of each group of digits in formatted monetary quantities. Each character in the string specifies the number of digits in a group. The initial character represents the size of the group immediately to the left of the decimal delimiter. The following characters define succeeding groups to the left of the previous group. If the last character is not UCHAR_MAX, the grouping is repeated using the last character as the size. If the last character is UCHAR_MAX, grouping is only performed for the groups already in the string (no repetition). See Table 1 on page 183 for an example of how grouping works.	""
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.	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.	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

Pointers to strings with a value of "" indicate that the value is not available in the C locale or is of zero length. Elements with char types with a value of UCHAR\_MAX indicate that the value is not available in the current locale.

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`.

### Grouping Example

Table 1. Grouping Example

Locale Source	Grouping String	Number	Formatted Number
-1	0x00	123456789	123456789
3	0x0300	123456789	123,456,789
3;-1	0x03FF00	123456789	123456,789
3;2;1	0x03020100	123456789	1,2,3,4,56,789

### Monetary Formatting Example:

Table 2. Monetary Formatting Example

Country	Positive Format	Negative Format	International Format
Italy	L.1.230	-L.1.230	ITL.1.230
Netherlands	F 1.234,56	F -1.234,56	NLG 1.234,56
Norway	kr1.234,56	kr1.234,56-	NOK1.234,56
Switzerland	SFRs.1,234.56	SFRx.1,234.56C	CHF 1,234.56

| The above table was generated by locales with the following monetary fields:

Table 3. Monetary Fields

	Italy	Netherlands	Norway	Switzerland
<code>int_curr_symbol</code>	"ITL."	"NLG"	"NOK"	"CHF"
<code>currency_symbol</code>	"L."	"F"	"kr"	"SFRs."
<code>mon_decimal_point</code>	""	","	"/	."
<code>mon_thousands_sep</code>	""	."	."	"/
<code>mon_grouping</code>	"\3"	"\3"	"\3"	"\3"
<code>positive_sign</code>	""	""	""	""
<code>negative_sign</code>	"_"	"_"	"_"	"C"
<code>int_frac_digits</code>	0	2	2	2
<code>frac_digits</code>	0	2	2	2
<code>p_cs_precedes</code>	1	1	1	1
<code>p_sep_by_space</code>	0	1	0	0
<code>n_cs_precedes</code>	1	1	1	1
<code>n_sep_by_space</code>	0	1	0	0
<code>p_sep_posn</code>	1	1	1	1
<code>n_sep_posn</code>	1	4	2	2

### Return Value

The `localeconv()` function returns a pointer to the structure.

### Example that uses \*CLD locale objects

This example prints out the default decimal point for your locale and then the decimal point for the LC\_C\_FRANCE 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, LC_C_FRANCE))) {
        mylocale = localeconv();

        /* A comma is set to be the decimal point when the locale is LC_C_FRANCE*/

        printf("France's decimal point is a %s\n", mylocale->decimal_point);

    } else {

        printf("setlocale(LC_ALL, LC_C_FRANCE) returned <NULL>\n");
    }
    return 0;
}
```

**Example that uses \*LOCALE objects**

```

/*****
This example prints out the default decimal point for
the C locale and then the decimal point for the French
locale using a *LOCALE object called
"QSYS.LIB/MYLIB.LIB/LC_FRANCE.LOCALE".

Step 1: Create a French *LOCALE object by entering the command
      CRTLOCALE LOCALE('QSYS.LIB/MYLIB.LIB/LC_FRANCE.LOCALE') +
              SRCFILE('QSYS.LIB/QSYSLOCALE.LIB/QLOCALESRC.FILE/ +
              FR_FR.MBR') CCSID(297)
Step 2: Compile the following C source, specifying
      LOCALETYPE(*LOCALE) on the compilation command.
Step 3: Run the program.
*****/

```

```

#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,
        "QSYS.LIB/MYLIB.LIB/LC_FRANCE.LOCALE"))) {
        mylocale = localeconv();

        /* A comma is set to be the decimal point in the French locale */
        printf("France's decimal point is a %s\n", mylocale->decimal_point);
    } else {
        printf("setlocale(LC_ALL, \"QSYS.LIB/MYLIB.LIB/LC_FRANCE.LOCALE\") \
            returned <NULL>\n");
    }
    return 0;
}

```

### Related Information

- “setlocale() — Set Locale” on page 339
- “<locale.h>” on page 7

---

## localtime() — Convert Time

### Format

```

#include <time.h>
struct tm *localtime(const time_t *timeval);

```

**Language Level:** ANSI

**Threadsafe:** No. Use `localtime_r()` instead.

**Locale Sensitive:** The behavior of this function might be affected by the `LC_TOD` category of the current locale.

### Description

The `localtime()` function converts a time value, in seconds, to a structure of type `tm`.

The `localtime()` function takes a `timeval` assumed to be Universal Coordinate Time (UTC) and converts it to job locale time. For this conversion `localtime()` checks the current locale setting for local time zone and daylight saving time (DST). If these values are not set in the current locale, `localtime()` gets the local

time zone and daylight saving time (DST) settings from the current job. Once converted, the time is returned in a structure of type *tm*. If the DST is set in the locale but the time zone information is not, the DST information in the locale is ignored.

The time value is usually obtained by a call to the `time()` function.

#### Notes:

1. The `gmtime()` and `localtime()` functions can use a common, statically allocated buffer for the conversion. Each call to one of these functions might destroy the result of the previous call. The `ctime_r()`, `gmtime_r()`, and `localtime_r()` functions do not use a common, statically allocated buffer. These functions can be used in place of the `asctime()`, `ctime()`, `gmtime()` and `localtime()` functions if reentrancy is desired.
2. Calendar time is the number of seconds that have elapsed since EPOCH, which is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).

#### Return Value

The `localtime()` function returns a pointer to the structure result. There is no error return value.

#### Example that uses `localtime()`

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;
|
|     ltime = time(&ltime);
|     newtime = localtime(&ltime);
|     printf("The date and time is %s", asctime(newtime));}
|
| /***** If the local time is 3 p.m. February 15, 2008, *****/
| *****/
|
| The date and time is Fri Feb 15 15:00:00 2008
| */
```

#### Related Information

- “`asctime()` — Convert Time to Character String” on page 39
- “`asctime_r()` — Convert Time to Character String (Restartable)” on page 41
- “`ctime()` — Convert Time to Character String” on page 71
- “`ctime64()` — Convert Time to Character String” on page 73
- “`ctime64_r()` — Convert Time to Character String (Restartable)” on page 76
- “`ctime_r()` — Convert Time to Character String (Restartable)” on page 74
- “`gmtime()` — Convert Time” on page 161
- “`gmtime64()` — Convert Time” on page 163
- “`gmtime64_r()` — Convert Time (Restartable)” on page 167
- “`gmtime_r()` — Convert Time (Restartable)” on page 165
- “`localtime_r()` — Convert Time (Restartable)” on page 188
- “`localtime64()` — Convert Time” on page 187
- “`localtime64_r()` — Convert Time (Restartable)” on page 189

- “mktime() — Convert Local Time” on page 218
- | • “mktime64() — Convert Local Time” on page 220
- “setlocale() — Set Locale” on page 339
- “time() — Determine Current Time” on page 411
- “time64() — Determine Current Time” on page 412
- “<time.h>” on page 18

---

## | localtime64() — Convert Time

### | Format

```
| #include <time.h>
| struct tm *localtime64(const time64_t *timeval);
```

| **Language Level:** ILE C Extension

| **Threadsafe:** No. Use localtime64\_r() instead.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_TOD category of the current locale.

### | Description

| The localtime64() function converts a time value, in seconds, to a structure of type *tm*.

| The localtime64() function takes a *timeval* assumed to be Universal Coordinate Time (UTC) and converts it to job locale time. For this conversion, localtime64() checks the current locale setting for local time zone and daylight saving time (DST). If these values are not set in the current locale, localtime64() gets the local time zone and daylight saving time (DST) settings from the current job. Once converted, the time is returned in a structure of type *tm*. If the DST is set in the locale but the time zone information is not, the DST information in the locale is ignored.

| The time value is usually obtained by a call to the time64() function.

### | Notes:

- | 1. The gmtime64() and localtime64() functions might use a common, statically allocated buffer for the conversion. Each call to one of these functions might alter the result of the previous call. The asctime\_r(), ctime64\_r(), gmtime64\_r() and localtime64\_r() functions do not use a common, statically allocated buffer. These functions can be used in place of the asctime(), ctime64(), gmtime64(), and localtime64() functions if thread safety is desired.
- | 2. Calendar time is the number of seconds that have elapsed since EPOCH, which is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).
- | 3. The supported date and time range for this function is 01/01/0001 00:00: 00 through 12/31/9999 23: 59: 59.

### | Return Value

| The localtime64() function returns a pointer to the structure result. If the given *timeval* is out of range, a NULL pointer is returned and errno is set to EOVERFLOW.

### | Example that uses localtime64()

| This example queries the system clock and displays the local time.

|

```

| #include <stdio.h>
| #include <time.h>
|
| int main(void)
| {
|     struct tm *newtime;
|     time64_t ltime;
|
|     ltime = time64(&ltime);
|     newtime = localtime64(&ltime);
|     printf("The date and time is %s", asctime(newtime));
| }
|
| /***** If the local time is 3 p.m. February 15, 2008, *****/
| *****/ the output should be: *****/
|
| The date and time is Fri Feb 15 15:00:00 2008
| */

```

### Related Information

- | • “asctime() — Convert Time to Character String” on page 39
- | • “asctime\_r() — Convert Time to Character String (Restartable)” on page 41
- | • “ctime() — Convert Time to Character String” on page 71
- | • “ctime64() — Convert Time to Character String” on page 73
- | • “ctime64\_r() — Convert Time to Character String (Restartable)” on page 76
- | • “ctime\_r() — Convert Time to Character String (Restartable)” on page 74
- | • “gmtime() — Convert Time” on page 161
- | • “gmtime64() — Convert Time” on page 163
- | • “gmtime64\_r() — Convert Time (Restartable)” on page 167
- | • “gmtime\_r() — Convert Time (Restartable)” on page 165
- | • “localtime() — Convert Time” on page 185
- | • “localtime64\_r() — Convert Time (Restartable)” on page 189
- | • “localtime\_r() — Convert Time (Restartable)”
- | • “mktime() — Convert Local Time” on page 218
- | • “mktime64() — Convert Local Time” on page 220
- | • “setlocale() — Set Locale” on page 339
- | • “time() — Determine Current Time” on page 411
- | • “time64() — Determine Current Time” on page 412
- | • “<time.h>” on page 18

---

## localtime\_r() — Convert Time (Restartable)

### Format

```

#include <time.h>
struct tm *localtime_r(const time_t *timeval, struct tm *result);

```

**Language Level:** XPG4

**Threadsafe:** Yes

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_TOD category of the current locale.

### Description

This function is the restartable version of `localtime()`. It is the same as `localtime()` except that it passes in the place to store the returned structure *result*.

### Return Value

The `localtime_r()` returns a pointer to the structure `result`. There is no error return value.

### Example that uses `localtime_r()`

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;
|     char buf[50];
|
|     ltime=time(&ltime);
|     localtime_r(&ltime, &newtime);
|     printf("The date and time is %s", asctime_r(&newtime, buf));
| }
|
| /****** If the local time is 3 p.m. February 15, 2008, *****/
| *****/ the output should be: *****/
|
| The date and time is Fri Feb 15 15:00:00 2008
| */
```

### Related Information

- “`asctime()` — Convert Time to Character String” on page 39
- “`asctime_r()` — Convert Time to Character String (Restartable)” on page 41
- “`ctime()` — Convert Time to Character String” on page 71
- “`ctime_r()` — Convert Time to Character String (Restartable)” on page 74
- “`gmtime()` — Convert Time” on page 161
- “`gmtime_r()` — Convert Time (Restartable)” on page 165
- “`localtime()` — Convert Time” on page 185
- “`mktime()` — Convert Local Time” on page 218
- “`time()` — Determine Current Time” on page 411
- “`<time.h>`” on page 18

---

## `localtime64_r()` — Convert Time (Restartable)

### Format

```
| #include <time.h>
| struct tm *localtime64_r(const time64_t *timeval, struct tm *result);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

**Locale Sensitive:** The behavior of this function might be affected by the `LC_TOD` category of the current locale.

### Description

| This function is the restartable version of `localtime64()`. It is the same as `localtime64()` except that it passes in the place to store the returned structure *result*.

| **Notes:**

- | 1. The `gmtime64()` and `localtime64()` functions might use a common, statically allocated buffer for the conversion. Each call to one of these functions might alter the result of the previous call. The `asctime_r()`, `ctime64_r()`, `gmtime64_r()`, and `localtime64_r()` functions do not use a common statically allocated buffer to hold the return string. These functions can be used in place of the `asctime()`, `ctime64()`, `gmtime64()`, and `localtime64()` functions if thread safety is desired.
- | 2. Calendar time is the number of seconds that have elapsed since EPOCH, which is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).
- | 3. The supported date and time range for this function is 01/01/0001 00:00:00 through 12/31/9999 23:59:59.

| **Return Value**

| The `localtime64_r()` function returns a pointer to the structure *result*. If the given *timeval* is out of range, a NULL pointer is returned and `errno` is set to `E_OVERFLOW`.

| **Example that uses `localtime64_r()`**

| This example queries the system clock and displays the local time.

```
| #include <stdio.h>
| #include <time.h>
|
| int main(void)
| {
|     struct tm newtime;
|     time64_t ltime;
|     char buf[50];
|
|     ltime = time64(&ltime);
|     localtime64_r(&ltime, &newtime);
|     printf("The date and time is %s\n", asctime_r(&newtime, buf));
| }
|
| /***** If the local time is 3 p.m. February 15, 2008, *****/
| *****/ the output should be: *****/
|
| The date and time is Fri Feb 15 15:00:00 2008
| */
```

| **Related Information**

- | • “`asctime()` — Convert Time to Character String” on page 39
- | • “`asctime_r()` — Convert Time to Character String (Restartable)” on page 41
- | • “`ctime64()` — Convert Time to Character String” on page 73
- | • “`ctime64_r()` — Convert Time to Character String (Restartable)” on page 76
- | • “`gmtime64()` — Convert Time” on page 163
- | • “`gmtime64_r()` — Convert Time (Restartable)” on page 167
- | • “`localtime64()` — Convert Time” on page 187
- | • “`mktime64()` — Convert Local Time” on page 220
- | • “`time64()` — Determine Current Time” on page 412
- | • “`<time.h>`” on page 18

---

## log() — Calculate Natural Logarithm

### Format

```
#include <math.h>
double log(double x);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `log()` function calculates the natural logarithm (base  $e$ ) of  $x$ .

### Return Value

The `log()` function returns the computed value. If  $x$  is negative, `log()` sets `errno` to `EDOM` and might return the value `-HUGE_VAL`. If  $x$  is zero, `log()` returns the value `-HUGE_VAL`, and might set `errno` to `ERANGE`.

### Example that uses `log()`

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 should be similar to: *****/

The natural logarithm of 1000.000000 is 6.907755
*/
```

### Related Information

- “`exp()` — Calculate Exponential Function” on page 89
- “`log10()` — Calculate Base 10 Logarithm”
- “`pow()` — Compute Power” on page 228
- “`<math.h>`” on page 8

---

## log10() — Calculate Base 10 Logarithm

### Format

```
#include <math.h>
double log10(double x);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

## Description

The `log10()` function calculates the base 10 logarithm of  $x$ .

## Return Value

The `log10()` function returns the computed value. If  $x$  is negative, `log10()` sets `errno` to `EDOM` and might return the value `-HUGE_VAL`. If  $x$  is zero, the `log10()` function returns the value `-HUGE_VAL`, and might set `errno` to `ERANGE`.

## Example that uses `log10()`

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 should be similar to: *****/

The base 10 logarithm of 1000.000000 is 3.000000
*/
```

## Related Information

- “`exp()` — Calculate Exponential Function” on page 89
- “`log()` — Calculate Natural Logarithm” on page 191
- “`pow()` — Compute Power” on page 228
- “`<math.h>`” on page 8

---

## **`_ltoa` - Convert Long Integer to String**

### Format

```
#include <stdlib.h>
char *_ltoa(long value, char *string, int radix);
```

**Note:** The `_ltoa` function is supported only for C++, not for C.

**Language Level:** Extension

**Threadsafe:** Yes.

## Description

`_ltoa` converts the digits of the given long integer *value* to a character string that ends with a null character and stores the result in *string*. The *radix* argument specifies the base of *value*; it must be in the range 2 to 36. If *radix* equals 10 and *value* is negative, the first character of the stored string is the minus sign (-).

**Note:** The space allocated for *string* must be large enough to hold the returned string. The function can return up to 33 bytes including the null character (`\0`).

## Return Value

`_ltoa` returns a pointer to *string*. There is no error return value.

When the string argument is NULL or the *radix* is outside the range 2 to 36, `errno` will be set to EINVAL.

## Example that uses `_ltoa()`

This example converts the integer value `-255L` to a decimal, a binary, and a hex value, and stores its character representation in the array *buffer*.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char buffer[35];
    char *p;
    p = _ltoa(-255L, buffer, 10);
    printf("The result of _ltoa(-255) with radix of 10 is %s\n", p);
    p = _ltoa(-255L, buffer, 2);
    printf("The result of _ltoa(-255) with radix of 2\n    is %s\n", p);
    p = _ltoa(-255L, buffer, 16);
    printf("The result of _ltoa(-255) with radix of 16 is %s\n", p);
    return 0;
}
```

The output should be:

```
The result of _ltoa(-255) with radix of 10 is -255
The result of _ltoa(-255) with radix of 2
    is 1111111111111111111111111111111100000001
The result of _ltoa(-255) with radix of 16 is fffff01
```

## Related Information:

- “`atol()` — `atoll()` — Convert Character String to Long or Long Long Integer” on page 49
- “`_gcvt` - Convert Floating-Point to String” on page 151
- “`_ltoa` - Convert Integer to String” on page 176
- “`strtol()` — `strtoll()` — Convert Character String to Long and Long Long Integer” on page 400
- “`_ultoa` - Convert Unsigned Long Integer to String” on page 419
- “`wcstol()` — `wcstoll()` — Convert Wide Character String to Long and Long Long Integer” on page 481
- “`<stdlib.h>`” on page 17

---

## `longjmp()` — Restore Stack Environment

### Format

```
#include <setjmp.h>
void longjmp(jmp_buf env, int value);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `longjmp()` function restores a stack environment previously saved in *env* by the `setjmp()` function. The `setjmp()` and `longjmp()` functions provide a way to perform a non-local goto. They are often used in signal handlers.

A call to the `setjmp()` function causes the current stack environment to be saved in *env*. A subsequent call to `longjmp()` restores the saved environment and returns control to a point in the program corresponding to the `setjmp()` call. Processing resumes as if the `setjmp()` call had just returned the given *value*.

All variables (except register variables) that are available to the function that receives control contain the values they had when `longjmp()` was called. The values of register variables are unpredictable. Nonvolatile auto variables that are changed between calls to the `setjmp()` and `longjmp()` functions are also unpredictable.

**Note:** Ensure that the function that calls the `setjmp()` function does not return before you call the corresponding `longjmp()` function. Calling `longjmp()` after the function calling the `setjmp()` function returns causes unpredictable program behavior.

The *value* argument must be nonzero. If you give a zero argument for *value*, `longjmp()` substitutes 1 in its place.

### Return Value

The `longjmp()` function does not use the normal function call and return mechanisms; it has no return value.

### Example that uses `longjmp()`

This example saves 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 the `setjmp()` function returns a 0 when it saves the environment. The program prints the message:

```
setjmp has been called
```

```
| The subsequent call to function p() causes it to call the longjmp() function. Control is transferred to the
| point in the main() function immediately after the call to the setjmp() function using the environment
| saved in the mark variable. This time, the condition is TRUE because -1 is specified in the second
| parameter on the longjmp() function call as the return value to be placed on the stack. The example then
| performs the statements in the block, prints the message "longjmp() has been called", calls the recover()
| function, and leaves the program.
```

```

| #include <stdio.h>
| #include <setjmp.h>
| #include <stdlib.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");
|     printf("Calling function p()\n");
|     p();
|     printf("This point should never be reached\n");
| }
|
| void p(void)
| {
|     printf("Calling longjmp() from inside function p()\n");
|     longjmp(mark, -1);
|     printf("This point should never be reached\n");
| }
|
| void recover(void)
| {
|     printf("Performing function recover()\n");
| }
|
| /*****Output should be as follows: *****/
| setjmp has been called
| Calling function p()
| Calling longjmp() from inside function p()
| longjmp has been called
| Performing function recover()
| *****/

```

### Related Information

- “setjmp() — Preserve Environment” on page 338
- “<setjmp.h>” on page 13

---

## malloc() — Reserve Storage Block

### Format

```

#include <stdlib.h>
void *malloc(size_t size);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `malloc()` function reserves a block of storage of *size* bytes. Unlike the `calloc()` function, `malloc()` does not initialize all elements to 0. The maximum size for a non-Teraspace `malloc()` is 16711568 bytes.

**Notes:**

1. All heap storage is associated with the activation group of the calling routine. As such, storage should be allocated and deallocated within the same activation group. You cannot allocate heap storage within one activation group and deallocate that storage from a different activation group. For more information about activation groups, see the *ILE Concepts* manual.
2. To use Teraspace storage instead of heap storage without changing the C source code, specify the TERASPACE(\*YES \*TSIFC) parameter on the CRTCMOD compiler command. This maps the malloc() library function to \_C\_TS\_malloc(), its Teraspace storage counterpart. The maximum amount of Teraspace storage that can be allocated by each call to \_C\_TS\_malloc() is 2GB - 224, or 2147483424 bytes. If more than 2147483408 bytes are needed on a single request, call \_C\_TS\_malloc64(unsigned long long int);.  
For more information, see the *ILE Concepts* manual.
3. For current statistics on the teraspace storage being used by MI programs in an activation group, call the \_C\_TS\_malloc\_info function. This function returns information including total bytes, allocated bytes and blocks, unallocated bytes and blocks, requested bytes, pad bytes, and overhead bytes. To get more detailed information about the memory structures used by the \_C\_TS\_malloc() and \_C\_TS\_malloc64() functions, call the \_C\_TS\_malloc\_debug function. You can use the information this function returns to identify memory corruption problems.
4. If the \_C\_Quickpool\_Init() function has been called in the current activation group, then the storage is retrieved using Quick Pool memory management. See \_C\_Quickpool\_Init() for more information.

**Return Value**

The malloc() function returns a pointer to the reserved space. The storage space to which the return value points is suitably aligned for storage of any type of object. The return value is NULL if not enough storage is available, or if *size* was specified as zero.

**Example that uses malloc()**

This example prompts 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 */
        perror( "Out of storage" );
        abort();
    }
}

/***** Output should be similar to: *****/

Enter the size of the array
array[ 0 ] = 0
array[ 1 ] = 1
array[ 2 ] = 2
array[ 3 ] = 3
array[ 4 ] = 4
*/

```

## | Related Information

- | • “calloc() — Reserve and Initialize Storage” on page 55
- | • “\_C\_Quickpool\_Debug() — Modify Quick Pool Memory Management Characteristics” on page 66
- | • “\_C\_Quickpool\_Init() — Initialize Quick Pool Memory Management” on page 68
- | • “\_C\_Quickpool\_Report() — Generate Quick Pool Memory Management Report” on page 70
- | • “free() — Release Storage Blocks” on page 128
- | • “realloc() — Change Reserved Storage Block Size” on page 264
- | • “<stdlib.h>” on page 17

---

## mblen() — Determine Length of a Multibyte Character

### Format

```

#include <stdlib.h>
int mblen(const char *string, size_t n);

```

**Language Level:** ANSI

**Threadsafe:** No. Use mbrlen() instead.

- | **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. This function might be affected by the LC\_UNI\_CTYPE category of the current locale if

| LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation  
| command. For more information, see “Understanding CCSIDs and Locales” on page 524.

## Description

The `mblen()` function determines the length in bytes of the multibyte character pointed to by *string*. *n* represents the maximum number of bytes examined.

## Return Value

If *string* is `NULL`, the `mblen()` function returns:

- Non-zero if the active locale allows mixed-byte strings. The function initializes the state variable.
- 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.

**Note:** The `mblen()`, `mbtowc()`, and `wctomb()` functions use their own statically allocated storage and are therefore not restartable. However, `mbrlen()`, `mbrtowc()`, and `wcrtomb()` are restartable.

## Example that uses `mblen()`

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)
{
    /* Initialize internal state variable */
    length = mblen(NULL, MB_CUR_MAX);

    /* Set string to point to a multibyte character */
    length = mblen(string, MB_CUR_MAX);
    temp = mbtowc(arr, string, length);
    arr[1] = L'\0';
    printf("wide character string: %ls\n", arr);
}
```

## Related Information

- “`mbrlen()` — Determine Length of a Multibyte Character (Restartable)” on page 199
- “`mbtowc()` — Convert Multibyte Character to a Wide Character” on page 211
- “`mbstowcs()` — Convert a Multibyte String to a Wide Character String” on page 207
- “`strlen()` — Determine String Length” on page 375
- “`wcslen()` — Calculate Length of Wide-Character String” on page 461
- “`wctomb()` — Convert Wide Character to Multibyte Character” on page 492
- “`<stdlib.h>`” on page 17

---

## mbrlen() — Determine Length of a Multibyte Character (Restartable)

### Format

```
#include <wchar.h>
size_t mbrlen (const char *s, size_t n, mbstate_t *ps);
```

**Language Level:** ANSI

**Threadsafe:** Yes, if *ps* is not NULL.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the  
| current locale. This function might also be affected by the LC\_UNI\_CTYPE category of the current locale  
| if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation  
| command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation  
| command. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

This function is the restartable version of `mblen()`.

The `mbrlen()` function determines the length of a multibyte character.

*n* is the number of bytes (at most) of the multibyte string to examine.

This function differs from its corresponding internal-state multibyte character function in that it has an extra parameter, *ps* of type pointer to `mbstate_t` that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If *ps* is a NULL pointer, `mbrlen()` behaves like `mblen()`.

`mbrlen()` is a restartable version of `mblen()`. In other words, shift-state information is passed as one of the arguments (*ps* represents the initial shift) 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.

### Return Value

If *s* is a null pointer and if the active locale allows mixed-byte strings, the `mbrlen()` function returns nonzero. If *s* is a null pointer and if the active locale does not allow mixed-byte strings, zero will be returned.

If *s* is not a null pointer, the `mbrlen()` function returns one of the following:

0        If *s* is a NULL string (*s* points to the NULL character).

#### positive

If the next *n* or fewer bytes comprise a valid multibyte character. The value returned is the number of bytes that comprise the multibyte character.

#### (size\_t)-1

If *s* does not point to a valid multibyte character.

#### (size\_t)-2

If the next *n* or fewer bytes contribute to an incomplete but potentially valid character and all *n* bytes have been processed

### Example that uses `mbrlen()`

```
/* This program is compiled with LOCALETYPE(*LOCALE) and */
/* SYSIFCOPT(*IFS10) */
```

```

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>
#include <errno.h>

#define LOCNAME      "qsys.lib/JA_JP.locale"
#define LOCNAME_EN  "qsys.lib/EN_US.locale"

int main(void)
{
    int length, sl = 0;
    char string[10];
    mbstate_t ps = 0;
    memset(string, '\0', 10);
    string[0] = 0xC1;
    string[1] = 0x0E;
    string[2] = 0x41;
    string[3] = 0x71;
    string[4] = 0x41;
    string[5] = 0x72;
    string[6] = 0x0F;
    string[7] = 0xC2;
    /* In this first example we will find the length of      */
    /* of a multibyte character when the CCSID of locale    */
    /* associated with LC_CTYPE is 37.                       */
    /* For single byte cases the state will always        */
    /* remain in the initial state 0                       */
    /*
    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");

    length = mbrlen(string, MB_CUR_MAX, &ps);

    /* In this case length is 1, which is always the case for */
    /* single byte CCSID */

    printf("length = %d, state = %d\n\n", length, ps);
    printf("MB_CUR_MAX: %d\n\n", MB_CUR_MAX);

    /* Now let's try a multibyte example. We first must set the */
    /* locale to a multibyte locale. We choose a locale with    */
    /* CCSID 5026 */

    if (setlocale(LC_ALL, LOCNAME) == NULL)
        printf("setlocale failed.\n");

    length = mbrlen(string, MB_CUR_MAX, &ps);

    /* The first is single byte so length is 1 and             */
    /* the state is still the initial state 0                 */
    /*
    printf("length = %d, state = %d\n\n", length, ps);
    printf("MB_CUR_MAX: %d\n\n", MB_CUR_MAX);

    sl += length;

    length = mbrlen(&string[sl], MB_CUR_MAX, &ps);

    /* The next character is a mixed byte. Length is 3 to     */
    /* account for the shiftout 0x0e. State is                */
    /* changed to double byte state.                          */
    /*
    printf("length = %d, state = %d\n\n", length, ps);

    sl += length;

```

```

length = mbrlen(&string[s1], MB_CUR_MAX, &ps);

/* The next character is also a double byte character. */
/* The state is changed to initial state since this was */
/* the last double byte character. Length is 3 to */
/* account for the ending 0x0f shiftin. */

printf("length = %d, state = %d\n\n", length, ps);

s1 += length;

length = mbrlen(&string[s1], MB_CUR_MAX, &ps);

/* The next character is single byte so length is 1 and */
/* state remains in initial state. */

printf("length = %d, state = %d\n\n", length, ps);

}
/* The output should look like this:

length = 1, state = 0
MB_CUR_MAX: 1
length = 1, state = 0
MB_CUR_MAX: 4
length = 3, state = 2
length = 3, state = 0
length = 1, state = 0
*/
* * * End of File * * *

```

### Related Information

- “mbrlen() — Determine Length of a Multibyte Character” on page 197
- “mbtowl() — Convert Multibyte Character to a Wide Character” on page 211
- “mbrtowc() — Convert a Multibyte Character to a Wide Character (Restartable)”
- “mbsrtowcs() — Convert a Multibyte String to a Wide Character String (Restartable)” on page 206
- “setlocale() — Set Locale” on page 339
- “wctomb() — Convert a Wide Character to a Multibyte Character (Restartable)” on page 446
- “wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)” on page 473
- “<locale.h>” on page 7
- “<wchar.h>” on page 18

---

## mbrtowc() — Convert a Multibyte Character to a Wide Character (Restartable)

### Format

```

#include <wchar.h>
size_t mbrtowc (wchar_t *pwc, const char *s, size_t n, mbstate_t *ps);

```

**Language Level:** ANSI

**Threadsafe:** Yes, if *ps* is not NULL

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. This function might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

## Description

This function is the restartable version of the `mbtowc()` function.

If *s* is a null pointer, the `mbrtowc()` function determines the number of bytes necessary to enter the initial shift state (zero if encodings are not state-dependent or if the initial conversion state is described). In this situation, the value of the *pwc* parameter will be ignored and the resulting shift state described will be the initial conversion state.

If *s* is not a null pointer, the `mbrtowc()` function determines the number of bytes that are in the multibyte character (and any leading shift sequences) pointed to by *s*, produces the value of the corresponding multibyte character and if *pwc* is not a null pointer, stores that value in the object pointed to by *pwc*. If the corresponding multibyte character is the null wide character, the resulting state will be reset to the initial conversion state.

This function differs from its corresponding internal-state multibyte character function in that it has an extra parameter, *ps* of type pointer to `mbstate_t` that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If *ps* is NULL, this function uses an internal static variable for the state.

At most, *n* bytes of the multibyte string are examined.

## Return Value

If *s* is a null pointer, the `mbrtowc()` function returns the number of bytes necessary to enter the initial shift state. The value returned must be less than the `MB_CUR_MAX` macro.

If a conversion error occurs, `errno` might be set to `ECONVERT`.

If *s* is not a null pointer, the `mbrtowc()` function returns one of the following:

**0** If the next *n* or fewer bytes form the multibyte character that corresponds to the null wide character.

### positive

If the next *n* or fewer bytes form a valid multibyte character. The value returned is the number of bytes that constitute the multibyte character.

### (size\_t)-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 the value of the `MB_CUR_MAX` macro.

### (size\_t)-1

If an encoding error occurs (when the next *n* or fewer bytes do not form a complete and correct multibyte character). The value of the macro `EILSEQ` is stored in `errno`, but the conversion state is unchanged.

**Note:** When a -2 value is returned, the string could contain redundant shift-out and shift-in characters or a partial UTF-8 character. To continue processing the multibyte string, increment the pointer by the value *n*, and call `mbrtowc()` again.

### Example that uses `mbrtowc()`

```

/* This program is compiled with LOCALETYPE(*LOCALE) and */
/* SYSIFCOPT(*IFSIO) */

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>
#include <errno.h>

#define LOCNAME "/qsys.lib/JA_JP.locale"
#define LOCNAME_EN "/qsys.lib/EN_US.locale"

int main(void)
{
    int length, sl = 0;
    char string[10];
    wchar_t buffer[10];
    mbstate_t ps = 0;
    memset(string, '\\0', 10);
    string[0] = 0xC1;
    string[1] = 0x0E;
    string[2] = 0x41;
    string[3] = 0x71;
    string[4] = 0x41;
    string[5] = 0x72;
    string[6] = 0x0F;
    string[7] = 0xC2;
    /* In this first example we will convert */
    /* a multibyte character when the CCSID of locale */
    /* associated with LC_CTYPE is 37. */
    /* For single byte cases the state will always */
    /* remain in the initial state 0 */

    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");

    length = mbrtowc(buffer, string, MB_CUR_MAX, &ps);

    /* In this case length is 1, and C1 is converted 0x00C1 */

    printf("length = %d, state = %d\n\n", length, ps);
    printf("MB_CUR_MAX: %d\n\n", MB_CUR_MAX);

    /* Now lets try a multibyte example. We first must set the */
    /* locale to a multibyte locale. We choose a locale with */
    /* CCSID 5026 */

    if (setlocale(LC_ALL, LOCNAME) == NULL)
        printf("setlocale failed.\n");

    length = mbrtowc(buffer, string, MB_CUR_MAX, &ps);

    /* The first is single byte so length is 1 and */
    /* the state is still the initial state 0. C1 is converted */
    /* to 0x00C1 */

    printf("length = %d, state = %d\n\n", length, ps);
    printf("MB_CUR_MAX: %d\n\n", MB_CUR_MAX);

    sl += length;
}

```

```

length = mbrtowc(&buffer[1], &string[s1], MB_CUR_MAX, &ps);

/* The next character is a mixed byte. Length is 3 to */
/* account for the shiftout 0x0e. State is */
/* changed to double byte state. 0x4171 is copied into */
/* the buffer */

printf("length = %d, state = %d\n\n", length, ps);

s1 += length;

length = mbrtowc(&buffer[2], &string[s1], MB_CUR_MAX, &ps);

/* The next character is also a double byte character. */
/* The state is changed to initial state since this was */
/* the last double byte character. Length is 3 to */
/* account for the ending 0x0f shiftin. 0x4172 is copied */
/* into the buffer. */

printf("length = %d, state = %d\n\n", length, ps);

s1 += length;

length = mbrtowc(&buffer[3], &string[s1], MB_CUR_MAX, &ps);

/* The next character is single byte so length is 1 and */
/* state remains in initial state. 0xC2 is converted to */
/* 0x00C2. The buffer now has the value: */
/* 0x00C14171417200C2 */

printf("length = %d, state = %d\n\n", length, ps);
}
/* The output should look like this:

length = 1, state = 0

MB_CUR_MAX: 1

length = 1, state = 0

MB_CUR_MAX: 4

length = 3, state = 2

length = 3, state = 0

length = 1, state = 0
*/

```

## Related Information

- “mblen() — Determine Length of a Multibyte Character” on page 197
- “mbrlen() — Determine Length of a Multibyte Character (Restartable)” on page 199
- “mbsrtowcs() — Convert a Multibyte String to a Wide Character String (Restartable)” on page 206
- “setlocale() — Set Locale” on page 339
- “wctomb() — Convert a Wide Character to a Multibyte Character (Restartable)” on page 446
- “wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)” on page 473
- “<locale.h>” on page 7
- “<wchar.h>” on page 18

---

## mbsinit() — Test State Object for Initial State

### Format

```
#include <wchar.h>
int mbsinit (const mbstate_t *ps);
```

**Language Level:** ANSI

**Threadsafe:** Yes

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the  
| current locale. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation  
| command. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

If *ps* is not a null pointer, the `mbsinit()` function specifies whether the pointed to `mbstate_t` object describes an initial conversion state.

### Return Value

The `mbsinit()` function returns nonzero if *ps* is a null pointer or if the pointed to object describes an initial conversion state. Otherwise, it returns zero.

### Example that uses `mbsinit()`

This example checks the conversion state to see if it is 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

- “`mbrlen()` — Determine Length of a Multibyte Character (Restartable)” on page 199
- “`mbrtowc()` — Convert a Multibyte Character to a Wide Character (Restartable)” on page 201
- “`mbsrtowcs()` — Convert a Multibyte String to a Wide Character String (Restartable)” on page 206
- “`setlocale()` — Set Locale” on page 339
- “`wcrtomb()` — Convert a Wide Character to a Multibyte Character (Restartable)” on page 446
- “`wcsrtombs()` — Convert Wide Character String to Multibyte String (Restartable)” on page 473
- “`<locale.h>`” on page 7
- “`<wchar.h>`” on page 18

---

## mbsrtowcs() — Convert a Multibyte String to a Wide Character String (Restartable)

### Format

```
#include <wchar.h>
size_t mbsrtowcs (wchar_t *dst, const char **src, size_t len,
                 mbstate_t *ps);
```

**Language Level:** ANSI

**Threadsafe:** Yes, if *ps* is not NULL.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. This function might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

This function is the restartable version of `mbstowcs()`.

The `mbsrtowcs()` function converts a sequence of multibyte characters that begins in the conversion state described by *ps* from the array indirectly pointed to by *src* into a sequence of corresponding wide characters. It then stores the converted characters into the array pointed to by *dst*.

Conversion continues up to and including an ending null character, which is also stored. Conversion will stop earlier in two cases: when a sequence of bytes are reached that do not form a valid multibyte character, or (if *dst* is not a null pointer) when *len* wide characters have been stored into the array pointed to by *dst*. Each conversion takes place as if by a call to `mbrtowc()` function.

If *dst* is not a null pointer, the pointer object pointed to by *src* will be assigned either a null pointer (if conversion stopped due to reaching an ending null character) or the address just past the last multibyte character converted. If conversion stopped due to reaching an ending null character, the initial conversion state is described.

### Return Value

If the input string does not begin with a valid multibyte character, an encoding error occurs, the `mbsrtowcs()` function stores the value of the macro `EILSEQ` in `errno`, and returns `(size_t) -1`, but the conversion state will be unchanged. Otherwise, it returns the number of multibyte characters successfully converted, which is the same as the number of array elements modified when *dst* is not a null pointer.

If a conversion error occurs, `errno` might be set to `ECONVERT`.

**Example that uses `mbsrtowcs()`**

```

#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <locale.h>

#define SIZE 10

int main(void)
{
    char        mbs1[] = "abc";
    char        mbs2[] = "\x81\x41" "m" "\x81\x42";
    const char  *pmbs1 = mbs1;
    const char  *pmbs2 = mbs2;
    mbstate_t   ss1 = 0;
    mbstate_t   ss2 = 0;
    wchar_t     wcs1[SIZE], wcs2[SIZE];

    if (NULL == setlocale(LC_ALL, "/qsys.lib/locale.lib/ja_jp939.locale"))
    {
        printf("setlocale failed.\n");
        exit(EXIT_FAILURE);
    }
    mbsrtowcs(wcs1, &pmbs1, SIZE, &ss1);
    mbsrtowcs(wcs2, &pmbs2, SIZE, &ss2);
    printf("The first wide character string is %ls.\n", wcs1);
    printf("The second wide character string is %ls.\n", wcs2);
    return 0;
}

```

```

/*****
The output should be similar to:

The first wide character string is abc.
The second wide character string is Am B.
*****/

```

Also, see the examples for “mbrtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 201.

### Related Information

- “mblen() — Determine Length of a Multibyte Character” on page 197
- “mbrlen() — Determine Length of a Multibyte Character (Restartable)” on page 199
- “mbrtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 201
- “mbstowcs() — Convert a Multibyte String to a Wide Character String”
- “setlocale() — Set Locale” on page 339
- “wctomb() — Convert a Wide Character to a Multibyte Character (Restartable)” on page 446
- “wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)” on page 473
- “<locale.h>” on page 7
- “<wchar.h>” on page 18

---

## mbstowcs() — Convert a Multibyte String to a Wide Character String

### Format

```

#include <stdlib.h>
size_t mbstowcs(wchar_t *pwc, const char *string, size_t n);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

- | **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. This function might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.
- | **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `mbstowcs()` function determines the length of the sequence of the multibyte characters pointed to by *string*. It then converts the multibyte character string that begins in the initial shift state into a wide character string, and stores the wide characters into the buffer that is pointed to by *pwc*. A maximum of *n* wide characters are written.

### Return Value

The `mbstowcs()` function returns the number of wide characters generated, not including any ending null wide characters. If a multibyte character that is not valid is encountered, the function returns `(size_t)-1`.

If a conversion error occurs, `errno` might be set to **ECONVERT**.

**Examples that use `mbstowcs()`**

```

/* This program is compiled with LOCALETYPE(*LOCALEUCS2) and */
/* SYSIFCOPT(*IFSIO) */

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>
#include <errno.h>

#define LOCNAME "qsys.lib/JA_JP.locale"
#define LOCNAME_EN "qsys.lib/EN_US.locale"

int main(void)
{
    int length, sl = 0;
    char string[10];
    char string2[] = "ABC";
    wchar_t buffer[10];
    memset(string, '\0', 10);
    string[0] = 0xC1;
    string[1] = 0x0E;
    string[2] = 0x41;
    string[3] = 0x71;
    string[4] = 0x41;
    string[5] = 0x72;
    string[6] = 0x0F;
    string[7] = 0xC2;
    /* In this first example we will convert */
    /* a multibyte character when the CCSID of locale */
    /* associated with LC_CTYPE is 37. */

    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");

    length = mbstowcs(buffer, string2, 10);

    /* In this case length ABC is converted to UNICODE ABC */
    /* or 0x004100420043. Length will be 3. */

    printf("length = %d\n\n", length);

    /* Now lets try a multibyte example. We first must set the */
    /* locale to a multibyte locale. We choose a locale with */
    /* CCSID 5026 */

    if (setlocale(LC_ALL, LOCNAME) == NULL)
        printf("setlocale failed.\n");

    length = mbstowcs(buffer, string, 10);

    /* The buffer now has the value: */
    /* 0x004103A103A30042 length is 4 */

    printf("length = %d\n\n", length);
}
/* The output should look like this:

length = 3

length = 4

*/

```

```

/* This program is compiled with LOCALETYPE(*LOCALE) and */
/* SYSIFCOPT(*IFSIO) */

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>
#include <errno.h>

#define LOCNAME "qsys.lib/JA_JP.locale"
#define LOCNAME_EN "qsys.lib/EN_US.locale"

int main(void)
{
    int length, sl = 0;
    char string[10];
    char string2[] = "ABC";
    wchar_t buffer[10];
    memset(string, '\0', 10);
    string[0] = 0xC1;
    string[1] = 0x0E;
    string[2] = 0x41;
    string[3] = 0x71;
    string[4] = 0x41;
    string[5] = 0x72;
    string[6] = 0x0F;
    string[7] = 0xC2;
    /* In this first example we will convert */
    /* a multibyte character when the CCSID of locale */
    /* associated with LC_CTYPE is 37. */

    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");

    length = mbstowcs(buffer, string2, 10);

    /* In this case length ABC is converted to */
    /* 0x00C100C200C3. Length will be 3. */

    printf("length = %d\n\n", length);

    /* Now lets try a multibyte example. We first must set the *
    /* locale to a multibyte locale. We choose a locale with
    /* CCSID 5026 */

    if (setlocale(LC_ALL, LOCNAME) == NULL)
        printf("setlocale failed.\n");

    length = mbstowcs(buffer, string, 10);

    /* The buffer now has the value: */
    /* 0x00C14171417200C2 length is 4 */

    printf("length = %d\n\n", length);
}

/* The output should look like this:

length = 3
length = 4
*/

```

## Related Information

- “mblen() — Determine Length of a Multibyte Character” on page 197

- “mbtowc() — Convert Multibyte Character to a Wide Character”
- “setlocale() — Set Locale” on page 339
- “wcslen() — Calculate Length of Wide-Character String” on page 461
- “wcstombs() — Convert Wide-Character String to Multibyte String” on page 483
- “<locale.h>” on page 7
- “<stdlib.h>” on page 17
- “<wchar.h>” on page 18

---

## mbtowc() — Convert Multibyte Character to a Wide Character

### Format

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *string, size_t n);
```

**Language Level:** ANSI

**Threadsafe:** No. Use mbrtowc() instead.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. This function might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The mbtowc() function first determines the length of the multibyte character pointed to by *string*. It then converts the multibyte character to a wide character as described in mbstowcs. A maximum of *n* bytes are examined.

### Return Value

If *string* is NULL, the mbtowc() function returns:

- Nonzero when the active locale is mixed byte. The function initializes the state variable.
- 0 otherwise.

If *string* is not NULL, the mbtowc() function returns:

- 0 if *string* points to the null character
- The number of bytes comprising the converted multibyte character
- -1 if *string* does not point to a valid multibyte character.

If a conversion error occurs, errno might be set to ECONVERT.

### Example that uses mbtowc()

This example uses the mblen() and mbtowc() functions to convert a multibyte character into a single wide character.

```

#include <stdio.h>
#include <stdlib.h>

#define LOCNAME "qsys.lib/mylib.lib/ja_jp959.locale"
/*Locale created from source JA_JP and CCSID 939 */

int length, temp;
char string [] = "\x0e\x41\x71\xf";
wchar_t arr[6];

int main(void)
{
    /* initialize internal state variable */
    temp = mbtowc(arr, NULL, 0);

    setlocale (LC_ALL, LOCNAME);
    /* Set string to point to a multibyte character. */
    length = mblen(string, MB_CUR_MAX);
    temp = mbtowc(arr, string, length);
    arr[1] = L'\0';
    printf("wide character string: %ls", arr);
}

```

### Related Information

- “mblen() — Determine Length of a Multibyte Character” on page 197
- “mbstowcs() — Convert a Multibyte String to a Wide Character String” on page 207
- “wcslen() — Calculate Length of Wide-Character String” on page 461
- “wctomb() — Convert Wide Character to Multibyte Character” on page 492
- “<stdlib.h>” on page 17

---

## memchr() — Search Buffer

### Format

```

#include <string.h>
void *memchr(const void *buf, int c, size_t count);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `memchr()` function searches the first *count* bytes of *buf* for the first occurrence of *c* converted to an unsigned character. The search continues until it finds *c* or examines *count* bytes.

### Return Value

The `memchr()` function returns a pointer to the location of *c* in *buf*. It returns NULL if *c* is not within the first *count* bytes of *buf*.

### Example that uses `memchr()`

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.

```

#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 should be similar to: *****/

The string starting with x is xing
*/

```

### Related Information

- “memcmp() — Compare Buffers”
- “memcpy() — Copy Bytes” on page 214
- “memmove() — Copy Bytes” on page 217
- “wmemchr() — Locate Wide Character in Wide-Character Buffer” on page 498
- “memset() — Set Bytes to Value” on page 218
- “strchr() — Search for Character” on page 359
- “<string.h>” on page 17

---

## memcmp() — Compare Buffers

### Format

```

#include <string.h>
int memcmp(const void *buf1, const void *buf2, size_t count);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `memcmp()` function compares the first *count* bytes of *buf1* and *buf2*.

### Return Value

The `memcmp()` function returns a value indicating the relationship between the two buffers as follows:

Value	Meaning
Less than 0	<i>buf1</i> less than <i>buf2</i>
0	<i>buf1</i> identical to <i>buf2</i>
Greater than 0	<i>buf1</i> greater than <i>buf2</i>

**Example that uses `memcmp()`**

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] );
    }
}

/***** If the program is passed the arguments *****/
/***** firststring and secondstring, *****/
/***** output should be: *****/

When the first 11 characters are compared,
"firststring" is less than "secondstring"
*****/
```

## Related Information

- “`memchr()` — Search Buffer” on page 212
- “`memcpy()` — Copy Bytes”
- “`wmemcmp()` — Compare Wide-Character Buffers” on page 499
- “`memmove()` — Copy Bytes” on page 217
- “`memset()` — Set Bytes to Value” on page 218
- “`strcmp()` — Compare Strings” on page 360
- “`<string.h>`” on page 17

---

## `memcpy()` — Copy Bytes

### Format

```
#include <string.h>
void *memcpy(void *dest, const void *src, size_t count);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

## Description

The `memcpy()` function copies *count* bytes of *src* to *dest*. The behavior is undefined if copying takes place between objects that overlap. The `memmove()` function allows copying between objects that might overlap.

## Return Value

The `memcpy()` function returns a pointer to *dest*.

## Example that uses `memcpy()`

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 );
}

/***** Expected output: *****/

Before memcpy, target is "This is the target string"
After memcpy, target becomes "This is the source string"
*/
```

## Related Information

- “`memchr()` — Search Buffer” on page 212
- “`memcmp()` — Compare Buffers” on page 213
- “`wmemcpy()` — Copy Wide-Character Buffer” on page 500
- “`memmove()` — Copy Bytes” on page 217
- “`memset()` — Set Bytes to Value” on page 218
- “`strcpy()` — Copy Strings” on page 364
- “`<string.h>`” on page 17

---

## `memcmp()` - Compare Bytes

### Format

```
#include <string.h> // also in <memory.h>
int memcmp(void *buf1, void *buf2, unsigned int cnt);
```

**Note:** The `memcmp` function is available for C++ programs. It is available for C only when the program defines the `__cplusplus__strings__` macro.

**Language Level:** Extension

**Threadsafe:** Yes.

- l **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

## Description

The `memicmp` function compares the first *cnt* bytes of *buf1* and *buf2* without regard to the case of letters in the two buffers. The function converts all uppercase characters into lowercase and then performs the comparison.

## Return Value

The return value of `memicmp` indicates the result as follows:

Value	Meaning
Less than 0	<i>buf1</i> less than <i>buf2</i>
0	<i>buf1</i> identical to <i>buf2</i>
Greater than 0	<i>buf1</i> greater than <i>buf2</i>

## Example that uses `memicmp()`

This example copies two strings that each contain a substring of 29 characters that are the same except for case. The example then compares the first 29 bytes without regard to case.

```
#include <stdio.h>
#include <string.h>
char first[100],second[100];
int main(void)
{
    int result;
    strcpy(first, "Those Who Will Not Learn From History");
    strcpy(second, "THOSE WHO WILL NOT LEARN FROM their mistakes");
    printf("Comparing the first 29 characters of two strings.\n");
    result = memicmp(first, second, 29);
    printf("The first 29 characters of String 1 are ");
    if (result < 0)
        printf("less than String 2.\n");
    else
        if (0 == result)
            printf("equal to String 2.\n");
        else
            printf("greater than String 2.\n");
    return 0;
}
```

The output should be:

```
Comparing the first 29 characters of two strings.
The first 29 characters of String 1 are equal to String 2
```

## Related Information:

- “`memchr()` — Search Buffer” on page 212
- “`memcmp()` — Compare Buffers” on page 213
- “`memcpy()` — Copy Bytes” on page 214
- “`memmove()` — Copy Bytes” on page 217
- “`memset()` — Set Bytes to Value” on page 218
- “`strcmp()` — Compare Strings” on page 360
- “`strncmpi()` - Compare Strings Without Case Sensitivity” on page 362
- “`stricmp()` - Compare Strings without Case Sensitivity” on page 374

- “strnicmp - Compare Substrings Without Case Sensitivity” on page 382
- “<string.h>” on page 17

---

## memmove() — Copy Bytes

### Format

```
#include <string.h>
void *memmove(void *dest, const void *src, size_t count);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `memmove()` function copies *count* bytes of *src* to *dest*. This function allows copying between objects that might overlap as if *src* is first copied into a temporary array.

### Return Value

The `memmove()` function returns a pointer to *dest*.

### Example that uses `memmove()`

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 );
}

/***** Expected output: *****/

Before memmove, target is "a shiny white sphere"
After memmove, target becomes "a shiny shiny sphere"
*/
```

### Related Information

- “`memchr()` — Search Buffer” on page 212
- “`memcmp()` — Compare Buffers” on page 213
- “`wmemmove()` — Copy Wide-Character Buffer” on page 501
- “`memcpy()` — Copy Bytes” on page 214
- “`memset()` — Set Bytes to Value” on page 218
- “`strcpy()` — Copy Strings” on page 364
- “<string.h>” on page 17

---

## memset() — Set Bytes to Value

### Format

```
#include <string.h>
void *memset(void *dest, int c, size_t count);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `memset()` function sets the first *count* bytes of *dest* to the value *c*. The value of *c* is converted to an unsigned character.

### Return Value

The `memset()` function returns a pointer to *dest*.

### Example that uses `memset()`

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

int main(void)
{
    char buffer[BUF_SIZE + 1];
    char *string;

    memset(buffer, 0, sizeof(buffer));
    string = (char *) memset(buffer, 'A', 10);
    printf("\nBuffer contents: %s\n", string);
    memset(buffer+10, 'B', 10);
    printf("\nBuffer contents: %s\n", buffer);
}

/***** Output should be similar to: *****/

Buffer contents: AAAAAAAAAA

Buffer contents: AAAAAAAAAABBBBBBBBBB
*/
```

### Related Information

- “`memchr()` — Search Buffer” on page 212
- “`memcmp()` — Compare Buffers” on page 213
- “`memcpy()` — Copy Bytes” on page 214
- “`memmove()` — Copy Bytes” on page 217
- “`wmemset()` — Set Wide Character Buffer to a Value” on page 502
- “`<string.h>`” on page 17

---

## mktime() — Convert Local Time

### Format

```
#include <time.h>
time_t mktime(struct tm *time);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_TOD category of the current  
| locale.

### Description

The `mktime()` function converts a stored `tm` structure (assume to be in job local time) pointed to by `time`, into a `time_t` structure suitable for use with other time functions. After the conversion, the `time_t` structure will be considered Universal Coordinate Time (UTC). For this conversion, `mktime()` checks the current locale setting for local time zone and daylight saving time (DST). If these values are not set in the current locale, `mktime()` gets the local time zone and daylight saving time settings from the current job. If the DST is set in the locale but the time zone information is not, the DST information in the locale is ignored. `mktime()` then uses the current time zone information to determine UTC.

The values of some structure elements pointed to by `time` are not restricted to the ranges shown for `gmtime()`.

The values of `tm_wday` and `tm_yday` passed to `mktime()` are ignored and are assigned their correct values on return.

| A positive or 0 value for `tm_isdst` causes `mktime()` to presume initially that DST, respectively, is or is not  
| in effect for the specified time. A negative value for `tm_isdst` causes `mktime()` to attempt to determine  
| whether DST is in effect for the specified time.

### Return Value

The `mktime()` function returns Universal Coordinate Time (UTC) having type `time_t`. The value `(time_t)(-1)` is returned if the Universal Coordinate Time cannot be represented.

### Example that uses `mktime()`

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 should be similar to: *****/

40 days and 16 hours from now, it will be a Sunday
*/

```

### Related Information

- “asctime() — Convert Time to Character String” on page 39
- “asctime\_r() — Convert Time to Character String (Restartable)” on page 41
- “ctime() — Convert Time to Character String” on page 71
- | • “ctime64() — Convert Time to Character String” on page 73
- | • “ctime64\_r() — Convert Time to Character String (Restartable)” on page 76
- “ctime\_r() — Convert Time to Character String (Restartable)” on page 74
- “gmtime() — Convert Time” on page 161
- | • “gmtime64() — Convert Time” on page 163
- | • “gmtime64\_r() — Convert Time (Restartable)” on page 167
- “gmtime\_r() — Convert Time (Restartable)” on page 165
- “localtime() — Convert Time” on page 185
- | • “localtime64() — Convert Time” on page 187
- | • “localtime64\_r() — Convert Time (Restartable)” on page 189
- “localtime\_r() — Convert Time (Restartable)” on page 188
- | • “mktime64() — Convert Local Time”
- “time() — Determine Current Time” on page 411
- | • “time64() — Determine Current Time” on page 412
- “<time.h>” on page 18

---

## | mktime64() — Convert Local Time

### | Format

```

#include <time.h>
time64_t mktime64(struct tm *time);

```

| **Language Level:** ILE C Extension

| **Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_TOD category of the current locale.

### | **Description**

| The `mktime64()` function converts a stored `tm` structure (assumed to be in job local time) pointed to by `time`, into a `time64_t` value suitable for use with other time functions. After the conversion, the `time64_t` value will be considered Universal Coordinate Time (UTC). For this conversion, `mktime64()` checks the current locale settings for the local time zone and daylight saving time (DST). If these values are not set in the current locale, `mktime64()` gets the local time zone and DST settings from the current job. If the DST is set in the locale but the time zone information is not, the DST information in the locale is ignored. The `mktime64()` function then uses the time zone information of the current job to determine UTC.

| The values of some structure elements pointed to by `time` are not restricted to the ranges shown for `gmtime64()`.

| The values of `tm_wday` and `tm_yday` passed to `mktime64()` are ignored and are assigned their correct values on return.

| A positive or 0 value for `tm_isdst` causes `mktime()` to presume initially that DST, respectively, is or is not in effect for the specified time. A negative value for `tm_isdst` causes `mktime()` to attempt to determine whether DST is in effect for the specified time.

| **Note:** The supported date and time range for this function is 01/01/1970 00:00:00 through 12/31/9999 23:59:59.

### | **Return Value**

| The `mktime64()` function returns Universal Coordinate Time (UTC) having type `time64_t`. The value `(time_t)(-1)` is returned if the Universal Coordinate Time cannot be represented or if the given `time` is out of range. If the given `time` is out of range, `errno` is set to `EOverflow`.

### | **Example that uses `mktime64()`**

| 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)
| {
|     time64_t t1, t3;
|     struct tm *t2;
|
|     t1 = time64(NULL);
|     t2 = localtime64(&t1);
|     t2 -> tm_mday += 40;
|     t2 -> tm_hour += 16;
|     t3 = mktime64(t2);
|
|     printf("40 days and 16 hours from now, it will be a %s \n",
|           wday[t2 -> tm_wday]);
| }
|
| /***** Output should be similar to: *****/
| 40 days and 16 hours from now, it will be a Sunday
| */

```

### Related Information

- | • “asctime() — Convert Time to Character String” on page 39
- | • “asctime\_r() — Convert Time to Character String (Restartable)” on page 41
- | • “ctime() — Convert Time to Character String” on page 71
- | • “ctime64() — Convert Time to Character String” on page 73
- | • “ctime64\_r() — Convert Time to Character String (Restartable)” on page 76
- | • “ctime\_r() — Convert Time to Character String (Restartable)” on page 74
- | • “gmtime() — Convert Time” on page 161
- | • “gmtime64() — Convert Time” on page 163
- | • “gmtime64\_r() — Convert Time (Restartable)” on page 167
- | • “gmtime\_r() — Convert Time (Restartable)” on page 165
- | • “localtime() — Convert Time” on page 185
- | • “localtime64() — Convert Time” on page 187
- | • “localtime64\_r() — Convert Time (Restartable)” on page 189
- | • “localtime\_r() — Convert Time (Restartable)” on page 188
- | • “mktime() — Convert Local Time” on page 218
- | • “time() — Determine Current Time” on page 411
- | • “time64() — Determine Current Time” on page 412
- | • “<time.h>” on page 18

---

## modf() — Separate Floating-Point Value

### Format

```

#include <math.h>
double modf(double x, double *intptr);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

## Description

The `modf()` function breaks down the floating-point value  $x$  into fractional and integral parts. The signed fractional portion of  $x$  is returned. The integer portion is stored as a double value pointed to by `intptr`. Both the fractional and integral parts are given the same sign as  $x$ .

## Return Value

The `modf()` function returns the signed fractional portion of  $x$ .

## Example that uses `modf()`

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 should be similar to: *****/

x = -14.876000
Integral part = -14.000000
Fractional part = -0.876000
*/
```

## Related Information

- “`fmod()` — Calculate Floating-Point Remainder” on page 108
- “`frexp()` — Separate Floating-Point Value” on page 132
- “`ldexp()` — Multiply by a Power of Two” on page 178
- “`<math.h>`” on page 8

---

## **nextafter() — nextafterl() — nexttoward() — nexttowardl() — Calculate the Next Representable Floating-Point Value**

### Format

```
#include <math.h>
double nextafter(double x, double y);
long double nextafterl(long double x, long double y);
double nexttoward(double x, long double y);
long double nexttowardl(long double x, long double y);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

| The `nextafter()`, `nextafterl()`, `nexttoward()`, and `nexttowardl()` functions calculate the next representable value after `x` in the direction of `y`.

### | Return Value

| The `nextafter()`, `nextafterl()`, `nexttoward()`, and `nexttowardl()` functions return the next representable value after `x` in the direction of `y`. If `x` is equal to `y`, they return `y`. If `x` or `y` is NaN (Not a Number), NaN is returned and `errno` is set to `EDOM`. If `x` is the largest finite value and the result is infinite or not representable, `HUGE_VAL` is returned and `errno` is set to `ERANGE`.

### | Example that uses `nextafter()`, `nextafterl()`, `nexttoward()`, and `nexttowardl()`

| This example converts a floating-point value to the next greater representable value and next smaller representable value. It prints out the converted values.

```
|
| #include <stdio.h>
| #include <math.h>
| int main(void)
| {
|     double x, y;
|     long double ld;
|
|     x = nextafter(1.234567899, 10.0);
|     printf("nextafter 1.234567899 is %#19.17g\n" x);
|     ld = nextafterl(1.234567899, -10.0);
|     printf("nextafterl 1.234567899 is %#19.17g\n" ld);
|
|     x = nexttoward(1.234567899, 10.0);
|     printf("nexttoward 1.234567899 is %#19.17g\n" x);
|     ld = nexttowardl(1.234567899, -10.0);
|     printf("nexttowardl 1.234567899 is %#19.17g\n" ld);
|
| }
|
| /***** Output should be similar to: *****/
| nextafter 1.234567899 is 1.23456789900000002
| nextafterl 1.234567899 is 1.23456789899999997
| nexttoward 1.234567899 is 1.23456789900000002
| nexttowardl 1.234567899 is 1.23456789899999997
|
| */
```

### | Related Information

- | • “`ceil()` — Find Integer  $\geq$ Argument” on page 61
- | • “`floor()` — Find Integer  $\leq$ Argument” on page 107
- | • “`frexp()` — Separate Floating-Point Value” on page 132
- | • “`modf()` — Separate Floating-Point Value” on page 222
- | • “`<math.h>`” on page 8

---

## nl\_langinfo() —Retrieve Locale Information

### Format

```
#include <langinfo.h>
#include <nl_types.h>
char *nl_langinfo(nl_item item);
```

**Language Level:** XPG4

**Threadsafe:** No.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE, LC\_MESSAGES,  
 | LC\_MONETARY, LC\_NUMERIC, and LC\_TIME categories of the current locale. This function is not  
 | available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information,  
 | see “Understanding CCSIDs and Locales” on page 524.

## Description

The `nl_langinfo()` function retrieves from the current locale the string that describes the requested information specified by *item*.

The retrieval of the following information from the current locale is supported:

Item	Explanation
CODESET	CCSID of locale in character form
D_T_FMT	string for formatting date and time
D_FMT	date format string
T_FMT	time format string
T_FMT_AMPM	a.m. or p.m. time format string
AM_STR	Ante Meridian affix
PM_STR	Post Meridian affix
DAY_1	name of the first day of the week (for example, Sunday)
DAY_2	name of the second day of the week (for example, Monday)
DAY_3	name of the third day of the week (for example, Tuesday)
DAY_4	name of the fourth day of the week (for example, Wednesday)
DAY_5	name of the fifth day of the week (for example, Thursday)
DAY_6	name of the sixth day of the week (for example, Friday)
DAY_7	name of the seventh day of the week (for example, Saturday)
ABDAY_1	abbreviated name of the first day of the week
ABDAY_2	abbreviated name of the second day of the week
ABDAY_3	abbreviated name of the third day of the week
ABDAY_4	abbreviated name of the fourth day of the week
ABDAY_5	abbreviated name of the fifth day of the week
ABDAY_6	abbreviated name of the sixth day of the week
ABDAY_7	abbreviated name of the seventh day of the week
MON_1	name of the first month of the year
MON_2	name of the second month of the year
MON_3	name of the third month of the year
MON_4	name of the fourth month of the year
MON_5	name of the fifth month of the year
MON_6	name of the sixth month of the year
MON_7	name of the seventh month of the year
MON_8	name of the eighth month of the year
MON_9	name of the ninth month of the year
MON_10	name of the tenth month of the year
MON_11	name of the eleventh month of the year

MON_12	name of the twelfth month of the year
ABMON_1	abbreviated name of the first month of the year
ABMON_2	abbreviated name of the second month of the year
ABMON_3	abbreviated name of the third month of the year
ABMON_4	abbreviated name of the fourth month of the year
ABMON_5	abbreviated name of the fifth month of the year
ABMON_6	abbreviated name of the sixth month of the year
ABMON_7	abbreviated name of the seventh month of the year
ABMON_8	abbreviated name of the eighth month of the year
ABMON_9	abbreviated name of the ninth month of the year
ABMON_10	abbreviated name of the tenth month of the year
ABMON_11	abbreviated name of the eleventh month of the year
ABMON_12	abbreviated name of the twelfth month of the year
ERA	era description segments
ERA_D_FMT	era date format string
ERA_D_T_FMT	era date and time format string
ERA_T_FMT	era time format string
ALT_DIGITS	alternative symbols for digits
RADIXCHAR	radix character
THOUSEP	separator for thousands
YESEXPR	affirmative response expression
NOEXPR	negative response expression
YESSTR	affirmative response for yes/no queries
NOSTR	negative response for yes/no queries
CRNCYSTR	currency symbol, preceded by '-' if the symbol should appear before the value, '+' if the symbol should appear after the value, or '.' if the symbol should replace the radix character

### Returned Value

The `nl_langinfo()` function returns a pointer to a null-ended string containing information concerning the active language or cultural area. The active language or cultural area is determined by the most recent `setlocale()` call. The array pointed to by the returned value is modified by subsequent calls to the function. The array should not be changed by the user's program.

If the item is not valid, the function returns a pointer to an empty string.

### Example that uses `nl_langinfo()`

This example retrieves the name of the codeset using the `nl_langinfo()` function.

```

#include <langinfo.h>
#include <locale.h>
#include <nl_types.h>
#include <stdio.h>

int main(void)
{
    printf("Current codeset is %s\n", nl_langinfo(CODESET));
    return 0;
}

/*****

The output should be similar to:

Current codeset is 37

*****/

```

### Related Information

- “localeconv() — Retrieve Information from the Environment” on page 181
- “setlocale() — Set Locale” on page 339
- “<langinfo.h>” on page 7
- “<nl\_types.h>” on page 9

---

## perror() — Print Error Message

### Format

```

#include <stdio.h>
void perror(const char *string);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `perror()` function prints an error message to `stderr`. If *string* is not `NULL` and does not point to a null character, the string pointed to by *string* is printed to the standard error stream, followed by a colon and a space. The message associated with the value in `errno` is then printed followed by a new-line character.

To produce accurate results, you should ensure that the `perror()` function is called immediately after a library function returns with an error; otherwise, subsequent calls might alter the `errno` value.

### Return Value

There is no return value.

The value of `errno` can be set to:

#### Value Meaning

##### EBADDDATA

The message data is not valid.

##### EBUSY

The record or file is in use.

**ENOENT**

The file or library cannot be found.

**EPERM**

Insufficient authorization for access.

**ENOREC**

Record not found.

**EIOERROR**

A non-recoverable I/O error occurred.

**EIORECERR**

A recoverable I/O error occurred.

**Example that uses perror()**

This example tries to open a stream. If `fopen()` fails, the example prints a message and ends the program.

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

int main(void)
{
    FILE *fh;

    if ((fh = fopen("mylib/myfile", "r")) == NULL)
    {
        perror("Could not open data file");
        abort();
    }
}
```

**Related Information**

- “clearerr() — Reset Error Indicators” on page 62
- “ferror() — Test for Read/Write Errors” on page 95
- “strerror() — Set Pointer to Runtime Error Message” on page 367
- “<stdio.h>” on page 15

---

## pow() — Compute Power

**Format**

```
#include <math.h>
double pow(double x, double y);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Description**

The `pow()` function calculates the value of  $x$  to the power of  $y$ .

**Return Value**

If  $y$  is 0, the `pow()` function returns the value 1. If  $x$  is 0 and  $y$  is negative, the `pow()` function sets `errno` to `EDOM` and returns 0. If both  $x$  and  $y$  are 0, or if  $x$  is negative and  $y$  is not an integer, the `pow()` function sets `errno` to `EDOM`, and returns 0. The `errno` variable can also be set to `ERANGE`. If an overflow results, the `pow()` function returns `+HUGE_VAL` for a large result or `-HUGE_VAL` for a small result.

## Example that uses pow()

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 should be similar to: *****/

2.000000 to the power of 3.000000 is 8.000000
*/
```

### Related Information

- “exp() — Calculate Exponential Function” on page 89
- “log() — Calculate Natural Logarithm” on page 191
- “log10() — Calculate Base 10 Logarithm” on page 191
- “sqrt() — Calculate Square Root” on page 353
- “<math.h>” on page 8

---

## printf() — Print Formatted Characters

### Format

```
#include <stdio.h>
int printf(const char *format-string, argument-list);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

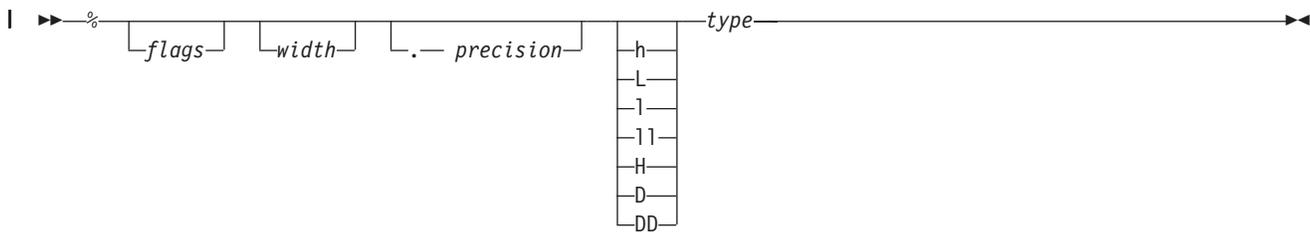
| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. The behavior might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The printf() function formats and prints a series of characters and values to the standard output stream stdout. Format specifications, beginning with a percent sign (%), determine the output format for any argument-list following the format-string. The format-string is a multibyte character string beginning and ending in its initial shift state.

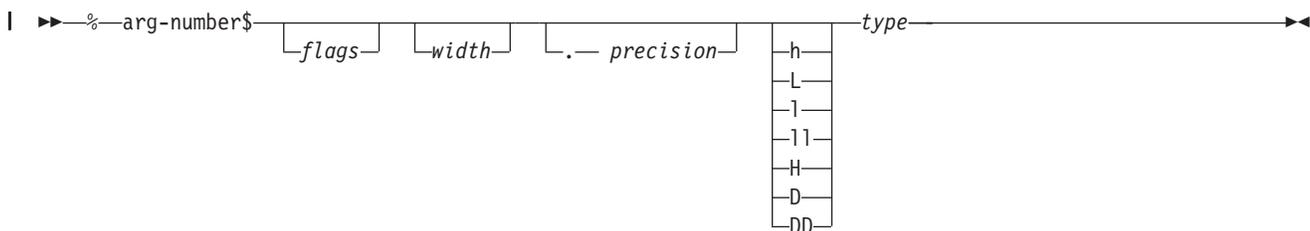
| The format-string is read left to right. When the first format specification is found, the value of the first argument after the format-string is converted and printed according to the format specification. The second format specification causes the second argument after the format-string to be converted and printed, 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.

A format specification has the following form:



Conversions can be applied to the *n*th argument after the *format-string* in the argument list, rather than to the next unused argument. In this case, the conversion character % is replaced by the sequence %*n*\$, where *n* is a decimal integer in the range 1 through `NL_ARGMAX`, giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages.

**Alternative format specification** has the following form:



As an alternative, specific entries in the argument-list can be assigned by using the format specification outlined in the preceding diagram. This format specification and the previous format specification cannot be mixed in the same call to `printf()`. Otherwise, unpredictable results might occur.

The *arg-number* is a positive integer constant where 1 refers to the first entry in the argument-list. *Arg-number* cannot be greater than the number of entries in the argument-list, or else the results are undefined. *Arg-number* also may not be greater than `NL_ARGMAX`.

In format strings containing the %*n*\$ form of conversion specifications, numbered arguments in the argument list can be referenced from the format string as many times as required.

In format strings containing the %*n*\$ form of a conversion specification, a field width or precision may be indicated by the sequence \**m*\$, where *m* is a decimal integer in the range 1 thru `NL_ARGMAX` giving the position in the argument list (after the format argument) of an integer argument containing the field width or precision, for example:

```
printf("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

The *format-string* can contain either numbered argument specifications (that is, %*n*\$ and \**m*\$), or unnumbered argument specifications (that is, % and \*), but normally not both. The only exception to this is that %% can be mixed with the %*n*\$ form. The results of mixing numbered and unnumbered argument specifications in a *format-string* are undefined. When numbered argument specifications are used, specifying the *n*th argument requires that all the leading arguments, from the first to the (*n*-1)th, are specified in the *format string*.

Each field of the format specification is a single character or number signifying a particular format option. The *type* character, which appears after the last optional format field, determines whether the

associated argument is interpreted as a character, a string, a number, or pointer. The simplest format specification contains only the percent sign and a *type* character (for example, %s).

The following optional fields control other aspects of the formatting:

**Field Description**

*flags* Justification of output and printing of signs, blanks, decimal points, octal, and hexadecimal prefixes, and the semantics for wchar\_t precision unit.

*width* Minimum number of bytes output.

*precision*

See Table 4 on page 235.

**h, l, ll, L, H, D, DD**

Size of argument expected:

**h** A prefix with d, i, o, u, x, X, and n types that specifies that the argument is a 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.

**ll** A prefix with d, i, o, u, x, X, and n types that specifies that the argument is a long long int or unsigned long long int.

**L** A prefix with e, E, f, F, g, or G types that specifies that the argument is long double.

**H** A prefix with e, E, f, F, g, or G types that specifies that the argument is \_Decimal32.

**D** A prefix with e, E, f, F, g, or G types that specifies that the argument is \_Decimal64.

**DD** A prefix with e, E, f, F, g, or G types that specifies that the argument is \_Decimal128.

Each field of the format specification is discussed in detail below. 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 *type* characters and their meanings are given in the following table:

Character	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.
D(n,p)	Packed decimal	It has the format [-] <i>ddd.dddd</i> where the number of digits after the decimal point is equal to the precision of the specification. If the precision is missing, the default is p; if the precision is zero, and the # flag is not specified, no decimal point character appears. If the n and the p are *, an argument from the argument list supplies the value. n and p must precede the value being formatted in the argument list. At least one character appears before a decimal point. The value is rounded to the appropriate number of digits.
f	Floating-point	Signed value having the form [-] <i>ddd.dddd</i> , where <i>ddd</i> 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. <sup>2</sup>

Character	Argument	Output Format
F	Floating-point	Identical to the f format except that uppercase alphabetic characters are used instead of lowercase alphabetic characters. <sup>2</sup>
e	Floating-point	Signed value having the form [-]d.dddd e[sign]ddd, where d is a single-decimal digit, dddd is one or more decimal digits, ddd is 2 or 4 decimal digits, and sign is + or -. <sup>2</sup>
E	Floating-point	Identical to the e format except that uppercase alphabetic characters are used instead of lowercase alphabetic characters. <sup>2</sup>
g	Floating-point	Signed value printed in f or e format. The e format is used only when the exponent of the value is less than -4 or greater than <i>precision</i> . Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it. <sup>2</sup>
G	Floating-point	Identical to the g format except that uppercase alphabetic characters are used instead of lowercase alphabetic characters. <sup>2</sup>
c	Character (byte)	Single character.
s	String	Characters (bytes) printed up to the first null character (\0) or until <i>precision</i> is reached.
n	Pointer to integer	Number of characters (bytes) successfully written so far to the <i>stream</i> or buffer; this value is stored in the integer whose address is given as the argument.
p	Pointer	Pointer converted to a sequence of printable characters. It can be one of the following: <ul style="list-style-type: none"> <li>• space pointer</li> <li>• system pointer</li> <li>• invocation pointer</li> <li>• procedure pointer</li> <li>• open pointer</li> <li>• suspend pointer</li> <li>• data pointer</li> <li>• label pointer</li> </ul>
lc or C	Wide Character	The (wchar_t) character is converted to a multibyte character as if by a call to wctomb(), and this character is printed out. <sup>1</sup>
ls or S	Wide Character	The (wchar_t) characters up to the first (wchar_t) null character (L\0), or until <i>precision</i> is reached, are converted to multibyte characters, as if by a call to wcstombs(), and these characters are printed out. If the argument is a null string, (null) is printed. <sup>1</sup>

**Notes:**

1. See the documentation for the wctomb() function or the documentation for the wcstombs() function for more information. You can also find additional information in “Wide Characters” on page 527.
2. If the H, D, or DD format size specifiers are not used, only 15 significant digits of output are guaranteed.

The following list shows the format of the printed values for i5/OS pointers, and gives a brief description of the components of the printed values.

Space pointer: SPP:Context:Object:Offset:AG

Context: type, subtype and name of the context  
Object: type, subtype and name of the object  
Offset: offset within the space

AG: Activation group ID

System pointer: SYP:Context:Object:Auth:Index:AG

Context: type, subtype and name of the context  
Object: type, subtype and name of the object  
Auth: authority  
Index: Index associated with the pointer  
AG: Activation group ID

Invocation pointer: IVP:Index:AG

Index: Index associated with the pointer  
AG: Activation group ID

Procedure pointer: PRP:Index:AG

Index: Index associated with the pointer  
AG: Activation group ID

Suspend pointer: SUP:Index:AG

Index: Index associated with the pointer  
AG: Activation group ID

Data pointer: DTP:Index:AG

Index: Index associated with the pointer  
AG: Activation group ID

Label pointer: LBP:Index:AG

Index: Index associated with the pointer  
AG: Activation group ID

NULL pointer: NULL

The following restrictions apply to pointer printing and scanning on the i5/OS operating system:

- If a pointer is printed out and scanned back from the same activation group, the scanned back pointer will be compared equal to the pointer printed out.
- If a `scanf()` family function scans a pointer that was printed out by a different activation group, the `scanf()` family function will set the pointer to NULL.
- If a pointer is printed out in a Teraspace environment, just the hexadecimal value of the pointer is printed out. These results are the same as when using `%#p`.

See the *WebSphere Development Studio: ILE C/C++ Programmer's Guide* for more information about using i5/OS pointers.

- | If a floating-point value of INFINITY or Not-a-Number (NaN) is formatted using the `e`, `f`, or `g` format, the output string is `infinity` or `nan`. If a floating-point value of INFINITY or Not-A-Number (NaN) is formatted using the `E`, `F`, or `G` format, the output string is `INFINITY` or `NAN`.

The *flag* characters and their meanings are as follows (notice that more than one *flag* can appear in a format specification):

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 (-).
<i>blank</i> (' ')	Prefix the output value with a blank if the output value is signed and positive. The + flag overrides the <i>blank</i> flag if both appear, and a positive signed value will be output with a sign.	No blank.
#	When used with the o, x, or X formats, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively.	No prefix.
	When used with the f, F, D(n,p), e, or E formats, the # flag forces the output value to contain a decimal point in all cases.	Decimal point appears only if digits follow it.
	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 or S format, the # flag causes precision to be measured in characters, regardless of the size of the character. For example, if single-byte characters are being printed, a precision of 4 would result in 4 bytes being printed. If double-byte characters are being printed, a precision of 4 would result in 8 bytes being printed.	Precision indicates the maximum number of bytes to be output.
	When used with the p format, the # flag converts the pointer to hex digits. These hex digits cannot be converted back into a pointer, unless in a Teraspace environment.	Pointer converted to a sequence of printable characters.
0	When used with the d, i, D(n,p) o, u, x, X, e, E, f, F, g, or G formats, the 0 flag causes leading 0s 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. No space padding for D(n,p).

The # flag should not be used with c, lc, d, i, u, or s types.

*Width* is a nonnegative decimal integer controlling the minimum number of characters printed. If the number of characters (bytes) 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 (bytes) in the output value is greater than the specified *width*, or *width* is not given, all characters of the value are printed (subject to the *precision* specification).

For the ls or S type, *width* is specified in bytes. If the number of bytes in the output value is less than the specified width, single-byte blanks are added on the left or the right (depending on whether the - flag is specified) until the minimum width is reached.

The *width* specification can be an asterisk (\*), in which case an argument from the argument list supplies the value. The *width* argument must precede the value being formatted in the argument list.

*Precision* is a nonnegative decimal integer preceded by a period, which specifies the number of characters to be printed 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 or packed decimal value.

The *precision* specification can be an asterisk (\*), in which case an argument from the argument list supplies the value. The *precision* argument must precede the value being formatted in the argument list.

The interpretation of the *precision* value and the default when the *precision* is omitted depend on the *type*, as shown in the following table:

Table 4. Values of Precision

Type	Meaning	Default
i d u o x X	<i>Precision</i> specifies the minimum number of digits to be printed. If the number of digits in the argument is less than <i>precision</i> , the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds <i>precision</i> .	If <i>precision</i> is 0 or omitted entirely, or if the period (.) appears without a number following it, the <i>precision</i> is set to 1.
f F D(n,p) e E	<i>Precision</i> specifies the number of digits to be printed after the decimal point. The last digit printed is rounded.	Default <i>precision</i> for f, F, e and E is six. Default <i>precision</i> for D(n,p) is p. If <i>precision</i> is 0 or the period appears without a number following it, no decimal point is printed.
g G	<i>Precision</i> specifies the maximum number of significant digits printed.	All significant digits are printed. Default <i>precision</i> is six.
c	No effect.	The character is printed.
lc	No effect.	The wchar_t character is converted and resulting multibyte character is printed.
s	<i>Precision</i> specifies the maximum number of characters (bytes) to be printed. Characters (bytes) in excess of <i>precision</i> are not printed.	Characters are printed until a null character is encountered.
ls	<i>Precision</i> specifies the maximum number of bytes to be printed. Bytes in excess of <i>precision</i> are not printed; however, multibyte integrity is always preserved.	wchar_t characters are converted and resulting multibyte characters are printed.

## Return Value

The printf() function returns the number of bytes printed. The value of errno may be set to:

### Value Meaning

#### EBADMODE

The file mode that is specified is not valid.

#### ECONVERT

A conversion error occurred.

#### EIOERROR

A non-recoverable I/O error occurred.

#### EIORECERR

A recoverable I/O error occurred.

#### EILSEQ

An invalid multibyte character sequence was encountered.

#### EPUTANDGET

An illegal write operation occurred after a read operation.

#### ESTDOUT

stdout cannot be opened.

| **Note:** The radix character for the printf() function is locale sensitive. The radix character is the decimal  
 | point to be used for the # flag character of the format string parameter for the format types f, F,  
 | D(n,p), e, E, g, and G.

### Example that uses printf()

This example prints data in a variety of formats.

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

int main(void)
{
    char ch = 'h', *string = "computer";
    int count = 234, hex = 0x10, oct = 010, dec = 10;
    double fp = 251.7366;
    wchar_t wc = (wchar_t)0x0058;
    wchar_t ws[4];

    printf("1234567890123%n4567890123456789\n\n", &count);
    printf("Value of count should be 13; count = %d\n\n", count);
    printf("%10c%5c\n", ch, ch);
    printf("%25s\n%25.4s\n\n", string, string);
    printf("%f    %.2f    %e    %E\n\n", fp, fp, fp, fp);
    printf("%i    %i    %i\n\n", hex, oct, dec);
}
/***** Output should be similar to: *****/

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

*****/
```

### Example that uses printf()

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
/* This program is compiled with LOCALETYPE(*LOCALEUCS2) and */
/* SYSIFCOPT(*IFSIO) */
/* We will assume the locale setting is the same as the CCSID of the */
/* job. We will also assume any files involved have a CCSID of */
/* 65535 (no convert). This way if printf goes to the screen or */
/* a file the output will be the same. */
int main(void)
{
    wchar_t wc = 0x0058;    /* UNICODE X */
    wchar_t ws[4];
    setlocale(LC_ALL,
        "/QSYS.LIB/EN_US.LOCALE"); /* a CCSID 37 locale */
    ws[0] = 0x0041;    /* UNICODE A */
    ws[1] = (wchar_t)0x0042;    /* UNICODE B */
    ws[2] = (wchar_t)0x0043;    /* UNICODE C */
    ws[3] = (wchar_t)0x0000;
```

```

/* The output displayed is CCSID 37 */
printf("%lc %ls\n\n",wc,ws);
printf("%lc %.2ls\n\n",wc,ws);

/* Now let's try a mixed-byte CCSID example */
/* You would need a device that can handle mixed bytes to */
/* display this correctly. */

setlocale(LC_ALL,
"/QSYS.LIB/JA_JP.LOCALE");/* a CCSID 5026 locale */

/* big A means an A that takes up 2 bytes on the screen */
/* It will look bigger then single byte A */
ws[0] = (wchar_t)0xFF21; /* UNICODE big A */
ws[1] = (wchar_t)0xFF22; /* UNICODE big B */
ws[2] = (wchar_t)0xFF23; /* UNICODE big C */
ws[3] = (wchar_t)0x0000;
wc = 0xff11; /* UNICODE big 1 */

printf("%lc %ls\n\n",wc,ws);

/* The output of this printf is not shown below and it */
/* will differ depending on the device you display it on,*/
/* but if you looked at the string in hex it would look */
/* like this: 0E42F10F404040400E42C142C242C30F */
/* 0E is shift out, 0F is shift in, and 42F1 is the */
/* big 1 in CCSID 5026 */

printf("%lc %.4ls\n\n",wc,ws);

/* The output of this printf is not shown below either. */
/* The hex would look like: */
/* 0E42F10F404040400E42C10F */
/* Since the precision is in bytes we only get 4 bytes */
/* of the string. */

printf("%lc %#.2ls\n\n",wc,ws);

/* The output of this printf is not shown below either. */
/* The hex would look like: */
/* 0E42F10F404040400E42C142C20F */
/* The # means precision is in characters regardless */
/* of size. So we get 2 characters of the string. */
}
/***** Output should be similar to: *****/

X ABC

X AB

*****/

```

### Example that uses printf()

```

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
/* This program is compile LOCALETYPE(*LOCALE) and */
/* SYSIFCOPT(*IFSIO) */
int main(void)
{
    wchar_t wc = (wchar_t)0x00C4; /* D */
    wchar_t ws[4];
    ws[0] = (wchar_t)0x00C1; /* A */
    ws[1] = (wchar_t)0x00C2; /* B */
    ws[2] = (wchar_t)0x00C3; /* C */

```

```

ws[3] = (wchar_t)0x0000;
/* The output displayed is CCSID 37 */
printf("%lc  %ls\n\n",wc,ws);

/* Now let's try a mixed-byte CCSID example */
/* You would need a device that can handle mixed bytes to */
/* display this correctly. */

setlocale(LC_ALL,
"/QSYS.lib/JA_JP.LOCALE"); /* a CCSID 5026 locale */

/* big A means an A that takes up 2 bytes on the screen */
/* It will look bigger than single byte A */

ws[0] = (wchar_t)0x42C1; /* big A */
ws[1] = (wchar_t)0x42C2; /* big B */
ws[2] = (wchar_t)0x42C3; /* big C */
ws[3] = (wchar_t)0x0000;
wc = 0x42F1; /* big 1 */

printf("%lc  %ls\n\n",wc,ws);

/* The output of this printf is not shown below and it */
/* will differ depending on the device you display it on,*/
/* but if you looked at the string in hex it would look */
/* like this: 0E42F10F404040400E42C142C242C30F */
/* 0E is shift out, 0F is shift in, and 42F1 is the */
/* big 1 in CCSID 5026 */

printf("%lc  %.4ls\n\n",wc,ws);

/* The output of this printf is not shown below either. */
/* The hex would look like: */
/* 0E42F10F404040400E42C10F */
/* Since the precision is in bytes we only get 4 bytes */
/* of the string. */

printf("%lc  %#.2ls\n\n",wc,ws);

/* The output of this printf is not shown below either. */
/* The hex would look like: */
/* 0E42F10F404040400E42C142C20F */
/* The # means precision is in characters regardless */
/* of size. So we get 2 characters of the string. */
}
/***** Output should be similar to: *****/

```

```
D    ABC
```

```
*****/
```

## Related Information

- “fprintf() — Write Formatted Data to a Stream” on page 116
- “fscanf() — Read Formatted Data” on page 132
- “scanf() — Read Data” on page 330
- “sprintf() — Print Formatted Data to Buffer” on page 352
- “sscanf() — Read Data” on page 355
- “vfprintf() — Print Argument Data to Stream” on page 425
- “vprintf() — Print Argument Data” on page 432
- “vsprintf() — Print Argument Data to Buffer” on page 436

- “wprintf() — Format Data as Wide Characters and Print” on page 503
- “<stdio.h>” on page 15

---

## putc() – putchar() — Write a Character

### Format

```
#include <stdio.h>
int putc(int c, FILE *stream);
int putchar(int c);
```

**Language Level:** ANSI

**Threadsafe:** No. #undef putc or #undef putchar allows the putc or putchar function to be called instead of the macro version of these functions. The functions are threadsafe.

### Description

The putc() function converts *c* to unsigned char and then writes *c* to the output *stream* at the current position. The putchar() is equivalent to putc(*c*, stdout).

The putc() function can be defined as a macro so the argument can be evaluated multiple times.

The putc() and putchar() functions are not supported for files opened with type=record.

### Return Value

The putc() and putchar() functions return the character written. A return value of EOF indicates an error.

The value of errno may be set to:

#### Value Meaning

##### ECONVERT

A conversion error occurred.

##### EPUTANDGET

An illegal write operation occurred after a read operation.

##### EIOERROR

A non-recoverable I/O error occurred.

##### EIORECERR

A recoverable I/O error occurred.

### Example that uses putc()

This example writes the contents of a buffer to a data stream. In this example, 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);
}

/***** Expected output: *****/

Hello world
*/

```

### Related Information

- “putc() — Write Character” on page 118
- “fwrite() — Write Items” on page 146
- “getc() – getchar() — Read a Character” on page 152
- “puts() — Write a String” on page 241
- “putwc() — Write Wide Character” on page 242
- “putwchar() — Write Wide Character to stdout” on page 244
- “<stdio.h>” on page 15

---

## putenv() — Change/Add Environment Variables

### Format

```

#include <stdlib.h>
int putenv(const char *varname);

```

**Language Level:** XPG4

**Threadsafe:** Yes

**Job CCSID Interface:** All character data sent to this function is expected to be in the CCSID of the job. All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and Locales” on page 524 for more information.

### Description

The `putenv()` function sets the value of an environment variables by altering an existing variable or creating a new one. The *varname* parameter points to a string of the form `var=x`, where `x` is the new value for the environment variable `var`.

The name cannot contain a blank or an equal (=) symbol. For example,

```
PATH NAME=/my_lib/joe_user
```

is not valid because of the blank between `PATH` and `NAME`. Similarly,

```
PATH=NAME=/my_lib/joe_user
```

is not valid because of the equal symbol between PATH and NAME. The system interprets all characters following the first equal symbol as being the value of the environment variable.

### Return Value

The `putenv()` function returns 0 if successful. If `putenv()` fails then -1 is returned and `errno` is set to indicate the error.

### Example that uses `putenv()`

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

int main(void)
{
    char *pathvar;

    if (-1 == putenv("PATH=./home/userid")) {
        printf("putenv failed \n");
        return EXIT_FAILURE;
    }
    /* getting and printing the current environment path    */

    pathvar = getenv("PATH");
    printf("The current path is: %s\n", pathvar);
    return 0;
}

/*****
The output should be:

The current path is: ./home/userid
*****/
```

### Related Information

- “`getenv()` — Search for Environment Variables” on page 154
- “`<stdlib.h>`” on page 17

---

## puts() — Write a String

### Format

```
#include <stdio.h>
int puts(const char *string);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `puts()` function writes the given *string* to the standard output stream `stdout`; it also appends a new-line character to the output. The ending null character is not written.

### Return Value

The `puts()` function returns EOF if an error occurs. A nonnegative return value indicates that no error has occurred.

The value of `errno` may be set to:

### Value Meaning

## ECONVERT

A conversion error occurred.

## EPUTANDGET

An illegal write operation occurred after a read operation.

## EIOERROR

A non-recoverable I/O error occurred.

## EIORECERR

A recoverable I/O error occurred.

### Example that uses puts()

This example writes Hello World to stdout.

```
#include <stdio.h>

int main(void)
{
    if ( puts("Hello World") == EOF )
        printf( "Error in puts\n" );
}

/***** Expected output: *****/

Hello World
*/
```

### Related Information

- “fputs() — Write String” on page 121
- “fputws() — Write Wide-Character String” on page 124
- “gets() — Read a Line” on page 156
- “putc() – putchar() — Write a Character” on page 239
- “putwc() — Write Wide Character”
- “<stdio.h>” on page 15

---

## putwc() — Write Wide Character

### Format

```
#include <stdio.h>
#include <wchar.h>
wint_t putwc(wint_t wc, FILE *stream);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the  
| current locale. This behavior might also be affected by the LC\_UNI\_CTYPE category of the current locale  
| if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation  
| command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation  
| command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified  
| on the compilation command.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

## Description

The `putc()` function writes the wide character *wc* to the stream at the current position. It also advances the file position indicator for the stream appropriately. The `putc()` function is equivalent to the `fputc()` function except that some platforms implement `putc()` as a macro. Therefore, for portability, the stream argument to `putc()` should not be an expression with side effects.

- | Using a non-wide-character function with the `putc()` function on the same stream results in undefined behavior. After calling the `putc()` function, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless EOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling the `putc()` function.

## Return Value

The `putc()` function returns the wide character written. If a write error occurs, it sets the error indicator for the stream and returns WEOF. If an encoding error occurs when a wide character is converted to a multibyte character, the `putc()` function sets `errno` to EILSEQ and returns WEOF.

For information about `errno` values for `putc()`, see “`fputc()` — Write Character” on page 118.

## Example that uses `putc()`

The following example uses the `putc()` function to convert the wide characters in *wcs* to multibyte characters and write them to the file **putc.out**.

```
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>
#include <errno.h>

int main(void)
{
    FILE *stream;
    wchar_t *wcs = L"A character string.";
    int i;

    if (NULL == (stream = fopen("putc.out", "w"))) {
        printf("Unable to open: \"putc.out\".\n");
        exit(1);
    }

    for (i = 0; wcs[i] != L'\0'; i++) {
        errno = 0;
        if (WEOF == putwc(wcs[i], stream)) {
            printf("Unable to putwc() the wide character.\n"
                "wcs[%d] = 0x%lx\n", i, wcs[i]);
            if (EILSEQ == errno)
                printf("An invalid wide character was encountered.\n");
            exit(1);
        }
    }
    fclose(stream);
    return 0;

    /*****
    The output file putwc.out should contain :

    A character string.
    *****/
}
```

## Related Information

- “fputc() — Write Character” on page 118
- “fputwc() — Write Wide Character” on page 122
- “fputws() — Write Wide-Character String” on page 124
- “getwc() — Read Wide Character from Stream” on page 157
- “putc() – putchar() — Write a Character” on page 239
- “putwchar() — Write Wide Character to stdout”
- “<stdio.h>” on page 15
- “<wchar.h>” on page 18

---

## putwchar() — Write Wide Character to stdout

### Format

```
#include <wchar.h>
wint_t putwchar(wint_t wc);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. This behavior might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified on the compilation command.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The putwchar() function converts the wide character *wc* to a multibyte character and writes it to stdout. A call to the putwchar() function is equivalent to putwc(wc, stdout).

Using a non-wide-character function with the putwchar() function on the same stream results in undefined behavior. After calling the putwchar() function, flush the buffer or reposition the stream pointer before calling a write function for the stream, unless EOF has been reached. After a write operation on the stream, flush the buffer or reposition the stream pointer before calling the putwchar() function.

### Return Value

The putwchar() function returns the wide character written. If a write error occurs, the putwchar() function sets the error indicator for the stream and returns WEOF. If an encoding error occurs when a wide character is converted to a multibyte character, the putwchar() function sets errno to EILSEQ and returns WEOF.

For information about errno values for putwc(), see “fputc() — Write Character” on page 118.

### Example that uses putwchar()

This example uses the putwchar() function to write the string in *wcs*.

```

#include <stdio.h>
#include <wchar.h>
#include <errno.h>
#include <stdlib.h>

int main(void)
{
    wchar_t *wcs = L"A character string.";
    int i;

    for (i = 0; wcs[i] != L'\0'; i++) {
        errno = 0;
        if (WEOF == putwchar(wcs[i])) {
            printf("Unable to putwchar() the wide character.\n");
            printf("wcs[%d] = 0x%lx\n", i, wcs[i]);
            if (EILSEQ == errno)
                printf("An invalid wide character was encountered.\n");
            exit(EXIT_FAILURE);
        }
    }
    return 0;

    /*****
    The output should be similar to :

    A character string.
    *****/
}

```

## Related Information

- “fputc() — Write Character” on page 118
- “fputwc() — Write Wide Character” on page 122
- “fputws() — Write Wide-Character String” on page 124
- “getwchar() — Get Wide Character from stdin” on page 159
- “putc() – putchar() — Write a Character” on page 239
- “putwc() — Write Wide Character” on page 242
- “<wchar.h>” on page 18

---

## qsort() — Sort Array

### Format

```

#include <stdlib.h>
void qsort(void *base, size_t num, size_t width,
           int(*compare)(const void *key, const void *element));

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `qsort()` function sorts an array of *num* elements, each of *width* bytes in size. The *base* pointer is a pointer to the array to be sorted. The `qsort()` function overwrites this array with the sorted elements.

The *compare* argument is a pointer to a function you must supply that takes a pointer to the *key* argument and to an array *element*, in that order. The `qsort()` function calls this function one or more times during the search. The function must compare the *key* and the *element* and return one of the following values:

Value	Meaning
-------	---------

Less than 0	<i>key less than element</i>
0	<i>key equal to element</i>
Greater than 0	<i>key greater than element</i>

### Value Meaning

#### Less than 0

*key less than element*

#### 0

*key equal to element*

#### Greater than 0

*key greater than element*

The sorted array elements are stored in ascending order, as defined by your *compare* function. You can sort in reverse order by reversing the sense of “greater than” and “less than” in *compare*. The order of the elements is unspecified when two elements compare equally.

### Return Value

There is no return value.

### Example that uses `qsort()`

This example sorts the arguments (`argv`) in ascending lexical sequence, 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));
}

/***** If the program is passed the arguments: *****/
/***** 'Does' 'this' 'really' 'sort' 'the' 'arguments' 'correctly?'*****/
/***** then the expected output is: *****/

arguments
correctly?
really
sort
the
this
Does
*/

```

### Related Information

- “bsearch() — Search Arrays” on page 51
- “<stdlib.h>” on page 17

---

## QXXCHGDA() — Change Data Area

### Format

```
#include <xxdtaa.h>
```

```
void QXXCHGDA(_DTAA_NAME_T dtaname, short int offset, short int len,
              char *dtaptr);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

- | **Job CCSID Interface:** All character data sent to this function is expected to be in the CCSID of the job.
- | All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and
- | Locales” on page 524 for more information.

### Description

The QXXCHGDA() function allows you to change the data area specified by *dtaname*, starting at position *offset*, with the data in the user buffer pointed to by *dataptr* of length *len*. The structure *dtaname* contains the names of the data area and the library that contains the data area. The values that can be specified for the data area name are:

**\*LDA** Specifies that the contents of the local data area are to be changed. The library name *dtaa\_lib* must be blank.

**\*GDA** Specifies that the contents of the group data area are to be changed. The library name *dtaa\_lib* must be blank.

#### **data-area-name**

Specifies that the contents of the data area created using the Create Data Area (CRTDTAARA) CL command are to be changed. The library name *dtaa\_lib* must be either \*LIBL, \*CURLIB, or the name of the library where the data area (*data-area-name*) is located. The data area is locked while it is being changed.

QXXCHGDA can only be used to change character data.

#### **Example that uses QXXCHGDA()**

```
#include <stdio.h>
#include <xxdtaa.h>

#define START 1
#define LENGTH 8

int main(void)
{
    char newdata[LENGTH] = "new data";

    /* The local data area will be changed */
    _DTAA_NAME_T dtaname = {"*LDA", " "};

    /* Use function to change the local data area. */
    QXXCHGDA(dtaname, START, LENGTH, newdata);
    /* The first 8 characters in the local data area */
    /* are: new data */
}
```

#### **Related Information**

- “QXXRTVDA() — Retrieve Data Area” on page 252

---

## **QXXDTOP() — Convert Double to Packed Decimal**

#### **Format**

```
#include <xxcvt.h>
void QXXDTOP(unsigned char *pptr, int digits, int fraction,
             double value);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

#### **Description**

The QXXDTOP function converts the *double* value specified in *value* to a packed decimal number with *digits* total digits, and *fraction* fractional digits. The result is stored in the array pointed to by *pptr*.

### Example that uses QXXDTOP()

```
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char pptr[10];
    int digits = 8, fraction = 6;
    double value = 3.141593;

    QXXDTOP(pptr, digits, fraction, value);
}
```

### Related Information

- “QXXDIOZ() — Convert Double to Zoned Decimal”
- “QXXITOP() — Convert Integer to Packed Decimal” on page 250
- “QXXITOOZ() — Convert Integer to Zoned Decimal” on page 250
- “QXXPTOD() — Convert Packed Decimal to Double” on page 251
- “QXXPTOI() — Convert Packed Decimal to Integer” on page 252
- “QXXZTOD() — Convert Zoned Decimal to Double” on page 254
- “QXXZTOI() — Convert Zoned Decimal to Integer” on page 255

---

## QXXDIOZ() — Convert Double to Zoned Decimal

### Format

```
#include <xxcvt.h>
void QXXDIOZ(unsigned char *zptr, int digits, int fraction,
             double value);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

### Description

The QXXDIOZ function converts the *double* value specified in *value* to a zoned decimal number with *digits* total digits, and *fraction* fractional digits. The result is stored in the array pointed to by *zptr*.

### Example that uses QXXDIOZ()

```
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char zptr[10];
    int digits = 8, fraction = 6;
    double value = 3.141593;

    QXXDIOZ(zptr, digits, fraction, value);
} /* Zoned value is : 03141593 */
```

### Related Information

- “QXXDTOP() — Convert Double to Packed Decimal” on page 248
- “QXXITOP() — Convert Integer to Packed Decimal” on page 250
- “QXXITOOZ() — Convert Integer to Zoned Decimal” on page 250

- “QXXPTOD() — Convert Packed Decimal to Double” on page 251
- “QXXPTOI() — Convert Packed Decimal to Integer” on page 252
- “QXXZTOD() — Convert Zoned Decimal to Double” on page 254
- “QXXZTOI() — Convert Zoned Decimal to Integer” on page 255

---

## QXXITOP() — Convert Integer to Packed Decimal

### Format

```
#include <xxcvt.h>
void QXXITOP(unsigned char *pptr, int digits, int fraction,
             int value);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

### Description

The QXXITOP function converts the integer specified in *value* to a packed decimal number with *digits* total digits, and *fraction* fractional digits. The result is stored in the array pointed to by *pptr*.

### Example that uses QXXITOP()

```
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char pptr[10];
    int digits = 3, fraction = 0;
    int value = 116;

    QXXITOP(pptr, digits, fraction, value);
}
```

### Related Information

- “QXXDTP() — Convert Double to Packed Decimal” on page 248
- “QXXDTPZ() — Convert Double to Zoned Decimal” on page 249
- “QXXITOPZ() — Convert Integer to Zoned Decimal”
- “QXXPTOD() — Convert Packed Decimal to Double” on page 251
- “QXXPTOI() — Convert Packed Decimal to Integer” on page 252
- “QXXZTOD() — Convert Zoned Decimal to Double” on page 254
- “QXXZTOI() — Convert Zoned Decimal to Integer” on page 255

---

## QXXITOPZ() — Convert Integer to Zoned Decimal

### Format

```
#include <xxcvt.h>
void QXXITOPZ(unsigned char *zptr, int digits, int fraction, int value);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

### Description

The QXXIT0Z function converts the integer specified in *value* to a zoned decimal number with *digits* total digits, and *fraction* fractional digits. The result is stored in the array pointed to by *zptr*.

#### Example that uses QXXIT0Z()

```
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char zptr[10];
    int digits = 9, fraction = 0;
    int value = 111115;

    QXXIT0Z(zptr, digits, fraction, value);
    /* Zoned value is : 000111115 */
}
```

#### Related Information

- “QXXDTP0() — Convert Double to Packed Decimal” on page 248
- “QXXDTP0Z() — Convert Double to Zoned Decimal” on page 249
- “QXXIT0P() — Convert Integer to Packed Decimal” on page 250
- “QXXPT0D() — Convert Packed Decimal to Double”
- “QXXPT0I() — Convert Packed Decimal to Integer” on page 252
- “QXXZTP0D() — Convert Zoned Decimal to Double” on page 254
- “QXXZTP0I() — Convert Zoned Decimal to Integer” on page 255

---

## QXXPT0D() — Convert Packed Decimal to Double

#### Format

```
#include <xxcvt.h>
double QXXPT0D(unsigned char *pptr, int digits, int fraction);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

#### Description

The QXXPT0D function converts a packed decimal number to a *double*.

#### Example that uses QXXPT0D()

```
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char pptr[10];
    int digits = 8, fraction = 6;
    double value = 6.123456, result;
    /* First convert an integer to a packed decimal,*/
    QXXDTP0(pptr, digits, fraction, value);
    /* then convert it back to a double. */
    result = QXXPT0D(pptr, digits, fraction);
    /* result = 6.123456 */
}
```

### Related Information

- “QXXDTOP() — Convert Double to Packed Decimal” on page 248
- “QXXDTOZ() — Convert Double to Zoned Decimal” on page 249
- “QXXITOP() — Convert Integer to Packed Decimal” on page 250
- “QXXITOEZ() — Convert Integer to Zoned Decimal” on page 250
- “QXXPTOI() — Convert Packed Decimal to Integer”
- “QXXZTOD() — Convert Zoned Decimal to Double” on page 254
- “QXXZTOI() — Convert Zoned Decimal to Integer” on page 255

---

## QXXPTOI() — Convert Packed Decimal to Integer

### Format

```
#include <xxcvt.h>
int QXXPTOI(unsigned char *pptr, int digits, int fraction);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

### Description

The QXXPTOI function converts a packed decimal number to an *integer*.

### Example that uses QXXPTOI()

```
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char pptr[10];
    int digits = 3, fraction = 0, value = 104, result;
    /* First convert an integer to a packed decimal,*/
    QXXITOP(pptr, digits, fraction, value);
    /* then convert it back to an integer.          */
    result = QXXPTOI(pptr, digits, fraction);
    /* result = 104                                */
}
```

### Related Information

- “QXXDTOP() — Convert Double to Packed Decimal” on page 248
- “QXXDTOZ() — Convert Double to Zoned Decimal” on page 249
- “QXXITOP() — Convert Integer to Packed Decimal” on page 250
- “QXXITOEZ() — Convert Integer to Zoned Decimal” on page 250
- “QXXPTOD() — Convert Packed Decimal to Double” on page 251
- “QXXZTOD() — Convert Zoned Decimal to Double” on page 254
- “QXXZTOI() — Convert Zoned Decimal to Integer” on page 255

---

## QXXRTVDA() — Retrieve Data Area

### Format

```
#include <xxdtaa.h>

void QXXRTVDA(_DTAA_NAME_T dtaname, short int offset,
              short int len, char *dtptr);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

- | **Job CCSID Interface:** All character data sent to this function is expected to be in the CCSID of the job.
- | All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and Locales” on page 524 for more information.

### Description

The following typedef definition is included in the <xxdtaa.h> header file. The character arrays are not null-ended strings so they must be blank filled.

```
typedef struct _DTAA_NAME_T {  
    char dtaa_name[10]; /* name of data area */  
    char dtaa_lib[10]; /* library that contains data area */  
}_DTAA_NAME_T;
```

The QXXRTVDA() function retrieves a copy of the data area specified by *dtaname* starting at position *offset* with a length of *len*. The structure *dtaname* contains the names of the data area and the library that contains the data area. The values that can be specified for the data area name are:

- \***LDA** The contents of the local data area are to be retrieved. The library name *dtaa\_lib* must be blank.
- \***GDA** The contents of the group data area are to be retrieved. The library name *dtaa\_lib* must be blank.
- \***PDA** Specifies that the contents of the program initialization parameters (PIP) data area are to be retrieved. The PIP data area is created for each pre-started job and is a character area up to 2000 characters in length. You cannot retrieve the PIP data area until you have acquired the requester. The library name *dtaa\_lib* must be blank.

#### **data-area-name**

Specifies that the contents of the data area created using the Create Data Area (CRTDTAARA) CL command are to be retrieved. The library name *dtaa\_lib* must be either \*LIBL, \*CURLIB, or the name of the library where the data area (*data-area-name*) is located. The data area is locked while the data is being retrieved.

The parameter *dtaptr* is a pointer to the storage that receives the retrieved copy of the data area. Only character data can be retrieved using QXXRTVDA.

**Example that uses QXXRTVDA()**

```

#include <stdio.h>
#include <xxdtaa.h>

#define DATA_AREA_LENGTH 30
#define START 6
#define LENGTH 7

int main(void)
{
    char uda_area[DATA_AREA_LENGTH];

    /* Retrieve data from user-defined data area currently in MYLIB */
    _DTAA_NAME_T dtaname = {"USRDDA", "MYLIB"};

    /* Use the function to retrieve some data into uda_area. */
    QXXRTVDA(dtaname, START, LENGTH, uda_area);

    /* Print the contents of the retrieved subset. */
    printf("uda_area contains %7.7s\n", uda_area);
}

```

### Related Information

- “QXXCHGDA() — Change Data Area” on page 247

---

## QXXZTOD() — Convert Zoned Decimal to Double

### Format

```

#include <xxcvt.h>
double QXXZTOD(unsigned char *zptr, int digits, int fraction);

```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

### Description

The QXXZTOD function converts to a *double*, the zoned decimal number (with *digits* total digits, and *fraction* fractional digits) pointed to by *zptr*. The resulting *double* value is returned.

### Example that uses QXXZTOD()

```

#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char zptr[] = "06123456";
    int digits = 8, fraction = 6;
    double result;

    result = QXXZTOD(zptr, digits, fraction);
    /* result = 6.123456 */
}

```

### Related Information

- “QXXDTOP() — Convert Double to Packed Decimal” on page 248
- “QXXDZTOZ() — Convert Double to Zoned Decimal” on page 249
- “QXXITOP() — Convert Integer to Packed Decimal” on page 250
- “QXXITDZ() — Convert Integer to Zoned Decimal” on page 250

- “QXXPTOD() — Convert Packed Decimal to Double” on page 251
- “QXXPTOI() — Convert Packed Decimal to Integer” on page 252
- “QXXZTOI() — Convert Zoned Decimal to Integer”

---

## QXXZTOI() — Convert Zoned Decimal to Integer

### Format

```
#include <xxcvt.h>
int QXXZTOI(unsigned char *zptr, int digits, int fraction);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

### Description

The QXXZTOI function converts to an *integer*, the zoned decimal number (with *digits* total digits, and *fraction* fractional digits) pointed to by *zptr*. The resulting integer is returned.

### Example that uses QXXZTOI()

```
#include <xxcvt.h>
#include <stdio.h>

int main(void)
{
    unsigned char zptr[] = "000111115";
    int digits = 9, fraction = 0, result;

    result = QXXZTOI(zptr, digits, fraction);
    /* result = 111115 */
}
```

### Related Information

- “QXXDTOP() — Convert Double to Packed Decimal” on page 248
- “QXXDZOZ() — Convert Double to Zoned Decimal” on page 249
- “QXXITOP() — Convert Integer to Packed Decimal” on page 250
- “QXXITZOZ() — Convert Integer to Zoned Decimal” on page 250
- “QXXPTOD() — Convert Packed Decimal to Double” on page 251
- “QXXPTOI() — Convert Packed Decimal to Integer” on page 252
- “QXXZTOD() — Convert Zoned Decimal to Double” on page 254

---

## raise() — Send Signal

### Format

```
#include <signal.h>
int raise(int sig);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `raise()` functions sends the signal *sig* to the running program. If compiled with `SYSIFCOPT(*ASYNCSIGNAL)` on the compilation command, this function uses asynchronous signals. The asynchronous version of this function throws a signal to the process or thread.

### Return Value

The `raise()` functions returns 0 if successful, nonzero if unsuccessful.

### Example that uses `raise()`

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>

void sig_hand(int); /* declaration of sig_hand() as a function */

int main(void)
{
    signal(SIGUSR1, sig_hand); /* set up handler for SIGUSR1 */

    raise(SIGUSR1); /* signal SIGUSR1 is raised */
                  /* sig_hand() is called */
}

void sig_hand(int sig)
{
    static int count = 0; /* initialized only once */

    count++;
    if (count == 10) /* ignore the first 9 occurrences of this signal */
        exit(10);
    else
        signal(SIGUSR1, sig_hand); /* set up the handler again */
}
/* This is a program fragment and not a complete program */
```

### Related Information

- “`signal()` — Handle Interrupt Signals” on page 346
- “Signal Handling Action Definitions” on page 511
- “`<signal.h>`” on page 13
- Signal APIs in the APIs topic in the i5/OS Information Center.
- POSIX thread APIs in the APIs topic in the i5/OS Information Center.

---

## `rand()`, `rand_r()` — Generate Random Number

### Format

```
#include <stdlib.h>
int rand(void);
int rand_r(unsigned int *seed);
```

**Language Level:** ANSI

**Threadsafe:** No. `rand()` is not threadsafe, but `rand_r()` is.

### Description

The `rand()` function generates a pseudo-random integer in the range 0 to `RAND_MAX` (macro defined in `<stdlib.h>`). Use the `srand()` function before calling `rand()` to set a starting point for the random number generator. If you do not call the `srand()` function first, the default seed is 1.

**Note:** The `rand_r()` function is the restartable version of `rand()`.

### Return Value

The `rand()` function returns a pseudo-random number.

### Example that uses `rand()`

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 should be similar to: *****/

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

- “`srand()` — Set Seed for `rand()` Function” on page 354
- “`<stdlib.h>`” on page 17

---

## **`_Racquire()` — Acquire a Program Device**

### Format

```
#include <recio.h>
int _Racquire(_RFILE *fp, char *dev);
```

**Language Level:** ILE C Extension

**Threadsafe:** No.

- | **Job CCSID Interface:** All character data sent to this function is expected to be in the CCSID of the job.
- | All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and
- | Locales” on page 524 for more information.

### Description

The `_Racquire()` function acquires the program device specified by the `dev` parameter and associates it with the file specified by `fp`. The `dev` parameter is a null-terminated C string. The program device name must be specified in uppercase. The program device must be defined to the file.

This function is valid for display and ICF files.

### Return Value

The `_Racquire()` function returns 1 if it is successful or zero if it is unsuccessful. The value of `errno` may be set to `EIOERROR` (a non-recoverable I/O error occurred) or `EIORECERR` (a recoverable I/O error occurred).

See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

### Example that uses `_Racquire()`

```
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

int main(void)
{
    _RFILE    *fp;
    _RIOFB_T  *rfb;

    /* Open the device file. */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit ( 1 );
    }

    _Racquire ( fp,"DEVICE1" );    /* Acquire another program device. */
                                  /* Replace with actual device name.*/

    _Rformat ( fp,"FORMAT1" );    /* Set the record format for the */
                                  /* display file. */

    rfb = _Rwrite ( fp, "", 0 );  /* Set up the display. */

    /* Do some processing... */

    _Rclose ( fp );
}
```

### Related Information

- “`_Rrelease()` — Release a Program Device” on page 314

---

## `_Rclose()` — Close a File

### Format

```
#include <recio.h>

int _Rclose(_RFILE *fp);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

## Description

The `_Rclose()` function closes the file specified by *fp*. Before this file is closed, all buffers associated with it are flushed and all buffers reserved for it are released. The file is closed even if an exception occurs. The `_Rclose()` function applies to all types of files.

**Note:** Closing a file more than once in a multi-threaded environment will cause undefined behavior.

## Return Value

The `_Rclose()` function returns zero if the file is closed successfully, or EOF if the close operation failed or the file was already closed. The file is closed even if an exception occurs, and zero is returned.

The value of `errno` may be set to:

Value	Meaning
ENOTOPEN	The file is not open.
EIOERROR	A non-recoverable I/O error occurred.
EIORECERR	A recoverable I/O error occurred.

See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

## Example that uses `_Rclose()`

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

int main(void)
{
    _RFILE    *fp;

    /* Open the file for processing in arrival sequence.          */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+", arrseq=Y" )) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }
    else
        /* Do some processing */;

    _Rclose ( fp );
}
```

## Related Information

- “`_Ropen()` — Open a Record File for I/O Operations” on page 289

---

## `_Rcommit()` — Commit Current Record

### Format

```
#include <recio.h>
int _Rcommit(char *cmtid);
```

**Language Level:** ILE C Extension

**Threadsafe:** No.

- | **Job CCSID Interface:** All character data sent to this function is expected to be in the CCSID of the job.
- | All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and
- | Locales” on page 524 for more information.

### Description

The `_Rcommit()` function completes the current transaction for the job that calls it and establishes a new commitment boundary. All changes made since the last commitment boundary are made permanent. Any file or resource that is open under commitment control in the job is affected.

The `cmtid` parameter is a null-ended C string used to identify the group of changes associated with a commitment boundary. It cannot be longer than 4000 bytes.

The `_Rcommit()` function applies to database and DDM files.

### Return Value

The `_Rcommit()` function returns 1 if the operation is successful or zero if the operation is unsuccessful. The value of `errno` may be set to `EIOERROR` (a non-recoverable I/O error occurred) or `EIORECERR` (a recoverable I/O error occurred).

See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

**Example that uses `_Rcommit()`**

```

#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char    buf[40];
    int     rc = 1;
    _RFILE  *purf;
    _RFILE  *dailyf;

    /* Open purchase display file and daily transaction file      */
    if ( ( purf = _Ropen ( "MYLIB/T1677RD3", "ar+,indicators=y" ) ) == NULL )
    {
        printf ( "Display file did not open.\n" );
        exit ( 1 );
    }

    if ( ( dailyf = _Ropen ( "MYLIB/T1677RDA", "wr,commit=y" ) ) == NULL )
    {
        printf ( "Daily transaction file did not open.\n" );
        exit ( 2 );
    }

    /* Select purchase record format */
    _Rformat ( purf, "PURCHASE" );

    /* Invite user to enter a purchase transaction.              */
    /* The _Rwrite function writes the purchase display.         */
    _Rwrite ( purf, "", 0 );
    _Rreadn ( purf, buf, sizeof(buf), __DFT );

    /* Update daily transaction file                              */
    rc = ( ( _Rwrite ( dailyf, buf, sizeof(buf) ) )->num_bytes );

    /* If the databases were updated, then commit the transaction. */
    /* Otherwise, rollback the transaction and indicate to the    */
    /* user that an error has occurred and end the application.   */
    if ( rc )
    {
        _Rcommit ( "Transaction complete" );
    }
    else
    {
        _Rrollbck ( );
        _Rformat ( purf, "ERROR" );
    }

    _Rclose ( purf );
    _Rclose ( dailyf );
}

```

### Related Information

- “\_Rrollbck() — Roll Back Commitment Control Changes” on page 317

---

## **\_Rdelete() — Delete a Record**

### Format

```

#include <recio.h>
_RIOFB_T *_Rdelete(_RFILE *fp);

```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

### **Description**

The `_Rdelete()` function deletes the record that is currently locked for update in the file specified by *fp*. After the delete operation, the record is not locked. The file must be open for update.

A record is locked for update by reading or locating to it unless `__NO_LOCK` is specified on the read or locate option. If the `__NO_POSITION` option is specified on the locate operation that locked the record, the record deleted may not be the record that the file is currently positioned to.

This function is valid for database and DDM files.

### **Return Value**

The `_Rdelete()` function returns a pointer to the `_RIOFB_T` structure associated with *fp*. If the operation is successful, the `num_bytes` field contains 1. If the operation is unsuccessful, the `num_bytes` field contains zero.

The value of `errno` may be set to:

#### **Value    Meaning**

##### **ENOTDLT**

The file is not open for delete operations.

##### **EIOERROR**

A non-recoverable I/O error occurred.

##### **EIORECERR**

A recoverable I/O error occurred.

See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

**Example that uses `_Rdelete()`**

```

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence.          */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+", arrseq=Y" )) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened.        */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s\nFile:   %10.10s\n",
            opfb->library_name,
            opfb->file_name);

    /* Get the first record.                                     */
    _Rreadf ( fp, NULL, 20, _DFT );
    printf ( "First record: %10.10s\n", *(fp->in_buf) );

    /* Delete the first record.                                 */
    _Rdelete ( fp );

    _Rclose ( fp );
}

```

### Related Information

- “\_Rrslck() — Release a Record Lock” on page 316

---

## \_Rdevatr() — Get Device Attributes

### Format

```

#include <recio.h>
#include <xxfdbk.h>
_XXDEV_ATR_T *_Rdevatr(_RFILE *fp, char *dev);

```

**Language Level:** ILE C Extension

**Threadsafe:** No.

- | **Job CCSID Interface:** All character data sent to this function is expected to be in the CCSID of the job.
- | All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and Locales” on page 524 for more information.

### Description

The `_Rdevatr()` function returns a pointer to a copy of the device attributes feedback area for the file pointed to by `fp`, and the device specified by `dev`.

The `dev` parameter is a null-ended C string. The device name must be specified in uppercase.

The `_Rdevatr()` function is valid for display and ICF files.

### Return Value

The `_Rdevatr()` function returns `NULL` if an error occurs.

See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

### Example that uses `_Rdevatr()`

```
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char ** argv)
{
    _RFILE *fp; /* File pointer */
    _RIOFB_T *rfb; /*Pointer to the file's feedback structure */
    _XXIOFB_T *iofb; /* Pointer to the file's feedback area */
    _XXDEV_ATTR_T *dv_atr; /* Pointer to a copy of the file's device
        /* attributes feedback area */

    /* Open the device file. */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit ( 1 );
    }

    dv_atr = _Rdevatr (fp, argv[1]);
    if (dv_atr == NULL)
        printf("Error occurred getting device attributes for %s.\n",
            argv[1]);

    _Rclose ( fp );
}
```

### Related Information

- “`_Racquire()` — Acquire a Program Device” on page 257
- “`_Rrelease()` — Release a Program Device” on page 314

---

## realloc() — Change Reserved Storage Block Size

### Format

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `realloc()` function changes the size of a previously reserved storage block. The *ptr* argument points to the beginning of the block. The *size* argument gives the new size of the block, in bytes. The contents of the block are unchanged up to the shorter of the new and old sizes.

If the *ptr* is `NULL`, `realloc()` reserves a block of storage of *size* bytes. It does not necessarily give all bits of each element an initial value of 0.

If *size* is 0 and the *ptr* is not `NULL`, `realloc()` frees the storage allocated to *ptr* and returns `NULL`.

**Notes:**

1. All heap storage is associated with the activation group of the calling routine. As such, storage should be allocated and deallocated within the same activation group. You cannot allocate heap storage within one activation group and deallocate that storage from a different activation group. For more information about activation groups, see the *ILE Concepts* manual.
2. If the `_C_Quickpool_Init()` function has been called in the current activation group then storage is retrieved using Quick Pool memory management. See `_C_Quickpool_Init()` for more information.

**Return Value**

The `realloc()` function returns a pointer to the reallocated storage block. The storage location of the block may be moved by the `realloc()` function. Thus, the *ptr* argument to the `realloc()` function is not necessarily the same as the return value.

If *size* is 0, the `realloc()` function returns NULL. If there is not enough storage to expand the block to the given size, the original block is unchanged and the `realloc()` function returns NULL.

The storage to which the return value points is aligned for storage of any type of object.

To use **Teraspace** storage instead of heap storage without changing the C source code, specify the `TERASPACE(*YES *TSIFC)` parameter on the `CRTCMOD` compiler command. This maps the `realloc()` library function to `_C_TS_realloc()`, its Teraspace storage counterpart. The maximum amount of Teraspace storage that can be allocated by each call to `_C_TS_realloc()` is 2GB - 240, or 214743408 bytes. For additional information about Teraspace, see the *ILE Concepts* manual.

**Example that uses `realloc()`**

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 */
        perror( "Out of storage" );
        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 */
        perror( "Out of storage" );
        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] );
}

/**** If the initial value entered is 2 and the second value entered
      is 4, then the expected output is:
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

- | • “calloc() — Reserve and Initialize Storage” on page 55
- | • “\_C\_Quickpool\_Debug() — Modify Quick Pool Memory Management Characteristics” on page 66
- | • “\_C\_Quickpool\_Init() — Initialize Quick Pool Memory Management” on page 68

- | • “\_C\_Quickpool\_Report() — Generate Quick Pool Memory Management Report” on page 70
- | • “free() — Release Storage Blocks” on page 128
- | • “malloc() — Reserve Storage Block” on page 195
- | • “<stdlib.h>” on page 17

---

## regcomp() — Compile Regular Expression

### Format

```
#include <regex.h>
int regcomp(regex_t *preg, const char *pattern, int cflags);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

- | **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_COLLATE categories of the current locale. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `regcomp()` function compiles the source regular expression pointed to by *pattern* into an executable version and stores it in the location pointed to by *preg*. You can then use the `regexexec()` function to compare the regular expression to other strings.

The *cflags* flag defines the attributes of the compilation process:

cflag	Description String
REG_ALT_NL	<ul style="list-style-type: none"> <li>• When LOCALETYPE(*LOCALE) is specified, the newline character of the integrated file system will be matched by regular expressions.</li> <li>• When LOCALETYPE(*LOCALEUTF) is specified, the database newline character will be matched.</li> </ul> <p>If the REG_ALT_NL flag is not set, the default for LOCALETYPE(*LOCALE) is to match the database newline, and the default for LOCALETYPE(*LOCALEUTF) is to match the integrated file system newline.</p> <p><b>Note:</b> For UTF-8 and UTF-32, the newline character of the integrated file system and the database newline character are the same.</p>
REG_EXTENDED	Support extended regular expressions.
REG_NEWLINE	Treat newline character as a special end-of-line character; it then establishes the line boundaries matched by the ] and \$ patterns, and can only be matched within a string explicitly using \n. (If you omit this flag, the newline character is treated like any other character.)
REG_ICASE	Ignore case in match.

cflag	Description String
REG_NOSUB	Ignore the number of subexpressions specified in <i>pattern</i> . When you compare a string to the compiled pattern (using <code>regexexec()</code> ), the string must match the entire pattern. The <code>regexexec()</code> function then returns a value that indicates only if a match was found; it does not indicate at what point in the string the match begins, or what the matching string is.

Regular expressions are a context-independent syntax that can represent a wide variety of character sets and character set orderings, which can be interpreted differently depending on the current locale. The functions `regcomp()`, `regerror()`, `regexexec()`, and `regfree()` use regular expressions in a similar way to the UNIX<sup>®</sup> `awk`, `ed`, `grep`, and `egrep` commands.

### Return Value

If the `regcomp()` function is successful, it returns 0. Otherwise, it returns an error code that you can use in a call to the `regerror()` function, and the content of *preg* is undefined.

### Example that uses `regcomp()`

```

#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    regex_t    preg;
    char       *string = "a very simple simple simple string";
    char       *pattern = "\\(sim[a-z]le\\) \\1";
    int        rc;
    size_t     nmatch = 2;
    regmatch_t pmatch[2];

    if (0 != (rc = regcomp(&preg, pattern, 0))) {
        printf("regcomp() failed, returning nonzero (%d)\n", rc);
        exit(EXIT_FAILURE);
    }

    if (0 != (rc = regexec(&preg, string, nmatch, pmatch, 0))) {
        printf("Failed to match '%s' with '%s', returning %d.\n",
            string, pattern, rc);
    }
    else {
        printf("With the whole expression, "
            "a matched substring \"%.*s\" is found at position %d to %d.\n",
            pmatch[0].rm_eo - pmatch[0].rm_so, &string[pmatch[0].rm_so],
            pmatch[0].rm_so, pmatch[0].rm_eo - 1);
        printf("With the sub-expression, "
            "a matched substring \"%.*s\" is found at position %d to %d.\n",
            pmatch[1].rm_eo - pmatch[1].rm_so, &string[pmatch[1].rm_so],
            pmatch[1].rm_so, pmatch[1].rm_eo - 1);
    }
    regfree(&preg);
    return 0;

    /*****
    The output should be similar to :

    With the whole expression, a matched substring "simple simple" is found
    at position 7 to 19.
    With the sub-expression, a matched substring "simple" is found
    at position 7 to 12.
    *****/
}

```

## Related Information

- “regerror() — Return Error Message for Regular Expression”
- “regexec() — Execute Compiled Regular Expression” on page 271
- “regfree() — Free Memory for Regular Expression” on page 273
- “<regex.h>” on page 12

---

## regerror() — Return Error Message for Regular Expression

### Format

```

#include <regex.h>
size_t regerror(int errcode, const regex_t *preg,
    char *errbuf, size_t errbuf_size);

```

**Language Level:** XPG4

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_COLLATE categories of the current locale. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

## Description

The `regerror()` function finds the description for the error code *errcode* for the regular expression *preg*. The description for *errcode* is assigned to *errbuf*. The *errbuf\_size* value specifies the maximum message size that can be stored (the size of *errbuf*). The description strings for *errcode* are:

errcode	Description String
REG_NOMATCH	regexec() failed to find a match.
REG_BADPAT	Invalid regular expression.
REG_ECOLLATE	Invalid collating element referenced.
REG_ECTYPE	Invalid character class type referenced.
REG_EESCAPE	Last character in regular expression is a \.
REG_ESUBREG	Number in \digit invalid, or error.
REG_EBRACK	[] imbalance.
REG_EPAREN	\( \) or () imbalance.
REG_EBRACE	\{ \} imbalance.
REG_BADDBR	Expression between \{ and \} is invalid.
REG_ERANGE	Invalid endpoint in range expression.
REG_ESPACE	Out of memory.
REG_BADRPT	?, *, or + not preceded by valid regular expression.
REG_ECHAR	Invalid multibyte character.
REG_EBOL	^ anchor not at beginning of regular expression.
REG_EEOL	\$ anchor not at end of regular expression.
REG_ECOMP	Unknown error occurred during regcomp() call.
REG_EEXEC	Unknown error occurred during regexec() call.

## Return Value

The `regerror()` returns the size of the buffer needed to hold the string that describes the error condition. The value of `errno` may be set to **ECONVERT** (conversion error).

## Example that uses regerror()

This example compiles an invalid regular expression, and prints an error message using the `regerror()` function.

```

#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    regex_t preg;
    char *pattern = "a[missing.bracket";
    int rc;
    char buffer[100];

    if (0 != (rc = regcomp(&preg, pattern, REG_EXTENDED))) {
        regerror(rc, &preg, buffer, 100);
        printf("regcomp() failed with '%s'\n", buffer);
        exit(EXIT_FAILURE);
    }
    return 0;
}

/*****
    The output should be similar to:

    regcomp() failed with '[' imbalance.'
    *****/

```

### Related Information

- “regcomp() — Compile Regular Expression” on page 267
- “regex() — Execute Compiled Regular Expression”
- “regfree() — Free Memory for Regular Expression” on page 273
- “<regex.h>” on page 12

---

## regex() — Execute Compiled Regular Expression

### Format

```

#include <regex.h>
int regex(const regex_t *preg, const char *string,
          size_t nmatch, regmatch_t *pmatch, int eflags);

```

**Language Level:** XPG4

**Threadsafe:** Yes.

**Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_COLLATE categories of the current locale. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The regex() function compares the null-ended *string* against the compiled regular expression *preg* to find a match between the two.

The *nmatch* value is the number of substrings in *string* that the regex() function should try to match with subexpressions in *preg*. The array you supply for *pmatch* must have at least *nmatch* elements.

The regex() function fills in the elements of the array *pmatch* with offsets of the substrings in *string* that correspond to the parenthesized subexpressions of the original pattern given to the regcomp() function to create *preg*. The zeroth element of the array corresponds to the entire pattern. If there are

more than *nmatch* subexpressions, only the first *nmatch* - 1 are stored. If *nmatch* is 0, or if the REG\_NOSUB flag was set when *preg* was created with the `regcomp()` function, the `regexec()` function ignores the *pmatch* argument.

The eflags *flag* defines customizable behavior of the `regexec()` function:

errflag	Description String
REG_NOTBOL	Indicates that the first character of <i>string</i> is not the beginning of line.
REG_NOTEOL	Indicates that the first character of <i>string</i> is not the end of line.

When a basic or extended regular expression is matched, any given parenthesized subexpression of the original pattern could participate in the match of several different substrings of *string*. The following rules determine which substrings are reported in *pmatch*:

1. If subexpression *i* in a regular expression is not contained within another subexpression, and it participated in the match several times, then the byte offsets in *pmatch[i]* will delimit the last such match.
2. If subexpression *i* is not contained within another subexpression, and it did not participate in an otherwise successful match, the byte offsets in *pmatch[i]* will be -1. A subexpression does not participate in the match when any of following conditions are true:
  - \* or \{ \} appears immediately after the subexpression in a basic regular expression.
  - \*, ?, or { } appears immediately after the subexpression in an extended regular expression, and the subexpression did not match (matched 0 times).
  - | is used in an extended regular expression to select this subexpression or another, and the other subexpression matched.
3. If subexpression *i* is contained within another subexpression *j*, and *i* is not contained within any other subexpression that is contained within *j*, and a match of subexpression *j* is reported in *pmatch[j]*, then the match or non-match of subexpression *i* reported in *pmatch[i]* will be as described in 1. and 2. above, but within the substring reported in *pmatch[j]* rather than the whole string.
4. If subexpression *i* is contained in subexpression *j*, and the byte offsets in *pmatch[j]* are -1, then the offsets in *pmatch[i]* also will be -1.
5. If subexpression *i* matched a zero-length string, then both byte offsets in *pmatch[i]* will be the byte offset of the character or null terminator immediately following the zero-length string.

If the REG\_NOSUB flag was set when *preg* was created by the `regcomp()` function, the contents of *pmatch* are unspecified. If the REG\_NEWLINE flag was set when *preg* was created, new-line characters are allowed in string.

### Return Value

If a match is found, the `regexec()` function returns 0. If no match is found, the `regexec()` function returns REG\_NOMATCH. Otherwise, it returns a nonzero value indicating an error. A nonzero return value can be used in a call to the `regerror()` function.

### Example that uses `regexec()`

```

#include <regex.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    regex_t    preg;
    char      *string = "a very simple simple simple string";
    char      *pattern = "\\(sim[a-z]le\\) \\1";
    int       rc;
    size_t    nmatch = 2;
    regmatch_t pmatch[2];

    if (0 != (rc = regcomp(&preg, pattern, 0))) {
        printf("regcomp() failed, returning nonzero (%d)\n", rc);
        exit(EXIT_FAILURE);
    }

    if (0 != (rc = regexec(&preg, string, nmatch, pmatch, 0))) {
        printf("Failed to match '%s' with '%s', returning %d.\n",
            string, pattern, rc);
    }
    else {
        printf("With the whole expression, "
            "a matched substring \"%.*s\" is found at position %d to %d.\n",
            pmatch[0].rm_eo - pmatch[0].rm_so, &string[pmatch[0].rm_so],
            pmatch[0].rm_so, pmatch[0].rm_eo - 1);
        printf("With the sub-expression, "
            "a matched substring \"%.*s\" is found at position %d to %d.\n",
            pmatch[1].rm_eo - pmatch[1].rm_so, &string[pmatch[1].rm_so],
            pmatch[1].rm_so, pmatch[1].rm_eo - 1);
    }
    regfree(&preg);
    return 0;
}

/*****
    The output should be similar to :

    With the whole expression, a matched substring "simple simple" is found
    at position 7 to 19.
    With the sub-expression, a matched substring "simple" is found
    at position 7 to 12.
*****/

```

## Related Information

- “regcomp() — Compile Regular Expression” on page 267
- “regerror() — Return Error Message for Regular Expression” on page 269
- “regfree() — Free Memory for Regular Expression”
- “<regex.h>” on page 12

---

## regfree() — Free Memory for Regular Expression

### Format

```

#include <regex.h>
void regfree(regex_t *preg);

```

**Language Level:** XPG4

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_COLLATE categories of the current locale. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

## Description

The `regfree()` function frees any memory that was allocated by the `regcomp()` function to implement the regular expression *preg*. After the call to the `regfree()` function, the expression that is defined by *preg* is no longer a compiled regular or extended expression.

## Return Value

There is no return value.

## Example that uses `regfree()`

This example compiles an extended regular expression.

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

int main(void)
{
    regex_t preg;
    char *pattern = ".*(simple).*";
    int rc;

    if (0 != (rc = regcomp(&preg, pattern, REG_EXTENDED))) {
        printf("regcomp() failed, returning nonzero (%d)\n", rc);
        exit(EXIT_FAILURE);
    }

    regfree(&preg);
    printf("regcomp() is successful.\n");
    return 0;
}

/*****
    The output should be similar to:

    regcomp() is successful.
*****/
```

## Related Information

- “`regcomp()` — Compile Regular Expression” on page 267
- “`regerror()` — Return Error Message for Regular Expression” on page 269
- “`regexexec()` — Execute Compiled Regular Expression” on page 271
- “`<regex.h>`” on page 12

---

## `remove()` — Delete File

### Format

```
#include <stdio.h>
int remove(const char *filename);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

## Description

The `remove()` function deletes the file specified by *filename*. If the filename contains the member name, the member is removed or the file is deleted.

**Note:** You cannot remove a nonexistent file or a file that is open.

## Return Value

The `remove()` function returns 0 if it successfully deletes the file. A nonzero return value indicates an error.

The value of `errno` may be set to `ECONVERT` (conversion error).

## Example that uses `remove()`

When you call 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 )
            perror( "Could not remove file" );
}
```

## Related Information

- “`fopen()` — Open Files” on page 109
- “`rename()` — Rename File”
- “`<stdio.h>`” on page 15

---

## `rename()` — Rename File

### Format

```
#include <stdio.h>
int rename(const char *oldname, const char *newname);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `rename()` function renames the file specified by *oldname* to the name given 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 a file with the name of an existing file. You also cannot rename an open file.

The file formats that can be used to satisfy the new name depend on the format of the old name. The following table shows the valid file formats that can be used to specify the old file name and the corresponding valid file formats for the new name.

If the format for both new name and old name is lib/file(member), then the file cannot change. If the file name changes, rename will not work. For example, the following is not valid: lib/file1(member1) lib/file2(member1).

Old Name	New Name
lib/file(member)	lib/file(member), lib/file, file, file(member)
lib/file	lib/file, file
file	lib/file, file
file(member)	lib/file(member), lib/file, file, file(member)

## Return Value

The rename() function returns 0 if successful. On an error, it returns a nonzero value.

The value of errno may be set to ECONVERT (conversion error).

## Example that uses rename()

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 )
        perror ( "Could not rename file" );
}
```

## Related Information

- “fopen() — Open Files” on page 109
- “remove() — Delete File” on page 274
- “<stdio.h>” on page 15

---

## rewind() — Adjust Current File Position

### Format

```
#include <stdio.h>
void rewind(FILE *stream);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The rewind() function repositions the file pointer associated with *stream* to the beginning of the file. A call to the rewind() function is the same as:

```
(void)fseek(stream, 0L, SEEK_SET);
```

except that the rewind() function also clears the error indicator for the *stream*.

The `rewind()` function is not supported for files opened with `type=record`.

## Return Value

There is no return value.

The value of `errno` may be set to:

### Value Meaning

#### EBADF

The file pointer or descriptor is not valid.

#### ENODEV

Operation attempted on a wrong device.

#### EIOERROR

A non-recoverable I/O error occurred.

#### EIORECERR

A recoverable I/O error occurred.

## Example that uses `rewind()`

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 in the data.

```
#include <stdio.h>

FILE *stream;

int data1, data2, data3, data4;
int main(void)
{
    data1 = 1; data2 = -37;

    /* Place data in the file */
    stream = fopen("mylib/myfile", "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 should be similar to: *****/

The values read back in are: 1 and -37
*/
```

## Related Information

- “`fgetpos()` — Get File Position” on page 99
- “`fseek()` — `fseeko()` — Reposition File Position” on page 134
- “`fsetpos()` — Set File Position” on page 136
- “`ftell()` — `ftello()` — Get Current Position” on page 138
- “`<stdio.h>`” on page 15

---

## **\_Rfeed() — Force the End-of-Data**

### **Format**

```
#include <recio.h>

int _Rfeed(_RFILE *fp);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

### **Description**

The `_Rfeed()` function forces an end-of-data condition for a device or member associated with the file specified by `fp`. Any outstanding updates, deletes or writes that the system is buffering will be forced to nonvolatile storage. If a database file is open for input, any outstanding locks will be released.

The `_Rfeed()` function positions the file to `*END` unless the file is open for multi-member processing and the current member is not the last member in the file. If multi-member processing is in effect and the current member is not the last member in the file, `_Rfeed()` will open the next member of the file and position it to `*START`.

The `_Rfeed()` function is valid for all types of files.

### **Return Value**

The `_Rfeed()` function returns 1 if multi-member processing is taking place and the next member has been opened. EOF is returned if the file is positioned to `*END`. If the operation is unsuccessful, zero is returned. The value of `errno` may be set to `EIOERROR` (a non-recoverable error occurred) or `EIORECERR` (a recoverable I/O error occurred). See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

### **Example that uses `_Rfeed()`**

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

int main(void)
{
    _RFILE    *in;
    char      new_purchase[21] = "PEAR    1002022244";

    /* Open the file for processing in keyed sequence.          */
    if ( (in = _Ropen("MYLIB/T1677RD4", "rr+", arrseq=N")) == NULL )
    {
        printf("Open failed\n");
        exit(1);
    };

    /* Update the first record in the keyed sequence.          */
    _Rlocate(in, NULL, 0, __FIRST);
    _Rupdate(in, new_purchase, 20);

    /* Force the end of data.                                    */
    _Rfeed(in);
}
```

### **Related Information**

- “\_Racquire() — Acquire a Program Device” on page 257
- “\_Rfeov() — Force the End-of-File”

---

## **\_Rfeov() — Force the End-of-File**

### **Format**

```
#include <recio.h>

int _Rfeov(_RFILE *fp);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

### **Description**

The `_Rfeov()` function forces an end-of-volume condition for a tape file that is associated with the file that is specified by *fp*. The `_Rfeov()` function positions the file to the next volume of the file. If the file is open for output, the output buffers will be flushed.

The `_Rfeov()` function is valid for tape files.

### **Return Value**

The `_Rfeov()` function returns 1 if the file has moved from one volume to the next. It will return EOF if it is called while processing the last volume of the file. It will return zero if the operation is unsuccessful. The value of `errno` may be set to `EIOERROR` (a non-recoverable error occurred) or `EIORECERR` (a recoverable I/O error occurred). See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

**Example that uses `_Rfeov()`**

```

#include <stdio.h>
#include <recio.h>
#include <stdlib.h>

int main(void)
{
    _RFILE *tape;
    _RFILE *fp;
    char buf[92];
    int i, feov2;

    /* Open source physical file containing C source.          */
    if (( fp = _Ropen ( "QCSRC(T1677SRC)", "rr blkrcd=y" )) == NULL )
    {
        printf ( "could not open C source file\n" );
        exit ( 1 );
    }

    /* Open tape file to receive C source statements          */
    if (( tape = _Ropen ( "T1677TPF", "wr lrecl=92 blkrcd=y" )) == NULL )
    {
        printf ( "could not open tape file\n" );
        exit ( 2 );
    }

    /* Read the C source statements, find their sizes          */
    /* and add them to the tape file.                          */
    while (( _Rreadn ( fp, buf, sizeof(buf), __DFT )) -> num_bytes != EOF
)
    {
        for ( i = sizeof(buf) - 1 ; buf[i] == ' ' && i > 12;      --i );
        i = (i == 12) ? 80 : (1-12);
        memmove( buf, buf+12, i );
        _Rwrite ( tape, buf, i );
    }
    feov2 = _Rfeov (fp);

    _Rclose ( fp );
    _Rclose ( tape );
}

```

### Related Information

- “\_Racquire() — Acquire a Program Device” on page 257
- “\_Rfeod() — Force the End-of-Data” on page 278

---

## **\_Rformat() — Set the Record Format Name**

### Format

```
#include <recio.h>
```

```
void _Rformat(_RFILE *fp, char *fmt);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

- | **Job CCSID Interface:** All character data sent to this function is expected to be in the CCSID of the job.
- | All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and
- | Locales” on page 524 for more information.

## Description

The `_Rformat()` function sets the record format to *fmt* for the file specified by *fp*.

The *fmt* parameter is a null-ended C string. The *fmt* parameter must be in uppercase.

The `_Rformat()` function is valid for multi-format logical database, DDM files, display, ICF and printer files.

## Return Value

The `_Rformat()` function returns void. See Table 12 on page 507 and Table 14 on page 510 for errno settings.

## Example that uses `_Rformat()`

This example shows how `_Rformat()` is used.

```

#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char    buf[40];
    int     rc = 1;
    _RFILE  *purf;
    _RFILE  *dailyf;

    /* Open purchase display file and daily transaction file      */
    if ( ( purf = _Ropen ( "MYLIB/T1677RD3", "ar+,indicators=y" ) ) == NULL )
    {
        printf ( "Display file did not open.\n" );
        exit ( 1 );
    }

    if ( ( dailyf = _Ropen ( "MYLIB/T1677RDA", "wr,commit=y" ) ) == NULL )
    {
        printf ( "Daily transaction file did not open.\n" );
        exit ( 2 );
    }

    /* Select purchase record format */
    _Rformat ( purf, "PURCHASE" );

    /* Invite user to enter a purchase transaction.              */
    /* The _Rwrite function writes the purchase display.         */
    _Rwrite ( purf, "", 0 );
    _Rreadn ( purf, buf, sizeof(buf), __DFT );

    /* Update daily transaction file                              */
    rc = ( ( _Rwrite ( dailyf, buf, sizeof(buf) ) )->num_bytes );

    /* If the databases were updated, then commit the transaction. */
    /* Otherwise, rollback the transaction and indicate to the   */
    /* user that an error has occurred and end the application.  */
    if ( rc )
    {
        _Rcommit ( "Transaction complete" );
    }
    else
    {
        _Rrollbck ( );
        _Rformat ( purf, "ERROR" );
    }

    _Rclose ( purf );
    _Rclose ( dailyf );
}

```

## Related Information

- “\_Ropen() — Open a Record File for I/O Operations” on page 289

---

## **\_Rindara() — Set Separate Indicator Area**

### Format

```
#include <recio.h>
```

```
void _Rindara(_RFILE *fp, char *indic_buf);
```

**Language Level:** ILE C Extension

**Threadsafe:** No.

- | **Job CCSID Interface:** All character data sent to this function is expected to be in the CCSID of the job.
- | All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and
- | Locales” on page 524 for more information.

### Description

The `_Rindara()` function registers *indic\_buf* as the separate indicator area to be used by the file specified by *fp*. The file must be opened with the keyword `indicators=Y` on the `_Ropen()` function. The DDS for the file should specify also that a separate indicator area is to be used. It is generally best to initialize a separate indicator area explicitly with '0' (character) in each byte.

The `_Rindara()` function is valid for display, ICF, and printer files.

### Return Value

The `_Rindara()` function returns void. See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

**Example that uses `_Rindara()`**

```

#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>
#define PF03 2
#define IND_OFF '0'
#define IND_ON '1'

int main(void)
{
    char    buf[40];
    int     rc = 1;
    _SYSindara ind_area;
    _RFILE  *purf;
    _RFILE  *dailyf;
    /* Open purchase display file and daily transaction file */
    if ( ( purf = _Ropen ( "MYLIB/T1677RD3", "ar+,indicators=y" ) ) == NULL )
    {
        printf ( "Display file did not open.\n" );
        exit ( 1 );
    }
    if ( ( dailyf = _Ropen ( "MYLIB/T1677RDA", "wr,commit=y" ) ) == NULL )
    {
        printf ( "Daily transaction file did not open.\n" );
        exit ( 2 );
    }
    /* Associate separate indicator area with purchase file */
    _Rindara ( purf, ind_area );
    /* Select purchase record format */
    _Rformat ( purf, "PURCHASE" );
    /* Invite user to enter a purchase transaction. */
    /* The _Rwrite function writes the purchase display. */
    _Rwrite ( purf, "", 0 );
    _Rreadn ( purf, buf, sizeof(buf), __DFT );
    /* While user is entering transactions, update daily and */
    /* monthly transaction files. */
    while ( rc && ind_area[PF03] == IND_OFF )
    {
        rc = ( ( _Rwrite ( dailyf, buf, sizeof(buf) )->num_bytes );
        /* If the databases were updated, then commit transaction */
        /* otherwise, rollback the transaction and indicate to the */
        /* user that an error has occurred and end the application. */
        if ( rc )
        {
            _Rcommit ( "Transaction complete" );
        }
        else
        {
            _Rrollbck ( );
            _Rformat ( purf, "ERROR" );
        }
        _Rwrite ( purf, "", 0 );
        _Rreadn ( purf, buf, sizeof(buf), __DFT );
    }

    _Rclose ( purf );
    _Rclose ( dailyf );
}

```

## Related Information

- “\_Ropen() — Open a Record File for I/O Operations” on page 289

---

## **\_Riobuf() — Obtain I/O Feedback Information**

### Format

```
#include <recio.h>
#include <xxfdbk.h>
_XXIOFB_T *_Riofbk(_RFILE *fp);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

## Description

The `_Riofbk()` function returns a pointer to a copy of the I/O feedback area for the file that is specified by `fp`.

The `_Riofbk()` function is valid for all types of files.

## Return Value

The `_Riofbk()` function returns NULL if an error occurs. See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

## Example that uses `_Riofbk()`

```
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>
typedef struct {
    char name[20];
    char address[25];
} format1 ;
typedef struct {
    char name[8];
    char password[10];
} format2 ;
typedef union {
    format1 fmt1;
    format2 fmt2;
} formats ;

int main(void)
{
    _RFILE *fp; /* File pointer */
    _RIOFB_T *rfb; /*Pointer to the file's feedback structure */
    _XXIOFB_T *iofb; /* Pointer to the file's feedback area */
    formats buf, in_buf, out_buf; /* Buffers to hold data */
    /* Open the device file. */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit ( 1 );
    }
    _Racquire ( fp,"DEVICE1" ); /* Acquire another device. Replace
    */
    /* with actual device name. */
    _Rformat ( fp,"FORMAT1" ); /* Set the record format for the */
    /* display file. */
    rfb = _Rwrite ( fp, "", 0 ); /* Set up the display. */
    _Rpgmdev ( fp,"DEVICE2" ); /* Change the default program device. */
    /* Replace with actual device name. */
    _Rformat ( fp,"FORMAT2" ); /* Set the record format for the */
    /* display file. */
    rfb = _Rwrite ( fp, "", 0 ); /* Set up the display. */
    rfb = _Rwriterd ( fp, &buf, sizeof(buf) );
    rfb = _Rwrread ( fp, &in_buf, sizeof(in_buf), &out_buf,
```

```

        sizeof(out_buf ));
_Rreadindv ( fp, &buf, sizeof(buf), __DFT );
        /* Read from the first device that */
        /* enters data - device becomes */
        /* default program device. */
/* Determine which terminal responded first. */
iofb = _Riobk ( fp );
if ( !strcmp ( "FORMAT1 ", iofb -> rec_format, 10 ))
{
    _Rrelease ( fp, "DEVICE1" );
}
else
{
    _Rrelease(fp, "DEVICE2" );
}
/* Continue processing. */
printf ( "Data displayed is %45.45s\n", &buf);
_Rclose ( fp );
}

```

### Related Information

- “\_Ropnfbk() — Obtain Open Feedback Information” on page 293

---

## **\_Rlocate()** — Position a Record

### Format

```
#include <recio.h>
```

```
_RIOFB_T *_Rlocate(_RFILE *fp, void *key, int klen_rrn, int opts);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

- | **Job CCSID Interface:** All character data sent to this function is expected to be in the CCSID of the job.
- | All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and
- | Locales” on page 524 for more information.

### Description

The `_Rlocate()` function positions to the record in the file associated with `fp` and specified by the `key`, `klen_rrn` and `opts` parameters. The `_Rlocate()` function locks the record specified by the `key`, `klen_rrn` and `opts` parameters unless `__NO_LOCK` is specified.

The `_Rlocate()` function is valid for database and DDM files that are opened with the `_Ropen()` function. The following are valid parameters of the `_Rlocate()` function.

*key*      Points to a string containing the key fields to be used for positioning.

*klen\_rrn*

Specifies the length of the key that is used if positioning by key or the relative record number if positioning by relative record number.

*opts*      Specifies positioning options to be used for the locate operation. The possible macros are:

`__DFT`

Default to `__KEY_EQ` and lock the record for update if the file is open for updating.

- \_\_END**  
Positions to just after the last record in a file. There is no record that is associated with this position.
- \_\_END\_FRC**  
Positions to just after the last record in a file. All buffered changes are made permanent. There is no record that is associated with this position.
- \_\_FIRST**  
Positions to the first record in the access path that is currently being used by *fp*. The *key* parameter is ignored.
- \_\_KEY\_EQ**  
Positions to the first record with the specified key.
- \_\_KEY\_GE**  
Positions to the first record that has a key greater than or equal to the specified key.
- \_\_KEY\_GT**  
Positions to the first record that has a key greater than the specified key.
- \_\_KEY\_LE**  
Positions to the first record that has a key less than or equal to the specified key.
- \_\_KEY\_LT**  
Positions to the first record that has a key less than the specified key.
- \_\_KEY\_NEXTEQ**  
Positions to the next record that has a key equal to the key value with a length of *klen\_rrn*, at the current position. The *key* parameter is ignored.
- \_\_KEY\_NEXTUNQ**  
Positions to the next record with a unique key from the current position in the access path. The *key* parameter is ignored.
- \_\_KEY\_PREVEQ**  
Positions to the previous record with a key equal to the key value with a length of *klen\_rrn*, at the current position. The *key* parameter is ignored.
- \_\_KEY\_PREVUNQ**  
Positions to the previous record with a unique key from the current position in the access path. The *key* parameter is ignored.
- \_\_LAST**  
Positions to the last record in the access path that is currently being used by *fp*. The *key* parameter is ignored.
- \_\_NEXT**  
Positions to the next record in the access path that is currently being used by *fp*. The *key* parameter is ignored.
- \_\_PREVIOUS**  
Positions to the previous record in the access path that is currently being used by *fp*. The *key* parameter is ignored.
- \_\_RRN\_EQ**  
Positions to the record that has the relative record number specified on the *klen\_rrn* parameter.
- \_\_START**  
Positions to just before the first record in the file. There is no record that is associated with this position.

**\_\_START\_FRC**

Positions to just before the first record in a file. There is no record that is associated with this position. All buffered changes are made permanent.

**\_\_DATA\_ONLY**

Positions to data records only. Deleted records will be ignored.

**\_\_KEY\_NULL\_MAP**

The NULL key map is to be considered when locating to a record by key.

**\_\_NO\_LOCK**

The record that is positioned will not be locked.

**\_\_NO\_POSITION**

The position of the file is not changed, but the located record will be locked if the file is open for update.

**\_\_PRIOR**

Positions to just before the requested record.

- | If you specify a start or end option (`__START`, `__START_FRC`, `__END` or `__END_FRC`) with any other
- | options, the start or end option takes precedence and the other options might be ignored.

If you are positioned to `__START` or `__END` and perform a `_Rreads` operation, `errno` is set to `EIOERROR`.

**Return Value**

The `_Rlocate()` function returns a pointer to the `_RIOFB_T` structure associated with `fp`. If the `_Rlocate()` operation is successful, the `num_bytes` field contains 1. If `__START`, `__START_FRC`, `__END` or `__END_FRC` are specified, the `num_bytes` field is set to EOF. If the `_Rlocate()` operation is unsuccessful, the `num_bytes` field contains zero. The `key` and `rrn` fields are updated, and the `key` field will contain the complete key even if a partial key is specified.

The value of `errno` may be set to:

Table 5.

Value	Meaning
EBADKEYLN	The key length that is specified is not valid.
ENOTREAD	The file is not open for read operations
EIOERROR	A non-recoverable I/O error occurred.
EIORECERR	A recoverable I/O error occurred.

See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

**Example that uses `_Rlocate()`**

```

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE  *in;
    char    new_purchase[21] = "PEAR    1002022244";

    /* Open the file for processing in keyed sequence.          */
    if ( (in = _Ropen("MYLIB/T1677RD4", "rr+", arrseq=N")) == NULL )
    {
        printf("Open failed\n");
        exit(1);
    };

    /* Update the first record in the keyed sequence.          */

    _Rlocate(in, NULL, 0, __FIRST);
    _Rupdate(in, new_purchase, 20);

    /* Force the end of data.                                   */

    _Rfeed(in);
    _Rclose(in);
}

```

### Related Information

- “\_Ropen() — Open a Record File for I/O Operations”

---

## \_Ropen() — Open a Record File for I/O Operations

### Format

```
#include <recio.h>
```

```
_RFILE *_Ropen(const char * filename, const char * mode, ...);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

### Description

The `_Ropen()` function opens the record file specified by *filename* according to the *mode* parameter, which may be followed by optional parameters, if the *varparm* keyword parameter is specified in the *mode* parameter. The open mode and keyword parameters may be separated by a comma and one or more spaces. The `_Ropen()` function does not dynamically create database files for you. All of the files you refer to in the `_Ropen()` function must exist, or the open operation will fail.

Files that are opened by the `_Ropen()` function are closed implicitly when the activation group they are opened in, is ended. If a pointer to a file opened in one activation group is passed to another activation group and the opening activation group is ended, the file pointer will no longer be valid.

The `_Ropen()` function applies to all types of files. The *filename* variable is any valid i5/OS system file name.

The *mode* parameter specifies the type of access that is requested for the file. It contains an open mode that is followed by optional keyword parameters. The *mode* parameter may be one of the following values:

**Mode Description**

<b>rr</b>	Open an existing file for reading records.
<b>wr</b>	Open an existing file for writing records. If the file contains data, the content is cleared unless the file is a logical file.
<b>ar</b>	Open an existing file for writing records to the end of the file (append).
<b>rr+</b>	Open an existing file for reading, writing or updating records.
<b>wr+</b>	Open an existing file for reading, writing or updating records. If the file contains data, the content is cleared unless the file is a logical file.
<b>ar+</b>	Open an existing file for reading and writing records. All data is written to the end of the file.

The *mode* may be followed by any of the following keyword parameters:

**Keyword Description**

**arrseq=*value***

Where *value* can be:

- Y** Specifies that the file is processed in arrival sequence.
- N** Specifies that the file is processed using the access path that is used when the file was created. This is the default.

**blkrcd=*value***

Where *value* can be:

- Y** Performs record blocking. The i5/OS operating system determines the most efficient block size for you. This parameter is valid for database, DDM, diskette and tape files. It is only valid for files opened for input-only or output-only (modes rr, wr, or ar).
- N** Does not perform record blocking. This is the default.

**ccsid=*value***

Specifies the CCSID that is used for translation of the file. The default is 0 which indicates that the job CCSID is used.

**commit=*value***

Where *value* can be:

- Y** Specifies that the database file is opened under commitment control. Commitment control must have been set up prior to this.
- N** Specifies that the database file is not opened under commitment control. This is the default.

**dupkey=*value***

*value* can be:

- Y** Duplicate key values will be flagged in the \_RIOFB\_T structure.
- N** Duplicate key values will not be flagged. This is the default.

**indicators=*value***

Indicators are valid for printer, display, and ICF files. *value* can be:

- Y** The indicators that are associated with the file are returned in a separate indicator area instead of in the I/O buffers.

N The indicators are returned in the I/O buffers. This is the default.

**lrecl=*value***

The length, in bytes, for fixed length records, and the maximum length for variable length records. This parameter is valid for diskette, display, printer, tape, and save files.

**nullcap=*value***

Where *value* can be:

Y The program is capable of handling null fields in records. This is valid for database and DDM files.

N The program cannot handle null fields in records. This is the default.

**riofb=*value***

Where *value* can be:

Y All fields in the `_RIOFB_T` structure are updated by any I/O operation that returns a pointer to the `_RIOFB_T` structure. However, the `blk_filled_by` field is not updated when using the `_Rreadk` function. This is the default.

N Only the `num_bytes` field in the `_RIOFB_T` structure is updated.

**rtncode=*value***

Where *value* can be:

Y Use this option to bypass exception generation and handling. This will improve performance in the end-of-file and record-not-found cases. If the end-of-file is encountered, `num_bytes` will be set to EOF, but no `errno` values will be generated. If no record is found, `num_bytes` will be set to zero, and `errno` will be set to EIORECERR. This parameter is only valid for database and DDM files. For DDM files, `num_bytes` is not updated for `_Rfeed`.

N The normal exception generation and handling process will occur for the cases of end-of-file and record-not-found. This is the default.

**secure=*value***

Where *value* can be:

Y Secures the file from overrides.

N Does not secure the file from overrides. This is the default.

**splfname=(*value*)**

For spooled output only. Where *value* can be:

\*FILE The name of the printer file is used for the spooled output file name.

**spool-file-name**

Specify the name of the spooled output file. A maximum of 10 characters can be used.

**usrdta=(*value*)**

To specify, for spooled output only, user-specified data that identifies the file.

**user-data**

Specify up to 10 characters of user-specified text.

**varparm=(*list*)**

Where (*list*) is a list of optional keywords indicating which optional parameters will be passed to `_Ropen()`. The order of the keywords within the list indicates the order that the optional parameters will appear after the *mode* parameter. The following is a valid optional keyword:

**lvlchk** The `lvlchk` keyword is used in conjunction with the `lvlchk` option on `#pragma mapinc`. When this keyword is used, a pointer to an object of type `_LVLCHK_T` (generated by `#pragma mapinc`) must be specified after the *mode* parameter on the `_Ropen()` function.

For more details on this pointer, see the `lvchk` option of `#pragma mapinc` in the *WebSphere Development Studio: ILE C/C++ Programmer's Guide*.

**vlr=*value***

Variable length record, where *value* is the minimum length of bytes of a record to be written to the file. The value can equal -1, or range from 0 to the maximum record length of the file. This parameter is valid for database and DDM files.

When VLR processing is required, `_Ropen()` will set `min_length` field. If the default value is not used, the minimum value that is provided by the user will be directly copied into `min_length` field. If the default value is specified, `_Ropen()` gets the minimum length from DB portion of the open data path.

### Return Value

The `_Ropen()` function returns a pointer to a structure of type `_RFILE` if the file is opened successfully. It returns `NULL` if opening the file is unsuccessful.

The value of `errno` may be set to:

#### Value Meaning

##### EBADMODE

The file mode that is specified is not valid.

##### EBADNAME

The file name that is specified is not valid.

##### ECONVERT

A conversion error occurred.

##### ENOTOPEN

The file is not open.

##### EIOERROR

A non-recoverable I/O error occurred.

##### EIORECERR

A recoverable I/O error occurred.

See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

**Example that uses `_Ropen()`**

```

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE    *fp;

    /* Open the file for processing in arrival sequence.          */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+", arrseq=Y" )) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }
    else
        /* Do some processing */;

    _Rclose ( fp );
}

```

### Related Information

- “\_Rclose() — Close a File” on page 258
- “<recio.h>” on page 9

---

## **\_Ropnfbk() — Obtain Open Feedback Information**

### Format

```

#include <recio.h>
#include <xxfdbk.h>

```

```

_XXOPFB_T *_Ropnfbk(_RFILE *fp);

```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

### Description

The `_Ropnfbk()` function returns a pointer to a copy of the open feedback area for the file that is specified by *fp*.

The `_Ropnfbk()` function is valid for all types of files.

### Return Value

The `_Ropnfbk()` function returns NULL if an error occurs. See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

**Example that uses `_Ropnfbk()`**

```

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence.          */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+", arrseq=Y" )) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened.        */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s\nFile:   %10.10s\n",
            opfb->library_name,
            opfb->file_name);

    _Rclose ( fp );
}

```

### Related Information

- “\_Rupfb() — Provide Information on Last I/O Operation” on page 320

---

## **\_Rpgmdev() — Set Default Program Device**

### Format

```

#include <recio.h>
int _Rpgmdev(_RFILE *fp, char *dev);

```

**Language Level:** ILE C Extension

**Threadsafe:** No.

- | **Job CCSID Interface:** All character data sent to this function is expected to be in the CCSID of the job.
- | All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and
- | Locales” on page 524 for more information.

### Description

The `_Rpgmdev()` function sets the current program device for the file that is associated with `fp` to `dev`. You must specify the device in uppercase.

The `dev` parameter is a null-ended C string.

The `_Rpgmdev()` function is valid for display, ICF, and printer files.

### Return Value

The `_Rpgmdev()` function returns 1 if the operation is successful or zero if the device specified has not been acquired for the file. See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

**Example that uses `_Rpgmdev()`**

```

#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
    char name[20];
    char address[25];
} format1 ;

typedef struct {
    char name[8];
    char password[10];
} format2 ;

typedef union {
    format1 fmt1;
    format2 fmt2;
} formats ;

int main(void)
{
    _RFILE *fp; /* File pointer */
    _RIOFB_T *rfb; /*Pointer to the file's feedback structure */
    formats buf, in_buf, out_buf; /* Buffers to hold data */

    /* Open the device file. */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit ( 1 );
    }

    _Rpgmdev ( fp,"DEVICE2" );/* Change the default program device. */
    /* Replace with actual device name. */

    _Rformat ( fp,"FORMAT2" ); /* Set the record format for the */
    /* display file. */

    rfb = _Rwrite ( fp, "", 0 ); /* Set up the display. */
    rfb = _Rwriterd ( fp, &buf, sizeof(buf) );

    rfb = _Rrread ( fp, &in_buf, sizeof(in_buf), &out_buf,
        sizeof(out_buf) );

    /* Continue processing. */

    _Rclose ( fp );
}

```

### Related Information

- “\_Racquire() — Acquire a Program Device” on page 257
- “\_Rrelease() — Release a Program Device” on page 314

---

## **\_Rreadd() — Read a Record by Relative Record Number**

### Format

```

#include <recio.h>

_RIOFB_T *_Rreadd (_RFILE *fp, void *buf, size_t size,
    int opts, long rrm);

```

**Language Level:** ILE C Extension

**Threadsafe:** Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

## Description

The `_Rread()` function reads the record that is specified by *rrn* in the arrival sequence access path for the file that is associated with *fp*. The `_Rread()` function locks the record specified by the *rrn* unless `__NO_LOCK` is specified. If the file is a keyed file, the keyed access path is ignored. Up to *size* number of bytes are copied from the record into *buf* (move mode only).

The following parameters are valid for the `_Rread()` function.

- buf* Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.
- size* Specifies the number of bytes that are to be read and stored in *buf*. If locate mode is used, this parameter is ignored.
- rrn* The relative record number of the record to be read.
- opts* Specifies the processing and access options for the file. The possible options are:
- `__DFT`  
If the file is opened for updating, then the record being read is locked for update. The previously locked record will no longer be locked.
  - `__NO_LOCK`  
Does not lock the record being positioned to.

The `_Rread()` function is valid for database, DDM and display (subfiles) files.

## Return Value

The `_Rread()` function returns a pointer to the `_RIOFB_T` structure associated with *fp*. If the `_Rread()` operation is successful the *num\_bytes* field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). If `blkrcd=Y` and `riofb=Y` are specified, the `blk_count` and the `blk_filled_by` fields of the `_RIOFB_T` structure are updated. The *key* and *rrn* fields are also updated. If the file associated with *fp* is a display file, the `sysparm` field is updated. If it is unsuccessful, the *num\_bytes* field is set to a value less than *size* and `errno` will be changed.

The value of `errno` may be set to:

### Value Meaning

#### ENOTREAD

The file is not open for read operations.

#### ETRUNC

Truncation occurred on an I/O operation.

#### EIOERROR

A non-recoverable I/O error occurred.

#### EIOECERR

A recoverable I/O error occurred.

See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

## Example that uses `_Rread()`

```

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence.          */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+", arrseq=Y" )) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened.        */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s\nFile:   %10.10s\n",
            opfb->library_name,
            opfb->file_name);

    /* Get the second record.                                     */
    _Rreadd ( fp, NULL, 20, __DFT, 2 );
    printf ( "Second record: %10.10s\n", *(fp->in_buf) );

    _Rclose ( fp );
}

```

## Related Information

- “\_Rreadf() — Read the First Record”
- “\_Rreadindv() — Read from an Invited Device” on page 299
- “\_Rreadk() — Read a Record by Key” on page 302
- “\_Rreadl() — Read the Last Record” on page 305
- “\_Rreadn() — Read the Next Record” on page 306
- “\_Rreadnc() — Read the Next Changed Record in a Subfile” on page 308
- “\_Rreadp() — Read the Previous Record” on page 310
- “\_Rreads() — Read the Same Record” on page 312

---

## **\_Rreadf() — Read the First Record**

### Format

```

#include <recio.h>

_RIOFB_T *_Rreadf (_RFILE *fp, void *buf, size_t size, int opts);

```

**Language Level:** ILE C Extension

**Threadsafe:** Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

### Description

The `_Rreadf()` function reads the first record in the access path that is currently being used for the file specified by `fp`. The access path may be keyed sequence or arrival sequence. The `_Rreadf()` function locks the first record unless `__NO_LOCK` is specified. Up to `size` number of bytes are copied from the record into `buf` (move mode only).

The following are valid parameters for the `_Rreadf()` function.

- buf* This parameter points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.
- size* This parameter specifies the number of bytes that are to be read and stored in *buf*. If locate mode is used, this parameter is ignored.
- opts* This parameter specifies the processing and access options for the file. The possible options are:
- \_\_DFT**  
If the file is opened for updating, then the record being read or positioned to is locked for update. The previously locked record will no longer be locked.
  - \_\_NO\_LOCK**  
Does not lock the record being positioned to.

The `_Rreadf()` function is valid for database and DDM files.

### Return Value

The `_Rreadf()` function returns a pointer to the `_RIOFB_T` structure that is specified by *fp*. If the `_Rreadf()` operation is successful the *num\_bytes* field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). The *key* and *rrn* fields are updated. If record blocking is taking place, the *blk\_count* and *blk\_filled\_by* fields are updated. The *num\_bytes* field is set to EOF if the file is empty. If it is unsuccessful, the *num\_bytes* field is set to a value less than *size*, and *errno* is changed.

The value of *errno* may be set to:

#### Value Meaning

##### ENOTREAD

The file is not open for read operations.

##### ETRUNC

Truncation occurred on an I/O operation.

##### EIOERROR

A non-recoverable I/O error occurred.

##### EIORECERR

A recoverable I/O error occurred.

See Table 12 on page 507 and Table 14 on page 510 for *errno* settings.

### Example that uses `_Rreadf()`

```

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence.          */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+", arrseq=Y" )) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened.        */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s\nFile:   %10.10s\n",
            opfb->library_name,
            opfb->file_name);

    /* Get the first record.                                      */
    _Rreadf ( fp, NULL, 20, _DFT );
    printf ( "First record: %10.10s\n", *(fp->in_buf) );

    /* Delete the first record.                                  */
    _Rdelete ( fp );

    _Rclose ( fp );
}

```

### Related Information

- “\_Rreadd() — Read a Record by Relative Record Number” on page 295
- “\_Rreadindv() — Read from an Invited Device”
- “\_Rreadk() — Read a Record by Key” on page 302
- “\_Rreadl() — Read the Last Record” on page 305
- “\_Rreadn() — Read the Next Record” on page 306
- “\_Rreadnc() — Read the Next Changed Record in a Subfile” on page 308
- “\_Rreadp() — Read the Previous Record” on page 310
- “\_Rreads() — Read the Same Record” on page 312

---

## **\_Rreadindv() — Read from an Invited Device**

### Format

```
#include <recio.h>
```

```
_RIOFB_T *_Rreadindv(_RFILE *fp, void *buf, size_t size, int opts);
```

**Language Level:** ILE C Extension

**Threadsafe:** No.

### Description

The `_Rreadindv()` function reads data from an invited device.

The following are valid parameters for the `_Rreadindv()` function.

- buf* Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.
- size* Specifies the number of bytes that are to be read and stored in *buf*. If locate mode is used, this parameter is ignored.
- opts* Specifies the processing options for the file. Possible values are:
- \_\_DFT**  
If the file is opened for updating, then the record being read or positioned to is locked. Otherwise, the option is ignored.

The `_Rreadindv()` function is valid for display and ICF files.

### Return Value

The `_Rreadindv()` function returns a pointer to the `_RIOFB_T` structure that is associated with *fp*. If the `_Rreadindv()` function is successful, the `num_bytes` field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). The `sysparm` and `rrn` (for subfiles) fields are also updated. The `num_bytes` field is set to EOF if the file is empty. If the `_Rreadindv()` function is unsuccessful, the `num_bytes` field is set to a value less than the value of *size* and the `errno` will be changed.

The value of `errno` may be set to:

#### Value Meaning

##### ENOTREAD

The file is not open for read operations.

##### ETRUNC

Truncation occurred on an I/O operation.

##### EIOERROR

A non-recoverable I/O error occurred.

##### EIORECERR

A recoverable I/O error occurred.

See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

### Example that uses `_Rreadindv()`

```

#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>
typedef struct {
    char name[20];
    char address[25];
} format1 ;
typedef struct {
    char name[8];
    char password[10];
} format2 ;
typedef union {
    format1 fmt1;
    format2 fmt2;
} formats ;
int main(void)
{
    _RFILE *fp;                /* File pointer
    */
    _RIOFB_T *rfb;            /* Pointer to the file's feedback structure
    */
    _XXIOFB_T *iofb;         /* Pointer to the file's feedback area
    */
    formats buf, in_buf, out_buf /* Buffers to hold data
    */
    /* Open the device file.
    */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit ( 1 );
    }
    _Racquire ( fp,"DEVICE1" ); /* Acquire another device. Replace
    */
    /* with actual device name.
    */
    _Rformat ( fp,"FORMAT1" ); /* Set the record format for the
    */
    /* display file.
    */
    rfb = _Rwrite ( fp, "", 0 ); /* Set up the display.
    */
    _Rpgmdev ( fp,"DEVICE2" ); /* Change the default program device.
    */
    /* Replace with actual device name.
    */
    _Rformat ( fp,"FORMAT2" ); /* Set the record format for the
    */
    /* display file.
    */
    rfb = _Rwrite ( fp, "", 0 ); /* Set up the display.
    */
    rfb = _Rwriterd ( fp, &buf, sizeof(buf) );
    rfb = _Rwrread ( fp, &in_buf, sizeof(in_buf), &out_buf,
        sizeof(out_buf ));
    _Rreadindv ( fp, &buf, sizeof(buf), __DFT );
    /* Read from the first device that
    */
    /* enters data - device becomes
    */
    /* default program device.
    */
    /* Determine which terminal responded first.
    */
    iofb = _Riofbk ( fp );
    if ( !strcmp ( "FORMAT1 ", iofb -> rec_format, 10 ))
    {
        _Rrelease ( fp, "DEVICE1" );
    }
    else
    {
        _Rrelease(fp, "DEVICE2" );
    }
    /* Continue processing.
    */
    printf ( "Data displayed is %45.45s\n", &buf);
    _Rclose ( fp );
}

```

## Related Information

- “\_Rreadd() — Read a Record by Relative Record Number” on page 295

- “\_Rreadf() — Read the First Record” on page 297
- “\_Rreadk() — Read a Record by Key”
- “\_Rreadl() — Read the Last Record” on page 305
- “\_Rreadn() — Read the Next Record” on page 306
- “\_Rreadnc() — Read the Next Changed Record in a Subfile” on page 308
- “\_Rreadp() — Read the Previous Record” on page 310
- “\_Rreads() — Read the Same Record” on page 312

---

## **\_Rreadk() — Read a Record by Key**

### **Format**

```
#include <recio.h>
```

```
_RIOFB_T *_Rreadk(_RFILE *fp, void *buf, size_t size,
                 int opts, void *key, unsigned int keylen);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

### **Description**

The `_Rreadk()` function reads the record in the keyed access path that is currently being used for the file that is associated with `fp`. Up to `size` number of bytes are copied from the record into `buf` (move mode only). The `_Rreadk()` function locks the record positioned to unless `__NO_LOCK` is specified. You must be processing the file using a keyed sequence path.

The following parameters are valid for the `_Rreadk()` function.

*buf* Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.

*size* Specifies the number of bytes that are to be read and stored in *buf*. If locate mode is used, this parameter is ignored.

*key* Points to the key to be used for reading.

*keylen* Specifies the total length of the key to be used.

*opts* Specifies the processing options for the file. Possible values are:

**\_\_DFT**

Default to `__KEY_EQ`.

**\_\_KEY\_EQ**

Positions to and reads the first record that has the specified key.

**\_\_KEY\_GE**

Positions to and reads the first record that has a key greater than or equal to the specified key.

**\_\_KEY\_GT**

Positions and reads to the first record that has a key greater than the specified key.

**\_\_KEY\_LE**

Positions to and reads the first record that has a key less than or equal to the specified key.

**\_\_KEY\_LT**

Positions to and reads the first record that has a key less than the specified key.

**\_\_KEY\_NEXTEQ**

Positions to and reads the next record that has a key equal to the key value at the current position. The *key* parameter is ignored.

**\_\_KEY\_NEXTUNQ**

Positions to and reads the next record with a unique key from the current position in the access path. The *key* parameter is ignored.

**\_\_KEY\_PREVEQ**

Positions to and reads the last record that has a key equal to the key value at the current position. The *key* parameter is ignored.

**\_\_KEY\_PREVUNQ**

Positions to and reads the previous record with a unique key from the current position in the access path. The *key* parameter is ignored.

**\_\_NO\_LOCK**

Do not lock the record for updating.

The positioning options are mutually exclusive.

The following options may be combined with the positioning options using the bit-wise OR (`|`) operator.

**\_\_KEY\_NULL\_MAP**

The NULL key map is to be considered when reading a record by key.

**\_\_NO\_LOCK**

The record that is positioned will not be locked.

The `_Rreadk()` function is valid for database and DDM files.

**Return Value**

The `_Rreadk()` function returns a pointer to the `_RIOFB_T` structure associated with *fp*. If the `_Rreadk()` operation is successful the *num\_bytes* field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). The *key* and *rrn* fields will be updated. The key field will always contain the complete key if a partial key is specified. When using record blocking with `_Rreadk()`, only one record is read into the block. Thus there are zero records remaining in the block and the *blk\_count* field of the `_RIOFB_T` structure will be updated with 0. The *blk\_filled\_by* field is not applicable to `_Rreadk()` and is not updated. If the record specified by *key* cannot be found, the *num\_bytes* field is set to zero. If you are reading a record by a partial key, then the entire key is returned in the feedback structure. If it is unsuccessful, the *num\_bytes* field is set to a value less than *size* and *errno* will be changed.

The value of *errno* may be set to:

**Value Meaning****EBADKEYLN**

The key length specified is not valid.

**ENOTREAD**

The file is not open for read operations.

**ETRUNC**

Truncation occurred on an I/O operation.

**EIOERROR**

A non-recoverable I/O error occurred.

## EIORECERR

A recoverable I/O error occurred.

See Table 12 on page 507 and Table 14 on page 510 for errno settings.

### Example that uses `_Rreadk()`

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

int main(void)
{
    _RFILE *fp;
    _RIOFB_T *fb;
    char buf[4];
    /* Create a physical file */
    system("CRTPF FILE(QTEMP/MY_FILE)");
    /* Open the file for write */
    if ( (fp = _Ropen("QTEMP/MY_FILE", "wr")) == NULL )
    {
        printf("open for write fails\n");
        exit(1);
    }
    /* write some records into the file */
    _Rwrite(fp, "KEY9", 4);
    _Rwrite(fp, "KEY8", 4);
    _Rwrite(fp, "KEY7", 4);
    _Rwrite(fp, "KEY6", 4);
    _Rwrite(fp, "KEY5", 4);
    _Rwrite(fp, "KEY4", 4);
    _Rwrite(fp, "KEY3", 4);
    _Rwrite(fp, "KEY2", 4);
    _Rwrite(fp, "KEY1", 4);
    /* Close the file */
    _Rclose(fp);
    /* Open the file for read */
    if ( (fp = _Ropen("QTEMP/MY_FILE", "rr")) == NULL )
    {
        printf("open for read fails\n");
        exit(2);
    }
    /* Read the record with key KEY3 */
    fb = _Rreadk(fp, buf, 4, __KEY_EQ, "KEY3", 4);
    printf("record %d with value %4.4s\n", fb->rrn, buf);
    /* Read the next record with key less than KEY3 */
    fb = _Rreadk(fp, buf, 4, __KEY_LT, "KEY3", 4);
    printf("record %d with value %4.4s\n", fb->rrn, buf);
    /* Read the next record with key greater than KEY3 */
    fb = _Rreadk(fp, buf, 4, __KEY_GT, "KEY3", 4);
    printf("record %d with value %4.4s\n", fb->rrn, buf);
    /* Read the next record with different key */
    fb = _Rreadk(fp, buf, 4, __KEY_NEXTUNQ, "", 4);
    printf("record %d with value %4.4s\n", fb->rrn, buf);
    /* Close the file */
    _Rclose(fp);
}
```

### Related Information

- “`_Rreadd()` — Read a Record by Relative Record Number” on page 295
- “`_Rreadf()` — Read the First Record” on page 297
- “`_Rreadindv()` — Read from an Invited Device” on page 299
- “`_Rreadl()` — Read the Last Record” on page 305

- “\_Rreadn() — Read the Next Record” on page 306
- “\_Rreadnc() — Read the Next Changed Record in a Subfile” on page 308
- “\_Rreadp() — Read the Previous Record” on page 310
- “\_Rreads() — Read the Same Record” on page 312

---

## **\_Rreadl() — Read the Last Record**

### **Format**

```
#include <recio.h>
```

```
_RIOFB_T *_Rreadl(_RFILE *fp, void *buf, size_t size, int opts);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

### **Description**

The `_Rreadl()` function reads the last record in the access path currently being used for the file specified by `fp`. The access path may be keyed sequence or arrival sequence. Up to `size` number of bytes are copied from the record into `buf` (move mode only). The `_Rreadl()` function locks the last record unless `__NO_LOCK` is specified.

The following parameters are valid for the `_Rreadl()` function.

*buf* Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.

*size* Specifies the number of bytes that are to be read and stored in *buf*. If locate mode is used, this parameter is ignored.

*opts* Specifies the processing options for the file. Possible values are:

#### **\_\_DFT**

If the file is opened for updating, then the record being read or positioned to is locked. The previously locked record will no longer be locked.

#### **\_\_NO\_LOCK**

Do not lock the record being positioned to.

The `_Rreadl()` function is valid for database and DDM files.

### **Return Value**

The `_Rreadl()` function returns a pointer to the `_RIOFB_T` structure that is associated with `fp`. If the `_Rreadl()` operation is successful the `num_bytes` field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). The `key` and `rrn` fields will be updated. If record blocking is taking place, the `blk_count` and `blk_filled_by` fields will be updated. If the file is empty, the `num_bytes` field is set to EOF. If it is unsuccessful, the `num_bytes` field is set to a value less than `size` and `errno` will be changed.

The value of `errno` may be set to:

### **Value Meaning**

#### **ENOTREAD**

The file is not open for read operations.

## ETRUNC

Truncation occurred on an I/O operation.

## EIOERROR

A non-recoverable I/O error occurred.

## EIORECERR

A recoverable I/O error occurred.

See Table 12 on page 507 and Table 14 on page 510 for errno settings.

### Example that uses `_Rreadl()`

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

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence.          */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+", arrseq=Y" )) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened.        */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s\nFile:   %10.10s\n",
            opfb->library_name,
            opfb->file_name);

    /* Get the last record.                                       */
    _Rreadl ( fp, NULL, 20, _DFT );
    printf ( "Last record: %10.10s\n", *(fp->in_buf) );

    _Rclose ( fp );
}
```

### Related Information

- “`_Rreadd()` — Read a Record by Relative Record Number” on page 295
- “`_Rreadf()` — Read the First Record” on page 297
- “`_Rreadindv()` — Read from an Invited Device” on page 299
- “`_Rreadk()` — Read a Record by Key” on page 302
- “`_Rreadn()` — Read the Next Record”
- “`_Rreadnc()` — Read the Next Changed Record in a Subfile” on page 308
- “`_Rreadp()` — Read the Previous Record” on page 310
- “`_Rreads()` — Read the Same Record” on page 312

---

## `_Rreadn()` — Read the Next Record

### Format

```
#include <recio.h>
```

```
_RIOFB_T *_Rreadn (_RFILE *fp, void *buf, size_t size, int opts);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

## Description

The `_Rreadn()` function reads the next record in the access path that is currently being used for the file that is associated with `fp`. The access path may be keyed sequence or arrival sequence. Up to `size` number of bytes are copied from the record into `buf` (move mode only). The `_Rreadn()` function locks the record positioned to unless `__NO_LOCK` is specified.

If the file associated with `fp` is opened for sequential member processing and the current record position is the last record of any member in the file except the last, `_Rreadn()` will read the first record in the next member of the file.

If an `_Rlocate()` operation positioned to a record specifying the `__PRIOR` option, `_Rreadn()` will read the record positioned to by the `_Rlocate()` operation.

If the file is open for record blocking and a call to `_Rreadp()` has filled the block, the `_Rreadn()` function is not valid if there are records remaining in the block. You can check the `blk_count` in `_RIOFB_T` to see if there are any remaining records.

The following are valid parameters for the `_Rreadn()` function.

*buf* Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.

*size* Specifies the number of bytes that are to be read and stored in *buf*. If locate mode is used, this parameter is ignored.

*opts* Specifies the processing options for the file. Possible values are:

### `__DFT`

If the file is opened for updating, then the record being read or positioned to is locked. The previously locked record will no longer be locked.

### `__NO_LOCK`

Do not lock the record being positioned to.

The `_Rreadn()` function is valid for all types of files except printer files.

## Return Value

The `_Rreadn()` function returns a pointer to the `_RIOFB_T` structure that is associated with `fp`. If the `_Rreadn()` operation is successful the `num_bytes` field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). The `key` and `rrn` fields are updated. If the file that is associated with `fp` is a display file, the `sysparm` field is also updated. If record blocking is taking place, the `blk_count` and the `blk_filled_by` fields of the `_RIOFB_T` structure are updated. If attempts are made to read beyond the last record in the file, the `num_bytes` field is set to EOF. If it is unsuccessful, the `num_bytes` field is set to a value less than `size`, and `errno` is changed. If you are using device files and specify zero as the `size`, check `errno` to determine if the function was successful.

The value of `errno` may be set to:

### Value Meaning

#### ENOTREAD

The file is not open for read operations.

## ETRUNC

Truncation occurred on an I/O operation.

## EIOERROR

A non-recoverable I/O error occurred.

## EIORECERR

A recoverable I/O error occurred.

See Table 12 on page 507 and Table 14 on page 510 for errno settings.

### Example that uses `_Rreadn()`

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

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence.          */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+, arrseq=Y" )) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened.        */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s\nFile:   %10.10s\n",
            opfb->library_name,
            opfb->file_name);

    /* Get the first record.                                     */
    _Rreadf ( fp, NULL, 20, __DFT );
    printf ( "First record: %10.10s\n", *(fp->in_buf) );

    /* Delete the second record.                                */
    _Rreadn ( fp, NULL, 20, __DFT );
    _Rdelete ( fp );

    _Rclose ( fp );
}
```

### Related Information

- “`_Rreadd()` — Read a Record by Relative Record Number” on page 295
- “`_Rreadf()` — Read the First Record” on page 297
- “`_Rreadindv()` — Read from an Invited Device” on page 299
- “`_Rreadk()` — Read a Record by Key” on page 302
- “`_Rreadl()` — Read the Last Record” on page 305
- “`_Rreadnc()` — Read the Next Changed Record in a Subfile”
- “`_Rreadp()` — Read the Previous Record” on page 310
- “`_Rreads()` — Read the Same Record” on page 312

---

## `_Rreadnc()` — Read the Next Changed Record in a Subfile

### Format

```
#include <recio.h>
_RIOFB_T *_Rreadnc(_RFILE *fp, void *buf, size_t size);
```

**Language Level:** ILE C Extension

**Threadsafe:** No.

### Description

The `_Rreadnc()` function reads the next changed record from the current position in the subfile that is associated with `fp`. The minimum *size* of data that is read from the screen are copied from the system buffer to `buf`.

The following are valid parameters for the `_Rreadnc()` function.

*buf*      Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.

*size*     Specifies the number of bytes that are to be read and stored in *buf*.

The `_Rreadnc()` function is valid for subfiles.

### Return Value

The `_Rreadnc()` function returns a pointer to the `_RIOFB_T` structure that is associated with `fp`. If the `_Rreadnc()` operation is successful the `num_bytes` field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). The `rrn` and `sysparm` fields are updated. If there are no changed records between the current position and the end of the file, the `num_bytes` field is set to EOF. If it is unsuccessful, the `num_bytes` field is set to a value less than *size*, and `errno` is changed.

The value of `errno` may be set to:

#### Value    Meaning

##### ENOTREAD

The file is not open for read operations.

##### ETRUNC

Truncation occurred on an I/O operation.

##### EIOERROR

A non-recoverable I/O error occurred.

##### EIORECERR

A recoverable I/O error occurred.

See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

**Example that uses `_Rreadnc()`**

```

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>
#define LEN      10
#define NUM_RECS 20
#define SUBFILENAME "MYLIB/T1677RD6"
#define PFILENAME  "MYLIB/T1677RDB"
typedef struct {
    char name[LEN];
    char phone[LEN];
} pf_t;
#define RECLEN sizeof(pf_t)
void init_subfile(_RFILE *, _RFILE *);

int main(void)
{
    _RFILE      *pf;
    _RFILE      *subf;
    /******
     * Open the subfile and the physical file.      *
     * *****/
    if ((pf = _Ropen(PFILENAME, "rr")) == NULL) {
        printf("can't open file %s\n", PFILENAME);
        exit(1);
    }
    if ((subf = _Ropen(SUBFILENAME, "ar+")) == NULL) {
        printf("can't open file %s\n", SUBFILENAME);
        exit(2);
    }
    /******
     * Initialize the subfile with records          *
     * from the physical file.                      *
     * *****/
    init_subfile(pf, subf);
    /******
     * Write the subfile to the display by writing   *
     * a record to the subfile control format.      *
     * *****/
    _Rformat(subf, "SFLCTL");
    _Rwrite(subf, "", 0);
    _Rreadnc(subf, "", 0);
    /******
     * Close the physical file and the subfile.     *
     * *****/
    _Rclose(pf);
    _Rclose(subf);
}

```

## Related Information

- “\_Rreadd() — Read a Record by Relative Record Number” on page 295
- “\_Rreadf() — Read the First Record” on page 297
- “\_Rreadindv() — Read from an Invited Device” on page 299
- “\_Rreadk() — Read a Record by Key” on page 302
- “\_Rreadl() — Read the Last Record” on page 305
- “\_Rreadn() — Read the Next Record” on page 306
- “\_Rreadp() — Read the Previous Record”
- “\_Rreads() — Read the Same Record” on page 312

---

## **\_Rreadp() — Read the Previous Record**

### Format

```
#include <recio.h>
_RIOFB_T *_Rreadp(_RFILE *fp, void *buf, size_t size, int opts);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

### Description

The `_Rreadp()` function reads the previous record in the access path that is currently being used for the file that is associated with `fp`. The access path may be keyed sequence or arrival sequence. Up to `size` number of bytes are copied from the record into `buf` (move mode only). The `_Rreadp()` function locks the record positioned to unless `__NO_LOCK` is specified.

If the file associated with `fp` is opened for sequential member processing and the current record position is the first record of any member in the file except the first, `_Rreadp()` will read the last record in the previous member of the file.

If the file is open for record blocking and a call to `_Rreadn()` has filled the block, the `_Rreadp()` function is not valid if there are records remaining in the block. You can check the `blk_count` in `_RIOFB_T` to see if there are any remaining records.

The following are valid parameters for the `_Rreadp()` function.

- buf* Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.
- size* Specifies the number of bytes that are to be read and stored in *buf*. If locate mode is used, this parameter is ignored.
- opts* Specifies the processing options for the file. Possible values are:
- `__DFT`  
If the file is opened for updating, then the record being read or positioned to is locked. The previously locked record will no longer be locked.
  - `__NO_LOCK`  
Do not lock the record being positioned to.

The `_Rreadp()` function is valid for database and DDM files.

### Return Value

The `_Rreadp()` function returns a pointer to the `_RIOFB_T` structure that is associated with `fp`. If the `_Rreadp()` operation is successful the `num_bytes` field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). The `key` and `rrn` fields are also updated. If record blocking is taking place, the `blk_count` and the `blk_filled_by` fields of the `_RIOFB_T` structure are updated. If attempts are made to read prior to the first record in the file, the `num_bytes` field is set to EOF. If it is unsuccessful, the `num_bytes` field is set to a value less than `size`, and `errno` is changed.

The value of `errno` may be set to:

#### Value Meaning

##### ENOTREAD

The file is not open for read operations.

## ETRUNC

Truncation occurred on an I/O operation.

## EIOERROR

A non-recoverable I/O error occurred.

## EIORECERR

A recoverable I/O error occurred.

See Table 12 on page 507 and Table 14 on page 510 for errno settings.

### Example that uses `_Rreadp()`

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

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence.          */
    if ( ( fp = _Ropen ( "MYLIB/T1677RD1", "rr+", arrseq=Y" ) ) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened.        */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s\nFile:   %10.10s\n",
            opfb->library_name,
            opfb->file_name);

    /* Get the last record.                                       */
    _Rreadl ( fp, NULL, 20, __DFT );
    printf ( "Last record: %10.10s\n", *(fp->in_buf) );

    /* Get the previous record.                                    */

    _Rreadp ( fp, NULL, 20, __DFT );
    printf ( "Next to last record: %10.10s\n", *(fp->in_buf) );

    _Rclose ( fp );
}
```

### Related Information

- “`_Rreadd()` — Read a Record by Relative Record Number” on page 295
- “`_Rreadf()` — Read the First Record” on page 297
- “`_Rreadindv()` — Read from an Invited Device” on page 299
- “`_Rreadk()` — Read a Record by Key” on page 302
- “`_Rreadl()` — Read the Last Record” on page 305
- “`_Rreadn()` — Read the Next Record” on page 306
- “`_Rreadnc()` — Read the Next Changed Record in a Subfile” on page 308
- “`_Rreads()` — Read the Same Record”

---

## `_Rreads()` — Read the Same Record

### Format

```
#include <recio.h>
_RIOFB_T *_Rreads(_RFILE *fp, void *buf, size_t size, int opts);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

### Description

The `_Rreads()` function reads the current record in the access path that is currently being used for the file that is associated with `fp`. The access path may be keyed sequence or arrival sequence. Up to `size` number of bytes are copied from the record into `buf` (move mode only). The `_Rreads()` function locks the record positioned to unless `__NO_LOCK` is specified.

If the current position in the file that is associated with `fp` has no record associated with it, the `_Rreads()` function will fail.

The `_Rreads()` function is not valid when the file is open for record blocking.

The following are valid parameters for the `_Rreads()` function.

*buf* Points to the buffer where the data that is read is to be stored. If locate mode is used, this parameter must be set to NULL.

*size* Specifies the number of bytes that are to be read and stored in *buf*. If locate mode is used, this parameter is ignored.

*opts* Specifies the processing options for the file. Possible values are:

`__DFT`

If the file is opened for updating, then the record being read or positioned to is locked. The previously locked record will no longer be locked.

`__NO_LOCK`

Do not lock the record being positioned to.

The `_Rreads()` function is valid for database and DDM files.

### Return Value

The `_Rreads()` function returns a pointer to the `_RIOFB_T` structure that is associated with `fp`. If the `_Rreads()` operation is successful the `num_bytes` field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). The `key` and `rrn` fields are also updated. If it is unsuccessful, the `num_bytes` field is set to a value less than `size`, and `errno` is changed.

The value of `errno` may be set to:

#### Value Meaning

##### ENOTREAD

The file is not open for read operations.

##### ETRUNC

Truncation occurred on an I/O operation.

##### EIOERROR

A non-recoverable I/O error occurred.

## EIORECERR

A recoverable I/O error occurred.

See Table 12 on page 507 and Table 14 on page 510 for errno settings.

### Example that uses `_Rreads()`

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

int main(void)
{
    _RFILE      *fp;
    _XXOPFB_T   *opfb;

    /* Open the file for processing in arrival sequence.          */
    if (( fp = _Ropen ( "MYLIB/T1677RD1", "rr+", arrseq=Y" )) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    }

    /* Get the library and file names of the file opened.        */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s\nFile:   %10.10s\n",
            opfb->library_name,
            opfb->file_name);

    /* Get the last record.                                       */
    _Rreadl ( fp, NULL, 20, __DFT );
    printf ( "Last record: %10.10s\n", *(fp->in_buf) );

    /* Get the same record without locking it.                    */
    _Rreads ( fp, NULL, 20, __NO_LOCK );
    printf ( "Same record: %10.10s\n", *(fp->in_buf) );

    _Rclose ( fp );
}
```

### Related Information

- “`_Rreadd()` — Read a Record by Relative Record Number” on page 295
- “`_Rreadf()` — Read the First Record” on page 297
- “`_Rreadindv()` — Read from an Invited Device” on page 299
- “`_Rreadk()` — Read a Record by Key” on page 302
- “`_Rreadl()` — Read the Last Record” on page 305
- “`_Rreadn()` — Read the Next Record” on page 306
- “`_Rreadnc()` — Read the Next Changed Record in a Subfile” on page 308
- “`_Rreadp()` — Read the Previous Record” on page 310

---

## `_Rrelease()` — Release a Program Device

### Format

```
#include <recio.h>
```

```
int _Rrelease(_RFILE *fp, char *dev);
```

**Language Level:** ILE C Extension

**Threadsafe:** No.

- | **Job CCSID Interface:** All character data sent to this function is expected to be in the CCSID of the job.
- | All character data returned by this function is in the CCSID of the job. See “Understanding CCSIDs and Locales” on page 524 for more information.

## Description

The `_Rrelease()` function releases the program device that is specified by *dev* from the file that is associated with *fp*. The device name must be specified in uppercase.

The *dev* parameter is a null-ended C string.

The `_Rrelease()` function is valid for display and ICF files.

## Return Value

The `_Rrelease()` function returns 1 if it is successful or zero if it is unsuccessful. The value of `errno` may be set to `EIOERROR` (a non-recoverable I/O error occurred) or `EIORECERR` (a recoverable I/O error occurred). See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

## Example that uses `_Rrelease()`

```
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>
typedef struct {
    char name[20];
    char address[25];
} format1 ;
typedef struct {
    char name[8];
    char password[10];
} format2 ;
typedef union {
    format1 fmt1;
    format2 fmt2;
} formats ;

int main(void)
{
    _RFILE *fp; /* File pointer */
    _RIOFB_T *rfb; /*Pointer to the file's feedback structure */
    _XXIOFB_T *iofb; /* Pointer to the file's feedback area */
    formats buf, in_buf, out_buf; /* Buffers to hold data */
    /* Open the device file. */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit ( 1 );
    }
    _Racquire ( fp,"DEVICE1" ); /* Acquire another device. Replace */
    /* with actual device name. */
    _Rformat ( fp,"FORMAT1" ); /* Set the record format for the */
    /* display file. */
    rfb = _Rwrite ( fp, "", 0 ); /* Set up the display. */
    _Rpgmdev ( fp,"DEVICE2" ); /* Change the default program device. */
    /* Replace with actual device name. */
    _Rformat ( fp,"FORMAT2" ); /* Set the record format for the */
    /* display file. */
    rfb = _Rwrite ( fp, "", 0 ); /* Set up the display. */
    rfb = _Rwriterd ( fp, &buf, sizeof(buf) );
    rfb = _Rrread ( fp, &in_buf, sizeof(in_buf), &out_buf,
        sizeof(out_buf) );
    _Rreadindv ( fp, &buf, sizeof(buf), __DFT );
```

```

/* Read from the first device that */
/* enters data - device becomes */
/* default program device. */
/* Determine which terminal responded first. */
iofb = _Riobk ( fp );
if ( !strcmp ( "FORMAT1 ", iofb -> rec_format, 10 ))
{
    _Rrelease ( fp, "DEVICE1" );
}
else
{
    _Rrelease(fp, "DEVICE2" );
}
/* Continue processing. */
printf ( "Data displayed is %45.45s\n", &buf);
_Rclose ( fp );
}

```

### Related Information

- “\_Racquire() — Acquire a Program Device” on page 257

---

## **\_Rrslck() — Release a Record Lock**

### Format

```

#include <recio.h>

int _Rrslck(_RFILE *fp);

```

**Language Level:** ILE C Extension

**Threadsafe:** Yes.

### Description

The `_Rrslck()` function releases the lock on the currently locked record for the file specified by *fp*. The file must be open for update, and a record must be locked. If the `_NO_POSITION` option was specified on the `_Rlocate()` operation that locked the record, the record released may not be the record currently positioned to.

The `_Rrslck()` function is valid for database and DDM files.

### Return Value

The `_Rrslck()` function returns 1 if the operation is successful, or zero if the operation is unsuccessful.

The value of `errno` may be set to:

#### Value Meaning

##### ENOTUPD

The file is not open for update operations.

##### EIOERROR

A non-recoverable I/O error occurred.

##### EIOECERR

A recoverable I/O error occurred.

See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

### Example that uses `_Rr1slck()`

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

int main(void)
{
    char        buf[21];
    _RFILE      *fp;
    _XXOPFB_T   *opfb;
    int         result;

    /* Open the file for processing in arrival sequence.          */
    if ( ( fp = _Ropen ( "MYLIB/T1677RD1", "rr+", arrseq=Y" ) ) == NULL )
    {
        printf ( "Open failed\n" );
        exit ( 1 );
    };

    /* Get the library and file names of the file opened.        */
    opfb = _Ropnfbk ( fp );
    printf ( "Library: %10.10s\nFile:   %10.10s\n",
            opfb->library_name,
            opfb->file_name);

    /* Get the last record.                                      */
    _Rreadl ( fp, NULL, 20, _DFT );
    printf ( "Last record: %10.10s\n", *(fp->in_buf) );

    /* _Rr1slck example.                                        */
    result = _Rr1slck ( fp );
    if ( result == 0 )
        printf("_Rr1slck failed.\n");

    _Rclose ( fp );
}
```

### Related Information

- “`_Rdelete()` — Delete a Record” on page 261

---

## `_Rrollbck()` — Roll Back Commitment Control Changes

### Format

```
#include <recio.h>

int _Rrollbck(void);
```

**Language Level:** ILE C Extension

**Threadsafe:** No.

### Description

The `_Rrollbck()` function reestablishes the last commitment boundary as the current commitment boundary. All changes that are made to the files under commitment control in the job, are reversed. All locked records are released. Any file that is open under commitment control in the job will be affected. You must specify the keyword parameter `commit=y` when the file is opened to be under commitment control. A commitment control environment must have been set up prior to this.

The `_Rrollbck()` function is valid for database and DDM files.

## Return Value

The `_Rrollbck()` function returns 1 if the operation is successful or zero if the operation is unsuccessful. The value of `errno` may be set to `EIOERROR` (a non-recoverable I/O error occurred) or `EIORECERR` (a recoverable I/O error occurred). See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

### Example that uses `_Rrollbck()`

```
#include <stdio.h>
#include <recio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char    buf[40];
    int     rc = 1;
    _RFILE  *purf;
    _RFILE  *dailyf;

    /* Open purchase display file and daily transaction file      */
    if ( ( purf = _Ropen ( "MYLIB/T1677RD3", "ar+,indicators=y" ) ) == NULL )
    {
        printf ( "Display file did not open.\n" );
        exit ( 1 );
    }

    if ( ( dailyf = _Ropen ( "MYLIB/T1677RDA", "wr,commit=y" ) ) == NULL )
    {
        printf ( "Daily transaction file did not open.\n" );
        exit ( 2 );
    }

    /* Select purchase record format */
    _Rformat ( purf, "PURCHASE" );

    /* Invite user to enter a purchase transaction.                */
    /* The _Rwrite function writes the purchase display.           */
    _Rwrite ( purf, "", 0 );
    _Rreadn ( purf, buf, sizeof(buf), __DFT );

    /* Update daily transaction file                                */
    rc = ( ( _Rwrite ( dailyf, buf, sizeof(buf) )->num_bytes );

    /* If the databases were updated, then commit the transaction. */
    /* Otherwise, rollback the transaction and indicate to the     */
    /* user that an error has occurred and end the application.     */
    if ( rc )
    {
        _Rcommit ( "Transaction complete" );
    }
    else
    {
        _Rrollbck ( );
        _Rformat ( purf, "ERROR" );
    }

    _Rclose ( purf );
    _Rclose ( dailyf );
}
```

### Related Information

- “`_Rcommit()` — Commit Current Record” on page 259
- *Recovering your system* manual

---

## **\_Rupdate() — Update a Record**

### **Format**

```
#include <recio.h>
```

```
_RIOFB_T *_Rupdate(_RFILE *fp, void *buf, size_t size);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

### **Description**

The `_Rupdate()` function updates the record that is currently locked for update in the file that is specified by `fp`. The file must be open for update. A record is locked for update by reading or locating to it unless `__NO_LOCK` is specified on the read or locate operation. If the `__NO_POSITION` option is specified on a locate operation the record updated may not be the record currently positioned to. After the update operation, the updated record is no longer locked.

The number of bytes that are copied from `buf` to the record is the minimum of `size` and the record length of the file (move mode only). If `size` is greater than the record length, the data is truncated, and `errno` is set to `ETRUNC`. One complete record is always written to the file. If the `size` is less than the record length of the file, the remaining data in the record will be the original data that was read into the system buffer by the read that locked the record. If a locate operation locked the record, the remaining data will be what was in the system input buffer prior to the locate.

The `_Rupdate()` function can be used to update deleted records and key fields. A deleted record that is updated will no longer be marked as a deleted record. In both of these cases any keyed access paths defined for `fp` will be changed.

**Note:** If locate mode is being used, `_Rupdate()` works on the data in the file's input buffer.

The `_Rupdate()` function is valid for database, display (subfiles) and DDM files.

### **Return Value**

The `_Rupdate()` function returns a pointer to the `_RIOFB_T` structure associated with `fp`. If the `_Rupdate()` function is successful, the `num_bytes` field is set to the number of bytes transferred from the system buffer to the user's buffer (move mode) or the record length of the file (locate mode). If `fp` is a display file, the `sysparm` field is updated. If the `_Rupdate()` function is unsuccessful, the `num_bytes` field is set to a value less than the `size` specified (move mode) or zero (locate mode). The `errno` value will also be changed.

The value of `errno` may be set to:

#### **Value Meaning**

##### **ENOTUPD**

The file is not open for update operations.

##### **EIOERROR**

A non-recoverable I/O error occurred.

##### **EIORECERR**

A recoverable I/O error occurred.

See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

### Example that uses `_Rupdate()`

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

int main(void)
{
    _RFILE    *in;
    char      new_purchase[21] = "PEAR    1002022244";

    /* Open the file for processing in keyed sequence.          */
    if ( (in = _Ropen("MYLIB/T1677RD4", "rr+", arrseq=N")) == NULL )
    {
        printf("Open failed\n");
        exit(1);
    };

    /* Update the first record in the keyed sequence.          */

    _Rlocate(in, NULL, 0, __FIRST);
    _Rupdate(in, new_purchase, 20);

    /* Force the end of data.                                  */

    _Rfeed(in);
    _Rclose(in);
}
```

### Related Information

- “`_Rread()` — Read a Record by Relative Record Number” on page 295
- “`_Rreadf()` — Read the First Record” on page 297
- “`_Rreadindv()` — Read from an Invited Device” on page 299
- “`_Rreadk()` — Read a Record by Key” on page 302
- “`_Rreadl()` — Read the Last Record” on page 305
- “`_Rreadn()` — Read the Next Record” on page 306
- “`_Rreadnc()` — Read the Next Changed Record in a Subfile” on page 308
- “`_Rreadp()` — Read the Previous Record” on page 310
- “`_Rreads()` — Read the Same Record” on page 312

---

## `_Rupfb()` — Provide Information on Last I/O Operation

### Format

```
#include <recio.h>

_RIOFB_T *_Rupfb(_RFILE *fp);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

### Description

The `_Rupfb()` function updates the feedback structure associated with the file specified by *fp* with information about the last I/O operation. The `_RIOFB_T` structure will be updated even if `riofb=N` was

specified when the file was opened. The `num_bytes` field of the `_RIOFB_T` structure will not be updated. See “<recio.h>” on page 9 for a description of the `_RIOFB_T` structure.

The `_Rupfb()` function is valid for all types of files.

## Return Value

The `_Rupfb()` function returns a pointer to the `_RIOFB_T` structure specified by `fp`. See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

### Example that uses `_Rupfb()`

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

int main(void)
{
    _RFILE    *fp;
    _RIOFB_T  *fb;
    /* Create a physical file */
    system("CRTPF FILE(QTEMP/MY_FILE) RCDLEN(80)");
    /* Open the file for write */
    if ( (fp = _Ropen("QTEMP/MY_FILE", "wr")) == NULL )
    {
        printf("open for write fails\n");
        exit(1);
    }
    /* Write some records into the file */
    _Rwrite(fp, "This is record 1", 16);
    _Rwrite(fp, "This is record 2", 16);
    _Rwrite(fp, "This is record 3", 16);
    _Rwrite(fp, "This is record 4", 16);
    _Rwrite(fp, "This is record 5", 16);
    _Rwrite(fp, "This is record 6", 16);
    _Rwrite(fp, "This is record 7", 16);
    _Rwrite(fp, "This is record 8", 16);
    _Rwrite(fp, "This is record 9", 16);
    /* Close the file */
    _Rclose(fp);
    /* Open the file for read */
    if ( (fp = _Ropen("QTEMP/MY_FILE", "rr, blkrcd = y")) == NULL )
    {
        printf("open for read fails\n");
        exit(2);
    }
    /* Read some records */
    _Rreadn(fp, NULL, 80, __DFT);
    _Rreadn(fp, NULL, 80, __DFT);
    /* Call _Rupfb and print feed back information */
    fb = _Rupfb(fp);
    printf("record number ----- %d\n",
           fb->rrrn);
    printf("number of bytes read ----- %d\n",
           fb->num_bytes);
    printf("number of records remaining in block --- %hd\n",
           fb->blk_count);
    if ( fb->blk_filled_by == __READ_NEXT )
    {
        printf("block filled by ----- __READ_NEXT\n");
    }
    else
    {
        printf("block filled by ----- __READ_PREV\n");
    }
}
```

```

    }
    /* Close the file */
    _Rclose(fp);
}

```

## Related Information

- “\_Ropnfbk() — Obtain Open Feedback Information” on page 293

---

## \_Rwrite() — Write the Next Record

### Format

```
#include <recio.h>
```

```
_RIOFB_T * _Rwrite(_RFILE *fp, void *buf, size_t size);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

### Description

The `_Rwrite()` function has two modes: move and locate. When `buf` points to a user buffer, `_Rwrite()` is in move mode. When `buf` is NULL, the function is in locate mode.

The `_Rwrite()` function appends a record to the file specified by `fp`. The number of bytes copied from `buf` to the record is the minimum of `size` and the record length of the file (move mode only). If `size` is greater than the record length, the data is truncated and `errno` is set to `ETRUNC`. One complete record is always written if the operation is successful.

If you are using `_Ropen()` and then `_Rwrite()` to output records to a source physical file, the sequence numbers must be manually appended.

The `_Rwrite()` function has no effect on the position of the file for a subsequent read operation.

| Records might be lost although the `_Rwrite()` function indicates success when the following items are true:

- | • Record blocking is taking place.
- | • The file associated with `fp` is approaching the limit of the number of records it can contain and the file cannot be extended.
- | • Multiple writers are writing to the same file.

| Because the output is buffered, the `_Rwrite` routine returns success that indicates the record is successfully copied to the buffer. However, when the buffer is flushed, the routine might fail because the file has been filled to capacity by another writer. In this case, the `_Rwrite()` function indicates that an error occurred only on the call to the `_Rwrite()` function that sends the data to the file.

The `_Rwrite()` function is valid for all types of files.

### Return Value

The `_Rwrite()` function returns a pointer to the `_RIOFB_T` structure that is associated with `fp`. If the `_Rwrite()` operation is successful the `num_bytes` field is set to the number of bytes written for both move mode and locate mode. The function transfers the bytes from the user’s buffer to the system buffer. If record blocking is taking place, the function only updates the `rrn` and `key` fields when it sends the block

to the database. If *fp* is a display, ICF or printer file, the function updates the *sysparm* field. If it is unsuccessful, the *num\_bytes* field is set to a value less than *size* specified (move mode) or zero (locate mode) and *errno* is changed.

The value of *errno* may be set to:

#### Value Meaning

##### ENOTWRITE

The file is not open for write operations.

##### ETRUNC

Truncation occurred on an I/O operation.

##### EIOERROR

A non-recoverable I/O error occurred.

##### EIORECERR

A recoverable I/O error occurred.

See Table 12 on page 507 and Table 14 on page 510 for *errno* settings.

#### Example that uses `_Rwrite()`

```
#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>
typedef struct {
    char name[20];
    char address[25];
} format1 ;
typedef struct {
    char name[8];
    char password[10];
} format2 ;
typedef union {
    format1 fmt1;
    format2 fmt2;
} formats ;

int main(void)
{
    _RFILE *fp; /* File pointer */
    _RIOFB_T *rfb; /*Pointer to the file's feedback structure */
    _XXIOFB_T *iofb; /* Pointer to the file's feedback area */
    formats buf, in_buf, out_buf; /* Buffers to hold data */
    /* Open the device file. */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit ( 1 );
    }
    _Racquire ( fp,"DEVICE1" ); /* Acquire another device. Replace */
                                /* with actual device name. */
    _Rformat ( fp,"FORMAT1" ); /* Set the record format for the */
                                /* display file. */
    rfb = _Rwrite ( fp, "", 0 ); /* Set up the display. */
    _Rpgmdev ( fp,"DEVICE2" ); /* Change the default program device. */
                                /* Replace with actual device name. */
    _Rformat ( fp,"FORMAT2" ); /* Set the record format for the */
                                /* display file. */
    rfb = _Rwrite ( fp, "", 0 ); /* Set up the display. */
    rfb = _Rwriterd ( fp, &buf, sizeof(buf) );
    rfb = _Rwrread ( fp, &in_buf, sizeof(in_buf), &out_buf,
                    sizeof(out_buf ));
```

```

    _Rreadindv ( fp, &buf, sizeof(buf), __DFT );
                                /* Read from the first device that */
                                /* enters data - device becomes */
                                /* default program device. */
/* Determine which terminal responded first. */
iofb = _Riofbk ( fp );
if ( !strncmp ( "FORMAT1 ", iofb -> rec_format, 10 ))
{
    _Rrelease ( fp, "DEVICE1" );
}
else
{
    _Rrelease(fp, "DEVICE2" );
}
/* Continue processing. */
printf ( "Data displayed is %45.45s\n", &buf);
_Rclose ( fp );
}

```

### Related Information

- “\_Rwrited() — Write a Record Directly”
- “\_Rwriterd() — Write and Read a Record” on page 327
- “\_Rwrread() — Write and Read a Record (separate buffers)” on page 328

---

## **\_Rwrited() — Write a Record Directly**

### Format

```
#include <recio.h>
```

```
_RIOFB_T *_Rwrited(_RFILE *fp, void *buf, size_t size, unsigned long rrm);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes. However, if the file pointer is passed among threads, the I/O feedback area is shared among those threads.

### Description

The `_Rwrited()` function writes a record to the file associated with `fp` at the position specified by `rrm`. The `_Rwrited()` function will only write over deleted records. The number of bytes copied from `buf` to the record is the minimum of `size` and the record length of the file (move mode only). If `size` is greater than the record length, the data is truncated, and `errno` is set to `ETRUNC`. One complete record is always written if the operation is successful.

The `_Rwrited()` function has no effect on the position of the file for a read operation.

The `_Rwrited()` function is valid for database, DDM and subfiles.

### Return Value

The `_Rwrited()` function returns a pointer to the `_RIOFB_T` structure associated with `fp`. If the `_Rwrited()` operation is successful the `num_bytes` field is set to the number of bytes transferred from the user's buffer to the system buffer (move mode) or the record length of the file (locate mode). The `rrm` field is updated. If `fp` is a display file, the `sysparm` field is updated. If it is unsuccessful, the `num_bytes` field is set to a value less than `size` specified (move mode) or zero (locate mode) and `errno` is changed.

The value of `errno` may be set to:

### Value Meaning

**ENOTWRITE**

The file is not open for write operations.

**ETRUNC**

Truncation occurred on an I/O operation.

**EIOERROR**

A non-recoverable I/O error occurred.

**EIORECERR**

A recoverable I/O error occurred.

See Table 12 on page 507 and Table 14 on page 510 for errno settings.

**Example that uses `_Rwrited()`**

```

#include <stdio.h>
#include <stdlib.h>
#include <recio.h>
#define LEN      10
#define NUM_RECS 20
#define SUBFILENAME "MYLIB/T1677RD6"
#define PFILENAME  "MYLIB/T1677RDB"
typedef struct {
    char name[LEN];
    char phone[LEN];
} pf_t;
#define RECLLEN sizeof(pf_t)
void init_subfile(_RFILE *, _RFILE *);
int main(void)
{
    _RFILE      *pf;
    _RFILE      *subf;
    /* Open the subfile and the physical file.      */
    if ((pf = _Ropen(PFILENAME, "rr")) == NULL) {
        printf("can't open file %s\n", PFILENAME);
        exit(1);
    }
    if ((subf = _Ropen(SUBFILENAME, "ar+")) == NULL) {
        printf("can't open file %s\n", SUBFILENAME);
        exit(2);
    }
    /* Initialize the subfile with records          */
    /* from the physical file.                      */
    init_subfile(pf, subf);
    /* Write the subfile to the display by writing   */
    /* a record to the subfile control format.      */
    _Rformat(subf, "SFLCTL");
    _Rwrite(subf, "", 0);
    _Rreadnc(subf, "", 0);
    /* Close the physical file and the subfile.    */
    _Rclose(pf);
    _Rclose(subf);
}
void init_subfile(_RFILE *pf, _RFILE *subf)
{
    _RIOFB_T      *fb;
    int           i;
    pf_t          record;
    /* Select the subfile record format.           */
    _Rformat(subf, "SFL");
    for (i = 1; i <= NUM_RECS; i++) {
        fb = _Rreadn(pf, &record, RECLLEN, __DFT);
        if (fb->num_bytes != RECLLEN) {
            printf("%d\n", fb->num_bytes);
            printf("%d\n", RECLLEN);
            printf("error occurred during read\n");
            exit(3);
        }
        fb = _Rwrited(subf, &record, RECLLEN, i);
        if (fb->num_bytes != RECLLEN) {
            printf("error occurred during write\n");
            exit(4);
        }
    }
}

```

## Related Information

- “\_Rwrite() — Write the Next Record” on page 322
- “\_Rwriterd() — Write and Read a Record” on page 327
- “\_Rwrread() — Write and Read a Record (separate buffers)” on page 328

---

## **\_Rwriterd() — Write and Read a Record**

### **Format**

```
#include <recio.h>
_RIOFB_T *_Rwriterd(_RFILE *fp, void *buf, size_t size);
```

**Language Level:** ILE C Extension

**Threadsafe:** No.

### **Description**

The `_Rwriterd()` function performs a write and then a read operation on the file that is specified by *fp*. The minimum of *size* and the length of the current record format determines the amount of data to be copied between the system buffer and *buf* for both the write and read parts of the operation. If *size* is greater than the record length of the current format, `errno` is set to `ETRUNC` on the write part of the operation. If *size* is less than the length of the current record format, `errno` is set to `ETRUNC` on the read part of the operation.

The `_Rwriterd()` function is valid for display and ICF files.

### **Return Value**

The `_Rwriterd()` function returns a pointer to the `_RIOFB_T` structure that is associated with *fp*. If the `_Rwriterd()` operation is successful, the `num_bytes` field is set to the number of bytes transferred from the system buffer to *buf* on the read part of the operation (move mode) or the record length of the file (locate mode).

The value of `errno` may be set to:

#### **Value Meaning**

##### **ENOTUPD**

The file is not open for update operations.

##### **ETRUNC**

Truncation occurred on an I/O operation.

##### **EIOERROR**

A non-recoverable I/O error occurred.

##### **EIORECERR**

A recoverable I/O error occurred.

See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

**Example that uses `_Rwriterd()`**

```

#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
    char name[20];
    char address[25];
} format1 ;

typedef struct {
    char name[8];
    char password[10];
} format2 ;

typedef union {
    format1 fmt1;
    format2 fmt2;
} formats ;

int main(void)
{
    _RFILE *fp; /* File pointer */
    _RIOFB_T *rfb; /*Pointer to the file's feedback structure */
    formats buf, in_buf, out_buf; /* Buffers to hold data */

    /* Open the device file. */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit ( 1 );
    }

    _Rpgmdev ( fp,"DEVICE2" );/* Change the default program device. */
                               /* Replace with actual device name. */

    _Rformat ( fp,"FORMAT2" ); /* Set the record format for the */
                               /* display file. */

    rfb = _Rwrite ( fp, "", 0 ); /* Set up the display. */
    rfb = _Rwriterd ( fp, &buf, sizeof(buf) );

    rfb = _Rwrread ( fp, &in_buf, sizeof(in_buf), &out_buf,
                    sizeof(out_buf ));

    /* Continue processing. */

    _Rclose ( fp );
}

```

### Related Information

- “\_Rwrite() — Write the Next Record” on page 322
- “\_Rwrited() — Write a Record Directly” on page 324
- “\_Rwrread() — Write and Read a Record (separate buffers)”

---

## **\_Rwrread() — Write and Read a Record (separate buffers)**

### Format

```

#include <recio.h>

_RIOFB_T *_Rwrread(_RFILE *fp, void *in_buf, size_t in_buf_size,
                  void *out_buf, size_t out_buf_size);

```

**Language Level:** ILE C Extension

**Threadsafe:** No.

### Description

The `_Rwrread()` function performs a write and then a read operation on the file that is specified by *fp*. Separate buffers may be specified for the input and output data. The minimum of size and the length of the current record format determines the amount of data to be copied between the system buffer and the buffers for both the write and read parts of the operation. If *out\_buf\_size* is greater than the record length of the current format, `errno` is set to `ETRUNC` on the write part of the operation. If *in\_buf\_size* is less than the length of the current record format, `errno` is set to `ETRUNC` on the read part of the operation.

The `_Rwrread()` function is valid for display and ICF files.

### Return Value

The `_Rwrread()` function returns a pointer to the `_RIOFB_T` structure that is associated with *fp*. If the `_Rwrread()` operation is successful, the `num_bytes` field is set to the number of bytes transferred from the system buffer to *in\_buf* in the read part of the operation (move mode) or the record length of the file (locate mode).

The value of `errno` may be set to:

#### Value Meaning

##### ENOTUPD

The file is not open for update operations.

##### ETRUNC

Truncation occurred on an I/O operation.

##### EIOERROR

A non-recoverable I/O error occurred.

##### EIORECERR

A recoverable I/O error occurred.

See Table 12 on page 507 and Table 14 on page 510 for `errno` settings.

**Example that uses `_Rwrread()`**

```

#include <stdio.h>
#include <recio.h>
#include <string.h>
#include <stdlib.h>

typedef struct {
    char name[20];
    char address[25];
} format1 ;

typedef struct {
    char name[8];
    char password[10];
} format2 ;

typedef union {
    format1 fmt1;
    format2 fmt2;
} formats ;

int main(void)
{
    _RFILE *fp; /* File pointer */
    _RIOFB_T *rfb; /*Pointer to the file's feedback structure */
    formats buf, in_buf, out_buf; /* Buffers to hold data */

    /* Open the device file. */
    if (( fp = _Ropen ( "MYLIB/T1677RD2", "ar+" )) == NULL )
    {
        printf ( "Could not open file\n" );
        exit ( 1 );
    }

    _Rpgmdev ( fp,"DEVICE2" );/* Change the default program device. */
                               /* Replace with actual device name. */

    _Rformat ( fp,"FORMAT2" ); /* Set the record format for the */
                               /* display file. */

    rfb = _Rwrite ( fp, "", 0 ); /* Set up the display. */
    rfb = _Rwriterd ( fp, &buf, sizeof(buf) );

    rfb = _Rrread ( fp, &in_buf, sizeof(in_buf), &out_buf,
                   sizeof(out_buf ));

    /* Continue processing. */

    _Rclose ( fp );
}

```

### Related Information

- “\_Rwrite() — Write the Next Record” on page 322
- “\_Rwrited() — Write a Record Directly” on page 324
- “\_Rwriterd() — Write and Read a Record” on page 327

---

## scanf() — Read Data

### Format

```

#include <stdio.h>
int scanf(const char *format-string, argument-list);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. The behavior might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

## Description

The `scanf()` function reads data from the standard input stream `stdin` into the locations that is given by each entry in *argument-list*. Each *argument* must be a pointer to a variable with a type that corresponds to a type specifier in *format-string*. The *format-string* controls the interpretation of the input fields, and is a multibyte character string that begins and ends in its initial shift state.

The *format-string* can contain one or more of the following:

- White-space characters, as specified by the `isspace()` function (such as blanks and new-line characters). A white-space character causes the `scanf()` function 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-whitespace character causes the `scanf()` function to read, but not to store, a matching non-whitespace character. If the next character in `stdin` does not match, the `scanf()` function ends.
- Format specifications, introduced by the percent sign (%). A format specification causes the `scanf()` function to read and convert characters in the input into values of a specified type. The value is assigned to an argument in the argument list.

The `scanf()` function reads *format-string* from left to right. Characters outside of format specifications are expected to match the sequence of characters in `stdin`; the matched characters in `stdin` are scanned but not stored. If a character in `stdin` conflicts with *format-string*, `scanf()` ends. The conflicting character is left in `stdin` as if it had not been read.

When the first format specification is found, the value of the first input field is converted according to the format specification and stored in the location specified by the first entry in *argument-list*. The second format specification converts the second input field and stores it in the second entry in *argument-list*, and so on through the end of *format-string*.

An input field is defined as all characters up to the first white-space character (space, tab, or new line), up to the first character that cannot be converted according to the format specification, or until the field *width* is reached, whichever comes first. If there are too many arguments for the format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for the format specifications.

A format specification has the following form:



Each field of the format specification is a single character or a number signifying a particular format option. The *type* character, 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 format specification contains only the percent sign and a *type* character (for example, %s).

Each field of the format specification is discussed in detail below. If a percent sign (%) is followed by a character that has no meaning as a format control character, that character and following characters up to the next percent sign are treated as an ordinary sequence of characters; that is, a sequence of characters that must match the input. For example, to specify a percent-sign character, use %%.

The following restrictions apply to pointer printing and scanning:

- If a pointer is printed out and scanned back from the same activation group, the scanned back pointer will be compared equal to the pointer that is printed out.
- If a scanf() family function scans a pointer that was printed out by a different activation group, the scanf() family function will set the pointer to NULL.

See the *WebSphere Development Studio: ILE C/C++ Programmer's Guide* for more information about using i5/OS pointers.

An asterisk (\*) following the percent sign suppresses assignment of the next input field, which is interpreted as a field of the specified *type*. The field is scanned but not stored.

The *width* is a positive decimal integer controlling the maximum number of characters to be read from stdin. 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 size modifiers h, l, ll, L, H, D, and DD indicate the size of the receiving object. The conversion characters d, i, and n must be preceded by h if the corresponding argument is a pointer to a short int rather than a pointer to an int, by l if it is a pointer to a long int, or by ll if it is a pointer to a long long int. Similarly, the conversion characters o, u, x, and X must be preceded by h if the corresponding argument is a pointer to an unsigned short int rather than a pointer to an unsigned int, by l if it is a pointer to an unsigned long int, or by ll if it is a pointer to an unsigned long long int. The conversion characters e, E, f, F, g, and G must be preceded by l if the corresponding argument is a pointer to a double rather than a pointer to a float, by L if it is a pointer to a long double, by H if it is a pointer to a \_Decimal32, by D if it is a pointer to a \_Decimal64, or by DD if it is a pointer to a \_Decimal128. Finally, the conversion characters c, s, and [ must be preceded by l if the corresponding argument is a pointer to a wchar\_t rather than a pointer to a single-byte character type. If an h, l, L, ll, H, D, or DD appears with any other conversion character, the behavior is undefined.

The *type* characters and their meanings are in the following table:

Character	Type of Input Expected	Type of Argument
d	Signed decimal integer	Pointer to int.
o	Unsigned octal integer	Pointer to unsigned int.
x, X	Unsigned hexadecimal integer	Pointer to unsigned int.
i	Decimal, hexadecimal, or octal integer	Pointer to int.
u	Unsigned decimal integer	Pointer to unsigned int.
e, E, f, F, g, 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 floating point.

Character	Type of Input Expected	Type of Argument
D(n,p)	Packed decimal value consisting of an optional sign (+ or -); then a non-empty sequence of digits, optionally a series of one or more decimal digits possibly containing a decimal point, but not a decimal suffix. The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-whitespace character, in the expected form. It contains no characters if the input string is empty or consists entirely of white space, or if the first non-whitespace character is anything other than a sign, a digit, or a decimal point character.	Pointer to decimal(n,p). Since the internal representation of the binary coded decimal object is the same as the internal representation of the packed decimal data type, you can use the type character D(n,p).
c	Character; white-space characters that are ordinarily skipped are read when c is specified	Pointer to char large enough for input field.
s	String	Pointer to character array large enough for input field plus a ending null character (\0), which is automatically appended.
n	No input read from <i>stream</i> or buffer	Pointer to int, into which is stored the number of characters successfully read from the <i>stream</i> or buffer up to that point in the call to <code>scanf()</code> .
p	Pointer to void converted to series of characters	Pointer to void.
lc	Multibyte character constant	Pointer to <code>wchar_t</code> .
ls	Multibyte string constant	Pointer to <code>wchar_t</code> string.

To read strings not delimited by space characters, substitute a set of characters in brackets ([ ]) for the *s* (string) type character. 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 caret (^), the effect is reversed: the input field is read up to the first character that does appear in the rest of the character set.

To store a string without storing an ending null character (\0), use the specification `%ac`, where *a* is a decimal integer. In this instance, the *c* type character 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.

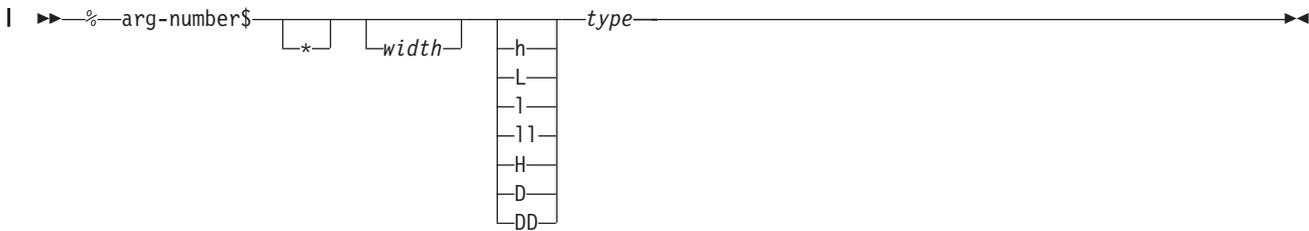
The input for a `%x` format specifier is interpreted as a hexadecimal number.

The `scanf()` function scans each input field character by character. It might stop reading a particular input field either before it reaches a space character, when the specified *width* is reached, or when the next character cannot be converted as specified. When a conflict occurs between the specification and the input character, the next input field begins at the first unread character. The conflicting character, if there was one, is considered unread and is the first character of the next input field or the first character in subsequent read operations on `stdin`.

For `%lc` and `%ls`, specifies the data that is read is a multibyte string and is converted to wide characters as if by calls to `mbtowc`.

- | For the `%e`, `%E`, `%f`, `%F`, `%g`, and `%G` format specifiers, a character sequence of INFINITY or NAN (ignoring case) is allowed and yields a value of INFINITY or Quiet Not-A-Number (NaN), respectively.

**Alternative format specification** has the following form:



As an alternative, specific entries in the argument-list may be assigned by using the format specification outlined in the diagram above. This format specification and the previous format specification may not be mixed in the same call to `scanf()`. Otherwise, unpredictable results may occur.

The `arg-number` is a positive integer constant where 1 refers to the first entry in the argument-list. `Arg-number` may not be greater than the number of entries in the argument-list, or else the results are undefined. `Arg-number` also may not be greater than `NL_ARGMAX`.

### Return Value

The `scanf()` function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is EOF for an attempt to read at end-of-file if no conversion was performed. A return value of 0 means that no fields were assigned.

### Error Conditions

If the type of the argument that is to be assigned into is different than the format specification, unpredictable results can occur. For example, reading a floating-point value, but assigning it into a variable of type `int`, is incorrect and would have unpredictable results.

If there are more arguments than format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for the format specifications.

If the format string contains an invalid format specification, and positional format specifications are being used, `errno` will be set to `EILSEQ`.

If positional format specifications are used and there are not enough arguments, `errno` will be set to `EINVAL`.

If a conversion error occurs, `errno` may be set to `ECONVERT`.

### Examples using `scanf()`

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 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);
    }
}

/***** If input is: 12 2.5 a yes, *****/
/***** then output should be similar to: *****/

Enter an integer, a real number, a character and a string :
integer = 12
real number = 2.500000
character = a
string = yes
*/

```

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);
    }
}

/***** If input is: 0x231 0xf5e 0x1 q, *****/
/***** then output should be similar to: *****/

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
*/

```

This example reads from stdin and assigns data by using the alternative positional format string.

```

#include <stdio.h>
int main(int argc, char *argv[])
{
    int i;
    char s[20];
    float f;

    scanf("%2$s %3$f %1$d",&i, s, &f);

    printf("The data read was %i\n%s\n%f\n",i,s,f);

    return 0;
}

/* If the input is : test 0.2 100
   then the output will be similar to: */
The data read was
100
test
0.20000

```

-----

This example reads in a multibyte character string into a wide Unicode string. The example can be compiled with either `LOCALETYPE(*LOCALEUCS2)` or `LOCALETYPE(*LOCALEUTF)`.

```

#include <locale.h>
#include <stdio.h>
#include <wchar.h>

void main(void)
{
    wchar_t uString[20];

    setlocale(LC_UNI_ALL, "");
    scanf("Enter a string %ls",uString);

    printf("String read was %ls\n",uString);
}

/* if the input is : ABC
   then the output will be similiar to:

   String read was ABC

*/

```

### Related Information

- “`fscanf()` — Read Formatted Data” on page 132
- “`printf()` — Print Formatted Characters” on page 229
- “`sscanf()` — Read Data” on page 355
- “`wscanf()` — Read Data Using Wide-Character Format String” on page 504
- “`fwscanf()` — Read Data from Stream Using Wide Character” on page 147
- “`swscanf()` — Read Wide Character Data” on page 407
- “`<stdio.h>`” on page 15

---

## setbuf() — Control Buffering

### Format

```
#include <stdio.h>
void setbuf(FILE *, char *buffer);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

If the operating system supports user-defined buffers, `setbuf()` controls buffering for the specified *stream*. The `setbuf()` function only works in ILE C when using the integrated file system. The *stream* pointer must refer to an open file before any I/O or repositioning has been done.

If the *buffer* argument is `NULL`, the *stream* is unbuffered. If not, the *buffer* must point to a character array of length `BUFSIZ`, which is the buffer size that is defined in the `<stdio.h>` include file. The system uses the *buffer*, which you specify, for input/output buffering instead of the default system-allocated buffer for the given *stream*. `stdout`, `stderr`, and `stdin` do not support user-defined buffers.

The `setvbuf()` function is more flexible than the `setbuf()` function.

### Return Value

There is no return value.

### Example that uses `setbuf()`

This example opens the file `setbuf.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;

    memset(buf, '\0', BUFSIZ); /* initialize buf to null characters */

    stream = fopen("setbuf.dat", "wb");

    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

- “`fclose()` — Close Stream” on page 91
- “`fflush()` — Write Buffer to File” on page 96
- “`fopen()` — Open Files” on page 109
- “`setvbuf()` — Control Buffering” on page 344
- “`<stdio.h>`” on page 15

---

## setjmp() — Preserve Environment

### Format

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `setjmp()` function saves a stack environment that can subsequently be restored by the `longjmp()` function. The `setjmp()` and `longjmp()` functions provide a way to perform a non-local goto. They are often used in signal handlers.

A call to the `setjmp()` function causes it to save the current stack environment in *env*. A subsequent call to the `longjmp()` function restores the saved environment and returns control to a point corresponding to the `setjmp()` call. The values of all variables (except register variables) available to the function receiving control contain the values they had when the `longjmp()` function was called. The values of register variables are unpredictable. Nonvolatile auto variables that are changed between calls to the `setjmp()` function and the `longjmp()` function are also unpredictable.

### Return Value

The `setjmp()` function returns the value 0 after saving the stack environment. If the `setjmp()` function returns as a result of a `longjmp()` call, it returns the *value* argument of the `longjmp()` function, or 1 if the *value* argument of the `longjmp()` function is 0. There is no error return value.

#### | Example that uses `setjmp()`

This example saves 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 the `setjmp()` function returns a 0 when it saves the environment. The program prints the message:

```
setjmp has been called
```

| The subsequent call to function `p()` causes it to call the `longjmp()` function. Control is transferred to the  
| point in the `main()` function immediately after the call to the `setjmp()` function using the environment  
| saved in the `mark` variable. This time, the condition is TRUE because -1 is specified in the second  
| parameter on the `longjmp()` function call as the return value to be placed on the stack. The example then  
| performs the statements in the block, prints the message "`longjmp()` has been called", calls the `recover()`  
| function, and leaves the program.

```

| #include <stdio.h>
| #include <setjmp.h>
| #include <stdlib.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");
|     printf("Calling function p()\n");
|     p();
|     printf("This point should never be reached\n");
| }
|
| void p(void)
| {
|     printf("Calling longjmp() from inside function p()\n");
|     longjmp(mark, -1);
|     printf("This point should never be reached\n");
| }
|
| void recover(void)
| {
|     printf("Performing function recover()\n");
| }
|
| /*****Output should be as follows: *****/
| setjmp has been called
| Calling function p()
| Calling longjmp() from inside function p()
| longjmp has been called
| Performing function recover()
| *****/

```

### Related Information

- “longjmp() — Restore Stack Environment” on page 193
- “<setjmp.h>” on page 13

---

## setlocale() — Set Locale

### Format

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

**Language Level:** ANSI

**Threadsafe:** No.

| **Locale Sensitive:** For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `setlocale()` function changes or queries variables that are defined in the `<locale.h>` include file, that indicate location. The values for *category* are listed below.

Category	Purpose
LC_ALL	Names entire locale of program.
LC_COLLATE	Affects behavior of the <code>strcoll()</code> and <code>strxfrm()</code> functions.
LC_CTYPE	Affects behavior of character handling functions.
LC_MONETARY	Affects monetary information returned by <code>localeconv()</code> and <code>nl_langinfo()</code> functions.
LC_NUMERIC	Affects the decimal-point character for the formatted input/output and string conversion functions, and the non-monetary formatting information returned by the <code>localeconv()</code> and <code>nl_langinfo()</code> functions.
LC_TIME	Affects behavior of the <code>strftime()</code> function and the time formatting information returned by the <code>nl_langinfo()</code> function.
LC_TOD	Affects the behavior of the time functions.  The category <code>LC_TOD</code> has several fields in it. The <code>TNAME</code> field is the time zone name. The <code>TZDIFF</code> field is the difference between local time and Greenwich Meridian time. If the <code>TNAME</code> field is nonblank, then the <code>TZDIFF</code> field is used when determining the values that are returned by some of the time functions. This value takes precedence over the system value, <code>QUTCOFFSET</code> .
LC_UNI_ALL*	This category causes <code>setlocale()</code> to load all of the the <code>LC_UNI_</code> categories from the locale specified. This category accepts only a locale with a UCS-2 or UTF-32 CCSID.
LC_UNI_COLLATE*	Affects behavior of the <code>wcscoll()</code> and <code>wcsxfrm()</code> functions. This category accepts only a locale with a UCS-2 or UTF-32 CCSID. <b>Note:</b> This category is not supported for UCS-2.
LC_UNI_CTYPE*	Affects the behavior of the wide character handling functions. This category accepts only a locale with a UCS-2 or UTF-32 CCSID.
LC_UNI_MESSAGES*	Affects the message formatting information returned by the <code>_WCS_nl_langinfo()</code> function. This category accepts only a locale with a UCS-2 or UTF-32 CCSID.
LC_UNI_MONETARY*	Affects the monetary information returned by the <code>wcslocaleconv()</code> and <code>_WCS_nl_langinfo()</code> functions. This category accepts only a locale with a UCS-2 or UTF-32 CCSID.
LC_UNI_NUMERIC*	Affects the decimal-point character for the wide character formatted input/output and wide character string conversion functions, and the non-monetary information returned by the <code>wcslocaleconv()</code> and <code>_WCS_nl_langinfo()</code> functions. This category accepts only a locale with a UCS-2 or UTF-32 CCSID.
LC_UNI_TIME*	Affects the behavior of the <code>wcsftime()</code> function and the time formatting information returned by the <code>_WCS_nl_langinfo()</code> functions. This category accepts only a locale with a UCS-2 or UTF-32 CCSID.
LC_UNI_TOD*	Affects the behavior of the wide character time functions. This category accepts only a locale with a UCS-2 or UTF-32 CCSID.
*	To use categories with UNI in the name, <code>LOCALETYPE(*LOCALEUCS2)</code> or <code>LOCALETYPE(*LOCALEUTF)</code> must be specified on the compilation command. If <code>LOCALETYPE(*LOCALEUCS2)</code> is used, the locale specified must be a UCS-2 locale. If <code>LOCALETYPE(*LOCALEUTF)</code> is used, the locale specified must be a UTF-32 locale.

**Note:** There are two ways of defining `setlocale()` and other locale-sensitive C functions on the System i platform. The original way to define `setlocale()` uses `*CLD` locale objects to set the locale and retrieve locale-sensitive data. The second way to define `setlocale()` uses `*LOCALE` objects to set the locale and retrieve locale-sensitive data. The original way is accessed by specifying `LOCALETYPE(*CLD)` on the compilation command. The second way is accessed by specifying `LOCALETYPE(*LOCALE)`, `LOCALETYPE(*LOCALEUCS2)`, or `LOCALETYPE(*LOCALEUTF)` on

the compilation command. For more information about the two methods of locale definition in ILE C, see the International Locale Support section in the *WebSphere Development Studio: ILE C/C++ Programmer's Guide*.

### Setlocale using \*CLD locale objects

You can set the value of *locale* to "C", "", LC\_C, LC\_C\_GERMANY, LC\_C\_FRANCE, LC\_C\_SPAIN, LC\_C\_ITALY, LC\_C\_USA or LC\_C\_UK. A *locale* value of "C" indicates the default C environment. A *locale* value of "" tells the `setlocale()` function to use the default locale for the implementation.

### Setlocale with \*LOCALE objects.

You can set the value of *locale* to "", "C", "POSIX", or the fully qualified Integrated File System path name of a \*LOCALE object enclosed in double quotes. A *locale* value of "C" or "POSIX" indicates the default C \*LOCALE object. A *locale* value of "" tells the `setlocale()` function to use the default locale for the process.

The default locale for the process is determined using the following table:

LC_ALL	<ol style="list-style-type: none"> <li>1. Check the LC_ALL environment variable<sup>1</sup>. If it is defined and not null, use the specified locale<sup>2</sup> for all POSIX locale categories. Otherwise, go to the next step.</li> <li>2. For each POSIX locale category (LC_CTYPE, LC_COLLATE, LC_TIME, LC_NUMERIC, LC_MESSAGES, LC_MONETARY, and LC_TOD), check the environment variable with the same name<sup>1</sup>. If it is defined and not null, use the locale specified<sup>2</sup>.</li> <li>3. Check the LANG environment variable<sup>1</sup>. For every locale category that was not set in the previous step, if the LANG environment variable is defined and not null, set the locale category to the specified locale<sup>2</sup>. Otherwise, set it to the default C *LOCALE object.</li> </ol>
LC_CTYPE LC_COLLATE LC_TIME LC_NUMERIC LC_MESSAGES LC_MONETARY LC_TOD	<ol style="list-style-type: none"> <li>1. Check the LC_ALL environment variable<sup>1</sup>. If it is defined and not null, use the specified locale<sup>2</sup>. Otherwise, go to the next step.</li> <li>2. Check the environment variable with the same name<sup>1</sup> as the specified locale category. If it is defined and not null, use the locale specified<sup>2</sup>. Otherwise, go to the next step.</li> <li>3. Check the LANG environment variable<sup>1</sup>. If it is defined and not null, set the locale category to the specified locale<sup>2</sup>. Otherwise, go to the next step.</li> <li>4. Set the locale category to the default C *LOCALE object.</li> </ol>

LC_UNI_ALL	<p>If your module is compiled with the LOCALETYPE(*LOCALEUCS2) option:</p> <ol style="list-style-type: none"> <li>1. Check the LC_UCS2_ALL environment variable<sup>1</sup>. If it is defined and not null, use the specified locale for all Unicode locale categories. Otherwise, go to the next step.</li> <li>2. For each Unicode locale category check the corresponding environment variable<sup>1</sup> (LC_UCS2_CTYPE, LC_UCS2_COLLATE, LC_UCS2_TIME, LC_UCS2_NUMERIC, LC_UCS2_MESSAGES, LC_UCS2_MONETARY, or LC_UCS2_TOD)<sup>3</sup>. If it is defined and not null, use the locale specified.</li> <li>3. Set the locale category to the default UCS-2 *LOCALE object.</li> </ol> <p>If your module is compiled with the LOCALETYPE(*LOCALEUTF) option:</p> <ol style="list-style-type: none"> <li>1. Check the LC_UTF_ALL environment variable<sup>1</sup>. If it is defined and not null, use the specified locale for all Unicode locale categories. Otherwise, go to the next step.</li> <li>2. For each Unicode locale category check the corresponding environment variable<sup>1</sup> (LC_UTF_CTYPE, LC_UTF_COLLATE, LC_UTF_TIME, LC_UTF_NUMERIC, LC_UTF_MESSAGES, LC_UTF_MONETARY, or LC_UTF_TOD)<sup>3</sup>. If it is defined and not null, use the locale specified.</li> <li>3. Check the LANG environment variable<sup>1</sup>. For every locale category that was not set in the previous step, if the LANG environment variable is defined and not null, set the locale category to the specified locale. Otherwise, set it to the default UTF *LOCALE object.</li> </ol>
LC_UNI_CTYPE LC_UNI_COLLATE LC_UNI_TIME LC_UNI_NUMERIC LC_UNI_MESSAGES LC_UNI_MONETARY LC_UNI_TOD	<p>If your module is compiled with the LOCALETYPE(*LOCALEUCS2) option:</p> <ol style="list-style-type: none"> <li>1. Check the environment variable corresponding to the specified locale category<sup>1</sup> (LC_UCS2_CTYPE, LC_UCS2_COLLATE, LC_UCS2_TIME, LC_UCS2_NUMERIC, LC_UCS2_MESSAGES, LC_UCS2_MONETARY, or LC_UCS2_TOD)<sup>3</sup>. If it is defined and not null, use the locale specified. Otherwise, go to the next step.</li> <li>2. Check the LC_UCS2_ALL environment variable<sup>1</sup>. If it is defined and not null, use the specified locale. Otherwise, go to the next step.</li> <li>3. Set the locale category to the default UCS-2 *LOCALE object.</li> </ol> <p>If your module is compiled with the LOCALETYPE(*LOCALEUTF) option:</p> <ol style="list-style-type: none"> <li>1. Check the environment variable corresponding to the specified locale category<sup>1</sup> (LC_UTF_CTYPE, LC_UTF_COLLATE, LC_UTF_TIME, LC_UTF_NUMERIC, LC_UTF_MESSAGES, LC_UTF_MONETARY, or LC_UTF_TOD)<sup>3</sup>. If it is defined and not null, use the locale specified. Otherwise, go to the next step.</li> <li>2. Check the LC_UTF_ALL environment variable<sup>1</sup>. If it is defined and not null, use the specified locale. Otherwise, go to the next step.</li> <li>3. Check the LANG environment variable<sup>1</sup>. If the LANG environment variable is defined and not null, set the locale category to the specified locale. Otherwise, set it to the default UTF *LOCALE object.</li> </ol>

**Note:** <sup>1</sup> The environment variables with names corresponding to locale categories are created by the user. The LANG environment variable is automatically created during job initiation when you specify a locale path name for either of the following:

- the LOCALE parameter in your user profile (see the CHGUSRPRF (Change User Profile) command information in the i5/OS Information Center).

- the QLOCALE system value (see the QLOCALE system value information in the i5/OS Information Center).

The locale environment variables are expected to contain a locale path name of the form /QSYS.LIB/<locname>.LOCALE or /QSYS.LIB/<libname>.LIB/<locname>.LOCALE. If your module is compiled with the LOCALETYPE(\*LOCALEUTF) option, the environment variable will be ignored if the <locname> portion of the path exceeds 8 characters. This restriction exists because a 2 character suffix must be appended to the locale name to get the name of the corresponding UTF locale.

**Note:** <sup>2</sup> When LOCALETYPE(\*LOCALEUTF) is specified on the compilation command, the setlocale() function appends a trailing \_8 to the LC\_ALL, LC\_CTYPE, LC\_COLLATE, LC\_TIME, LC\_NUMERIC, LC\_MESSAGES, LC\_MONETARY, LC\_TOD, and LANG environment variables. If this locale is not found, the UTF default locale object is used. For example, setlocale(LC\_ALL, "") when LANG is set to /QSYS.LIB/EN\_US.LOCALE causes setlocale() to attempt to load the locale /QSYS.LIB/EN\_US\_8.LOCALE. If the LANG environment variable is used to set one of the Unicode locale categories (LC\_UNI\_ALL, LC\_UNI\_CTYPE, LC\_UNI\_COLLATE, LC\_UNI\_TIME, LC\_UNI\_NUMERIC, LC\_UNI\_MESSAGES, LC\_UNI\_MONETARY, or LC\_UNI\_TOD), setlocale() appends a trailing \_4 to the locale name stored in the environment variable. This is an attempt to locate the corresponding UTF-32 locale. If this locale is not found, the default UTF-32 locale object is used. For example, setlocale(LC\_UNI\_TIME, "") when LANG is set to /QSYS.LIB/EN\_US.LOCALE causes setlocale() to attempt to load the locale /QSYS.LIB/EN\_US\_4.LOCALE. Locale names ending in \_4 and \_8 follow a naming convention introduced by the CRTLOCALE CL command (see the CRTLOCALE (Create Locale) command information in the i5/OS Information Center) for locales created with CCSID(\*UTF).

**Note:** <sup>3</sup> The LC\_UNI\_ALL, LC\_UNI\_COLLATE, LC\_UNI\_CTYPE, LC\_UNI\_TIME, LC\_UNI\_NUMERIC, LC\_UNI\_MESSAGES, LC\_UNI\_MONETARY, and LC\_UNI\_TOD locale category names are shared between UCS-2 and UTF. The environment variables corresponding to these categories cannot be shared, so the names of the environment variables do not exactly match the locale category names. For UCS-2 environment variable names, UNI is replaced with UCS2 (for example, LC\_UNI\_ALL locale category becomes LC\_UCS2\_ALL environment variable). For UTF environment variable names, UNI is replaced with UTF (for example, LC\_UNI\_ALL locale category becomes LC\_UTF\_ALL environment variable).

If compiled with LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF), the locale must be a pointer to a valid Unicode locale for the categories starting with LC\_UNI\_, and must not be a Unicode locale for the other categories.

## Return Value

The setlocale() function returns a pointer to a string that represents the current locale setting. If the returned string is stored, the stored string value can be used as input to the setlocale() function to restore the locale setting at any time. However, you need to copy the string to a user-defined buffer; otherwise, the string is overwritten on subsequent calls to setlocale().

**Note:** Because the string to which a successful call to setlocale() points may be overwritten by subsequent calls to the setlocale() function, you should copy the string if you plan to use it later. The exact format of the locale string is different between locale types of \*CLD, \*LOCALE, \*LOCALEUCS2, and \*LOCALEUTF.

To query the locale, give a NULL as the second parameter. For example, to query all the categories of your locale, enter the following statement:

```
char *string = setlocale(LC_ALL, NULL);
```

## Error Conditions

On error, the `setlocale()` function returns `NULL`, and the program's locale is not changed.

### Example that uses \*CLD locale objects

```
/******  
  
This example sets the locale of the program to  
LC_C_FRANCE *CLD and prints the string  
that is associated with the locale. This example must be compiled with  
the LOCALETYPE(*CLD) parameter on the compilation command.  
*  
  
*****/  
  
#include <stdio.h>  
#include <locale.h>  
  
char *string;  
  
int main(void)  
{  
    string = setlocale(LC_ALL, LC_C_FRANCE);  
    if (string != NULL)  
        printf(" %s \n",string);  
}
```

### Example that uses \*LOCALE objects

```
/******  
  
This example sets the locale of the program to be "POSIX" and prints  
the string that is associated with the locale. This example must be  
compiled with the LOCALETYPE(*LOCALE) parameter on the CRTCMOD or  
CRTBNDC command.  
  
*****/  
  
#include <stdio.h>  
#include <locale.h>  
  
char *string;  
  
int main(void)  
{  
  
    string = setlocale(LC_ALL, "POSIX");  
    if (string != NULL)  
        printf(" %s \n",string);  
}
```

### Related Information

- “`getenv()` — Search for Environment Variables” on page 154
- “`localeconv()` — Retrieve Information from the Environment” on page 181
- “`nl_langinfo()` — Retrieve Locale Information” on page 224
- “`<locale.h>`” on page 7

---

## setvbuf() — Control Buffering

### Format

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `setvbuf()` function allows control over the buffering strategy and buffer size for a specified stream. The `setvbuf()` function only works in ILE C when using the integrated file system. The stream must refer to a file that has been opened, but not read or written to.

The array pointed to by *buf* designates an area that you provide that the C library may choose to use as a buffer for the stream. A *buf* value of `NULL` indicates that no such area is supplied and that the C 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.

The *type* must be one of the following:

#### Value Meaning

##### `_IONBF`

No buffer is used.

##### `_IOFBF`

Full buffering is used for input and output. Use *buf* as the buffer and *size* as the size of the buffer.

##### `_IOLBF`

Line buffering is used. The buffer is deleted when a new-line character is written, when the buffer is full, or when input is requested.

If *type* is `_IOFBF` or `_IOLBF`, *size* is the size of the supplied buffer. If *buf* is `NULL`, the C library takes *size* as the suggested size for its own buffer. If *type* is `_IONBF`, both *buf* and *size* are ignored.

The value for *size* must be greater than 0.

### Return Value

The `setvbuf()` function returns 0 if successful. It returns nonzero if a value that is not valid was specified in the parameter list, or if the request cannot be performed.

The `setvbuf()` function has no effect on `stdout`, `stdin`, or `stderr`.

**Warning:** The array that is used as the buffer must still exist when the specified *stream* is closed. For example, if the buffer is declared within the scope of a function block, the *stream* must be closed before the function is ended and frees the storage allocated to the buffer.

### Example that uses `setvbuf()`

This example sets up a buffer of *buf* for *stream1* and specifies that input to *stream2* is to be unbuffered.

```

#include <stdio.h>

#define BUF_SIZE 1024

char buf[BUF_SIZE];
FILE *stream1, *stream2;

int main(void)
{
    stream1 = fopen("myfile1.dat", "r");
    stream2 = fopen("myfile2.dat", "r");

    /* stream1 uses a user-assigned buffer of BUF_SIZE bytes */
    if (setvbuf(stream1, buf, _IOFBF, sizeof(buf)) != 0)
        printf("Incorrect type or size of buffer\n");

    /* stream2 is unbuffered */
    if (setvbuf(stream2, NULL, _IONBF, 0) != 0)
        printf("Incorrect type or size of buffer\n");

    /* This is a program fragment and not a complete function example */
}

```

### Related Information

- “fclose() — Close Stream” on page 91
- “fflush() — Write Buffer to File” on page 96
- “fopen() — Open Files” on page 109
- “setbuf() — Control Buffering” on page 336
- “<stdio.h>” on page 15

---

## signal() — Handle Interrupt Signals

### Format

```

#include <signal.h>
void ( *signal (int sig, void(*func)(int)) )(int);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `signal()` function allows a program to choose one of several ways to handle an interrupt signal from the operating system or from the `raise()` function. If compiled with the `SYSIFCOPT(*ASYNC SIGNAL)` option, this function uses asynchronous signals. The asynchronous version of this function behaves like `sigaction()` with `SA_NODEFER` and `SA_RESETHAND` options. Asynchronous signal handlers may not call `abort()` or `exit()`. The remainder of this function description will describe synchronous signals.

The *sig* argument must be one of the macros `SIGABRT`, `SIGALL`, `SIGILL`, `SIGINT`, `SIGFPE`, `SIGIO`, `SIGOTHER`, `SIGSEGV`, `SIGTERM`, `SIGUSR1`, or `SIGUSR2`, defined in the `signal.h` include file. `SIGALL`, `SIGIO`, and `SIGOTHER` are only supported by the ILE C/C++ runtime library. The *func* argument must be one of the macros `SIG_DFL` or `SIG_IGN`, defined in the `<signal.h>` include file, or a function address.

The meaning of the values of *sig* is as follows:

### Value Meaning

**SIGABRT**

Abnormal termination

**SIGALL**

Catch-all for signals whose current handling action is SIG\_DFL.

When SYSIFCOPT(\*ASYNC SIGNAL) is specified, SIGALL is not a catch-all signal. A signal handler for SIGALL is only invoked for a user-raised SIGALL signal.

**SIGILL**

Detection of a function image that was not valid

**SIGFPE**

Arithmetic exceptions that are not masked, such as overflow, division by zero, and operations that are not valid

**SIGINT**

Interactive attention

**SIGIO**

Record file I/O error

**SIGOTHER**

ILE C signal

**SIGSEGV**

Access to memory that was not valid

**SIGTERM**

End request sent to the program

**SIGUSR1**

Intended for use by user applications. (extension to ANSI)

**SIGUSR2**

Intended for use by user applications. (extension to ANSI)

The action that is taken when the interrupt signal is received depends on the value of *func*.

**Value Meaning****SIG\_DFL**

Default handling for the signal will occur.

**SIG\_IGN**

The signal is to be ignored.

**Return Value**

A return value of SIG\_ERR indicates an error in the call to `signal()`. If successful, the call to `signal()` returns the most recent value of *func*. The value of `errno` may be set to EINVAL (the signal is not valid).

**Example that uses `signal()`**

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(SIG, StrCln) { \
    if (signal((SIG), (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;
    *buffer = malloc(size*ONE_K); /* get the size in number of K */
    if (*buffer == NULL) {
        if (raise(OUT_OF_STORAGE)) {
            perror("Could not raise user signal");
            abort();
        }
    }
    return;
}
/* This is a program fragment and not a complete function example */

```

### Related Information

- “abort() — Stop a Program” on page 36
- “atexit() — Record Program Ending Function” on page 45
- “exit() — End Program” on page 88
- “raise() — Send Signal” on page 255
- “<signal.h>” on page 13
- signal() API in the APIs topic in the i5/OS Information Center.

---

## sin() — Calculate Sine

### Format

```
#include <math.h>
double sin(double x);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `sin()` function calculates the sine of  $x$ , with  $x$  expressed in radians. If  $x$  is too large, a partial loss of significance in the result may occur.

### Return Value

The `sin()` function returns the value of the sine of  $x$ . The value of `errno` may be set to either `EDOM` or `ERANGE`.

### Example that uses `sin()`

This example computes  $y$  as the sine of  $\pi/2$ .

```
#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 should be similar to: *****/

sin( 1.570796 ) = 1.000000
*/
```

### Related Information

- “`acos()` — Calculate Arccosine” on page 38
- “`asin()` — Calculate Arcsine” on page 42
- “`atan()` – `atan2()` — Calculate Arctangent” on page 44
- “`cos()` — Calculate Cosine” on page 64
- “`cosh()` — Calculate Hyperbolic Cosine” on page 65
- “`sinh()` — Calculate Hyperbolic Sine”
- “`tan()` — Calculate Tangent” on page 409
- “`tanh()` — Calculate Hyperbolic Tangent” on page 410
- “`<math.h>`” on page 8

---

## `sinh()` — Calculate Hyperbolic Sine

### Format

```
#include <math.h>
double sinh(double x);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

## Description

The `sinh()` function calculates the hyperbolic sine of  $x$ , with  $x$  expressed in radians.

## Return Value

The `sinh()` function returns the value of the hyperbolic sine of  $x$ . If the result is too large, the `sinh()` function sets `errno` to `ERANGE` and returns the value `HUGE_VAL` (positive or negative, depending on the value of  $x$ ).

## Example that uses `sinh()`

This example computes  $y$  as the hyperbolic sine of  $\pi/2$ .

```
#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 should be similar to: *****/

sinh( 1.570796 ) = 2.301299
*/
```

## Related Information

- “`acos()` — Calculate Arccosine” on page 38
- “`asin()` — Calculate Arcsine” on page 42
- “`atan()` – `atan2()` — Calculate Arctangent” on page 44
- “`cos()` — Calculate Cosine” on page 64
- “`cosh()` — Calculate Hyperbolic Cosine” on page 65
- “`sin()` — Calculate Sine” on page 348
- “`tan()` — Calculate Tangent” on page 409
- “`tanh()` — Calculate Hyperbolic Tangent” on page 410
- “`<math.h>`” on page 8

---

## `snprintf()` — Print Formatted Data to Buffer

### Format

```
#include <stdio.h>
int snprintf(char *buffer, size_t n, const char *format-string,
             argument-list);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. The behavior might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

## Description

The `snprintf()` function formats and stores a series of characters and values in the array *buffer*. Any *argument-list* is converted and put out according to the corresponding format specification in the *format-string*. The `snprintf()` function is identical to the `sprintf()` function with the addition of the *n* argument, which indicates the maximum number of characters (including the ending null character) to be written to *buffer*.

The *format-string* consists of ordinary characters and has the same form and function as the format string for the `printf()` function.

## Return Value

The `snprintf()` function returns the number of bytes that are written in the array, not counting the ending null character.

## Example that uses `snprintf()`

This example uses `snprintf()` 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 = snprintf(buffer, 6, "%s\n", s);
    j += snprintf(buffer+j, 6, "%c\n", c);
    j += snprintf(buffer+j, 6, "%d\n", i);
    j += snprintf(buffer+j, 6, "%f\n", fp);
    printf("string:\n%s\ncharacter count = %d\n", buffer, j);
}

/***** Output should be similar to: *****/

string:
baltil
35
1.732
character count = 15      */
```

## Related Information

- “`fprintf()` — Write Formatted Data to a Stream” on page 116
- “`printf()` — Print Formatted Characters” on page 229
- “`sprintf()` — Print Formatted Data to Buffer” on page 352
- “`vsnprintf()` — Print Argument Data to Buffer” on page 435

- “<stdio.h>” on page 15

---

## printf() — Print Formatted Data to Buffer

### Format

```
#include <stdio.h>
int printf(char *buffer, const char *format-string, argument-list);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC  
| categories of the current locale. The behavior might also be affected by the LC\_UNI\_CTYPE category of  
| the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the  
| compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The printf() function formats and stores a series of characters and values in the array *buffer*. Any *argument-list* is converted and put out according to the corresponding format specification in the *format-string*.

The *format-string* consists of ordinary characters and has the same form and function as the *format-string* argument for the printf() function.

### Return Value

The printf() function returns the number of bytes that are written in the array, not counting the ending null character.

### Example that uses printf()

This example uses printf() to format and print various data.

```

#include <stdio.h>

char buffer[200];
int i, j;
double fp;
char *s = "baltimore";
char c;

int main(void)
{
    c = 'l';
    i = 35;
    fp = 1.7320508;

    /* Format and print various data */
    j = sprintf(buffer, "%s\n", s);
    j += sprintf(buffer+j, "%c\n", c);
    j += sprintf(buffer+j, "%d\n", i);
    j += sprintf(buffer+j, "%f\n", fp);
    printf("string:\n%s\ncharacter count = %d\n", buffer, j);
}

/***** Output should be similar to: *****/

string:
baltimore
l
35
1.732051

character count = 24      */

```

### Related Information

- “fprintf() — Write Formatted Data to a Stream” on page 116
- “printf() — Print Formatted Characters” on page 229
- “scanf() — Read Data” on page 355
- “swprintf() — Format and Write Wide Characters to Buffer” on page 406
- “vfprintf() — Print Argument Data to Stream” on page 425
- “vprintf() — Print Argument Data” on page 432
- “vsprintf() — Print Argument Data to Buffer” on page 436
- “<stdio.h>” on page 15

---

## sqrt() — Calculate Square Root

### Format

```

#include <math.h>
double sqrt(double x);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `sqrt()` function calculates the nonnegative value of the square root of  $x$ .

### Return Value

The `sqrt()` function returns the square root result. If  $x$  is negative, the function sets `errno` to `EDOM`, and returns 0.

### Example that uses `sqrt()`

This example computes the square root of the quantity that is 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( %lf ) = %lf\n", value, sqrt( value ));
    }
}

/***** If the input is 45, *****/
/***** then the output should be similar to: *****/

sqrt( 45.000000 ) = 6.708204
*/
```

### Related Information

- “`exp()` — Calculate Exponential Function” on page 89
- “`hypot()` — Calculate Hypotenuse” on page 168
- “`log()` — Calculate Natural Logarithm” on page 191
- “`log10()` — Calculate Base 10 Logarithm” on page 191
- “`pow()` — Compute Power” on page 228
- “`<math.h>`” on page 8

---

## **`srand()` — Set Seed for `rand()` Function**

### Format

```
#include <stdlib.h>
void srand(unsigned int seed);
```

**Language Level:** ANSI

**Threadsafe:** No.

### Description

The `srand()` function sets the starting point for producing a series of pseudo-random integers. If `srand()` is not called, the `rand()` seed is set as if `srand(1)` were called at program start. Any other value for *seed* sets the generator to a different starting point.

The `rand()` function generates the pseudo-random numbers.

### Return Value

There is no return value.

### Example that uses `srand()`

This example first calls `srand()` with a value other than 1 to initiate the random value sequence. Then the program computes five random values for the array of integers that are called **ranvals**.

```
#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 should be similar to: *****/

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

- “`rand()`, `rand_r()` — Generate Random Number” on page 256
- “`<stdlib.h>`” on page 17

---

## sscanf() — Read Data

### Format

```
#include <stdio.h>
int sscanf(const char *buffer, const char *format, argument-list);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the `LC_CTYPE` and `LC_NUMERIC`  
| categories of the current locale. The behavior might also be affected by the `LC_UNI_CTYPE` category of  
| the current locale if `LOCALETYPE(*LOCALEUCS2)` or `LOCALETYPE(*LOCALEUTF)` is specified on the  
| compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `sscanf()` function reads data from *buffer* into the locations that are given by *argument-list*. Each *argument* must be a pointer to a variable with a type that corresponds to a type specifier in the *format-string*.

## Return Value

The `sscanf()` function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned.

The return value is EOF when the end of the string is encountered before anything is converted.

## Example that uses `sscanf()`

This example uses `sscanf()` to read various data from the string *tokenstring*, and then displays that data.

```
#include <stdio.h>
#include <stddef.h>

int main(void)
{
    char    *tokenstring = "15 12 14";
    char    *string = "ABC Z";
    wchar_t ws[81];
    wchar_t wc;
    int     i;
    float   fp;
    char    s[81];
    char    c;

    /* Input various data */
    /* In the first invocation of sscanf, the format string is */
    /* "%s %c%d%f". If there were no space between %s and %c, */
    /* sscanf would read the first character following the */
    /* string, which is a blank space. */

    sscanf(tokenstring, "%s %c%d%f", s, &c, &i, &fp);
    sscanf(string, "%ls %lc", ws,&wc);

    /* Display the data */
    printf("\nstring = %s\n",s);
    printf("character = %c\n",c);
    printf("integer = %d\n",i);
    printf("floating-point number = %f\n",fp);
    printf("wide-character string = %S\n",ws);
    printf("wide-character = %C\n",wc);
}

/***** Output should be similar to: *****/

string = 15
character = 1
integer = 2
floating-point number = 14.000000
wide-character string = ABC
wide-character = Z

*****/
```

## Related Information

- “`fscanf()` — Read Formatted Data” on page 132
- “`scanf()` — Read Data” on page 330
- “`swscanf()` — Read Wide Character Data” on page 407
- “`fwscanf()` — Read Data from Stream Using Wide Character” on page 147
- “`wscanf()` — Read Data Using Wide-Character Format String” on page 504
- “`sprintf()` — Print Formatted Data to Buffer” on page 352

- “<stdio.h>” on page 15

---

## strcasemp() — Compare Strings without Case Sensitivity

### Format

```
#include <strings.h>
int strcasemp(const char *string1, const char *string2);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the  
| current locale. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation  
| command. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The strcasemp() function compares *string1* and *string2* without sensitivity to case. All alphabetic characters in *string1* and *string2* are converted to lowercase before comparison.

The strcasemp() function operates on null terminated strings. The string arguments to the function are expected to contain a null character ('\0') marking the end of the string.

### Return Value

The strcasemp() function returns a value indicating the relationship between the two strings, as follows:

Table 6. Return values of strcasemp()

Value	Meaning
Less than 0	<i>string1</i> less than <i>string2</i>
0	<i>string1</i> equivalent to <i>string2</i>
Greater than 0	<i>string1</i> greater than <i>string2</i>

### Example that uses strcasemp()

This example uses strcasemp() to compare two strings.

```
#include <stdio.h>
#include <strings.h>

int main(void)
{
    char_t *str1 = "STRING";
    char_t *str2 = "string";
    int result;

    result = strcasemp(str1, str2);

    if (result == 0)
        printf("Strings compared equal.\n");
    else if (result < 0)
        printf("\'%s\' is less than \''%s\'.\n", str1, str2);
    else
        printf("\'%s\' is greater than \''%s\'.\n", str1, str2);

    return 0;
}
```

```
/****** The output should be similar to: *****/
```

```
Strings compared equal.
```

```
*****/
```

### Related Information

- “strncasecmp() — Compare Strings without Case Sensitivity” on page 376
- “strncmp() — Compare Strings” on page 379
- “stricmp() - Compare Strings without Case Sensitivity” on page 374
- “wcsncmp() — Compare Wide-Character Strings” on page 453
- “wcsncmp() — Compare Wide-Character Strings” on page 464
- “\_\_wcsicmp() — Compare Wide Character Strings without Case Sensitivity” on page 460
- “\_\_wcsnicmp() — Compare Wide Character Strings without Case Sensitivity” on page 467
- “<strings.h>” on page 17

---

## strcat() — Concatenate Strings

### Format

```
#include <string.h>
char *strcat(char *string1, const char *string2);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `strcat()` function concatenates *string2* to *string1* and ends the resulting string with the null character.

The `strcat()` function operates on null-ended strings. The string arguments to the function should contain a null character (`\0`) that marks the end of the string. No length checking is performed. 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.

### Return Value

The `strcat()` function returns a pointer to the concatenated string (*string1*).

### Example that uses `strcat()`

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 should be similar to: *****/

buffer1 = computer program
*/

```

### Related Information

- “[strchr\(\)](#) — Search for Character”
- “[strcmp\(\)](#) — Compare Strings” on page 360
- “[strcpy\(\)](#) — Copy Strings” on page 364
- “[strcspn\(\)](#) — Find Offset of First Character Match” on page 365
- “[strncat\(\)](#) — Concatenate Strings” on page 377
- “[wcsat\(\)](#) — Concatenate Wide-Character Strings” on page 451
- “[wcsncat\(\)](#) — Concatenate Wide-Character Strings” on page 463
- “[<string.h>](#)” on page 17

---

## strchr() — Search for Character

### Format

```

#include <string.h>
char *strchr(const char *string, int c);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. For more information, see “[Understanding CCSIDs and Locales](#)” on page 524.

### Description

The `strchr()` function finds the first occurrence of a character in a string. The character `c` can be the null character (`\0`); the ending null character of `string` is included in the search.

The `strchr()` function operates on null-ended strings. The string arguments to the function should contain a null character (`\0`) that marks the end of the string.

### Return Value

The `strchr()` function returns a pointer to the first occurrence of `c` that is converted to a character in `string`. The function returns `NULL` if the specified character is not found.

## Example that uses strchr()

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 should be similar to: *****/

The first occurrence of p in 'computer program' is 'puter program'
*/
```

## Related Information

- "strcat() — Concatenate Strings" on page 358
- "strcmp() — Compare Strings"
- "strcpy() — Copy Strings" on page 364
- "strcspn() — Find Offset of First Character Match" on page 365
- "strncmp() — Compare Strings" on page 379
- "strpbrk() — Find Characters in String" on page 384
- "strrchr() — Locate Last Occurrence of Character in String" on page 389
- "strspn() — Find Offset of First Non-matching Character" on page 390
- "wcschr() — Search for Wide Character" on page 452
- "wcspn() — Find Offset of First Non-matching Wide Character" on page 474
- "<string.h>" on page 17

---

## strcmp() — Compare Strings

### Format

```
#include <string.h>
int strcmp(const char *string1, const char *string2);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `strcmp()` function compares *string1* and *string2*. The function operates on null-ended strings. The string arguments to the function should contain a null character (`\0`) that marks the end of the string.

### Return Value

The `strcmp()` function returns a value indicating the relationship between the two strings, as follows:

Value	Meaning
Less than 0	<i>string1</i> less than <i>string2</i>
0	<i>string1</i> identical to <i>string2</i>
Greater than 0	<i>string1</i> greater than <i>string2</i>

### Example that uses `strcmp()`

This example compares the two strings that are 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] );
    }
}

/***** 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

- “`strcat()` — Concatenate Strings” on page 358
- “`strchr()` — Search for Character” on page 359
- “`strcpy()` — Copy Strings” on page 364
- “`strcspn()` — Find Offset of First Character Match” on page 365
- “`strncmp()` — Compare Strings” on page 379
- “`strpbrk()` — Find Characters in String” on page 384
- “`strrchr()` — Locate Last Occurrence of Character in String” on page 389
- “`strspn()` — Find Offset of First Non-matching Character” on page 390
- “`wcschr()` — Search for Wide Character” on page 452
- “`wcspn()` — Find Offset of First Non-matching Wide Character” on page 474
- “`<string.h>`” on page 17

---

## strncmpi() - Compare Strings Without Case Sensitivity

### Format

```
#include <string.h>
int strncmpi(const char *string1, const char *string2);
```

**Note:** The `strncmpi` function is available for C++ programs. It is available for C only when the program defines the `__cplusplus__strings__` macro.

**Language Level:** Extension

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the `LC_CTYPE` category of the  
| current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

`strncmpi` compares *string1* and *string2* without sensitivity to case. All alphabetic characters in the two arguments *string1* and *string2* are converted to lowercase before the comparison.

The function operates on null-ended strings. The string arguments to the function are expected to contain a null character (`\0`) marking the end of the string.

### Return Value

`strncmpi` returns a value indicating the relationship between the two strings, as follows:

Value	Meaning
Less than 0	<i>string1</i> less than <i>string2</i>
0	<i>string1</i> equivalent to <i>string2</i>
Greater than 0	<i>string1</i> greater than <i>string2</i>

### Example that uses `strncmpi()`

This example uses `strncmpi` to compare two strings.

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    /* Compare two strings without regard to case */
    if (0 == strncmpi("hello", "HELLO"))
        printf("The strings are equivalent.\n");
    else
        printf("The strings are not equivalent.\n");
    return 0;
}
```

The output should be:

```
The strings are equivalent.
```

### Related Information:

- “`strcoll()` — Compare Strings” on page 363
- “`strcspn()` — Find Offset of First Character Match” on page 365
- “`strdup` - Duplicate String” on page 366

- “stricmp() - Compare Strings without Case Sensitivity” on page 374
- “strncmp() — Compare Strings” on page 379
- “strnicmp - Compare Substrings Without Case Sensitivity” on page 382
- “wcscmp() — Compare Wide-Character Strings” on page 453
- “wcsncmp() — Compare Wide-Character Strings” on page 464
- “strcasemp() — Compare Strings without Case Sensitivity” on page 357
- “strncasemp() — Compare Strings without Case Sensitivity” on page 376
- “<string.h>” on page 17

---

## strcoll() — Compare Strings

### Format

```
#include <string.h>
int strcoll(const char *string1, const char *string2);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Locale Sensitive:** The behavior of this function might be affected by the LC\_COLLATE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `strcoll()` function compares two strings using the collating sequence that is specified by the program’s locale.

### Return Value

The `strcoll()` function returns a value indicating the relationship between the strings, as listed below:

Value	Meaning
Less than 0	<i>string1</i> less than <i>string2</i>
0	<i>string1</i> equivalent to <i>string2</i>
Greater than 0	<i>string1</i> greater than <i>string2</i>

If `strcoll()` is unsuccessful, `errno` is changed. The value of `errno` may be set to `EINVAL` (the *string1* or *string2* arguments contain characters that are not available in the current locale).

### Example that uses `strcoll()`

This example compares the two strings that are passed to `main()` using `strcoll()`:

```

#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] );
    }
}

/***** If the input is the strings *****/
/***** "firststring" and "secondstring", *****/
/***** then the expected output is: *****/

"firststring" is less than "secondstring"
*/

```

### Related Information

- “setlocale() — Set Locale” on page 339
- “strcmp() — Compare Strings” on page 360
- “strncmp() — Compare Strings” on page 379
- “wscoll() —Language Collation String Comparison” on page 455
- “<string.h>” on page 17

---

## strcpy() — Copy Strings

### Format

```

#include <string.h>
char *strcpy(char *string1, const char *string2);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `strcpy()` function copies *string2*, including the ending null character, to the location that is specified by *string1*.

The `strcpy()` function operates on null-ended strings. The string arguments to the function should contain a null character (`\0`) that marks the end of the string. No length checking is performed. You should not use a literal string for a *string1* value, although *string2* may be a literal string.

### Return Value

The `strcpy()` function returns a pointer to the copied string (*string1*).

### Example that uses `strcpy()`

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 should be similar to: *****/

destination is originally = "And this is the destination string"
After strcpy, destination becomes "This is the source string"
*/
```

### Related Information

- “`strcat()` — Concatenate Strings” on page 358
- “`strchr()` — Search for Character” on page 359
- “`strcmp()` — Compare Strings” on page 360
- “`strcspn()` — Find Offset of First Character Match”
- “`strncpy()` — Copy Strings” on page 380
- “`strpbrk()` — Find Characters in String” on page 384
- “`strrchr()` — Locate Last Occurrence of Character in String” on page 389
- “`strspn()` — Find Offset of First Non-matching Character” on page 390
- “`wscpy()` — Copy Wide-Character Strings” on page 456
- “`wcsncpy()` — Copy Wide-Character Strings” on page 466
- “`<string.h>`” on page 17

---

## `strcspn()` — Find Offset of First Character Match

### Format

```
#include <string.h>
size_t strcspn(const char *string1, const char *string2);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the `LC_CTYPE` category of the  
| current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `strcspn()` function finds the first occurrence of a character in *string1* that belongs to the set of characters that is specified by *string2*. Null characters are not considered in the search.

The `strcspn()` function operates on null-ended strings. The string arguments to the function should contain a null character (`\0`) marking the end of the string.

### Return Value

The `strcspn()` function returns the index of the first character found. This value is equivalent to the length of the initial substring of *string1* that consists entirely of characters not in *string2*.

### Example that uses `strcspn()`

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 should be similar to: *****/

The first 10 characters in the string "This is the source string"
are not in the string "axle"
*/
```

### Related Information

- “`strcat()` — Concatenate Strings” on page 358
- “`strchr()` — Search for Character” on page 359
- “`strcmp()` — Compare Strings” on page 360
- “`strcpy()` — Copy Strings” on page 364
- “`strncmp()` — Compare Strings” on page 379
- “`strpbrk()` — Find Characters in String” on page 384
- “`strrchr()` — Locate Last Occurrence of Character in String” on page 389
- “`strspn()` — Find Offset of First Non-matching Character” on page 390
- “`<string.h>`” on page 17

---

## strdup - Duplicate String

### Format

```
#include <string.h>
char *strdup(const char *string);
```

**Note:** The `strdup` function is available for C++ programs. It is available for C only when the program defines the `__cplusplus__strings__` macro.

**Language Level:** XPG4, Extension

**Threadsafe:** Yes.

### Description

`strdup` reserves storage space for a copy of *string* by calling `malloc`. The *string* argument to this function is expected to contain a null character (`\0`) marking the end of the string. Remember to free the storage reserved with the call to `strdup`.

### Return Value

`strdup` returns a pointer to the storage space containing the copied string. If it cannot reserve storage `strdup` returns `NULL`.

### Example that uses `strdup()`

This example uses `strdup` to duplicate a string and print the copy.

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char *string = "this is a copy";
    char *newstr;
    /* Make newstr point to a duplicate of string          */
    if ((newstr = strdup(string)) != NULL)                */
        printf("The new string is: %s\n", newstr);
    return 0;
}
```

The output should be:

```
    The new string is: this is a copy
```

### Related Information:

- “`strcpy()` — Copy Strings” on page 364
- “`strncpy()` — Copy Strings” on page 380
- “`wscpy()` — Copy Wide-Character Strings” on page 456
- “`wcncpy()` — Copy Wide-Character Strings” on page 466
- “`wcspn()` — Find Offset of First Wide-Character Match” on page 457
- “`<string.h>`” on page 17

---

## **`strerror()` — Set Pointer to Runtime Error Message**

### Format

```
#include <string.h>
char *strerror(int errnum);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `strerror()` function maps the error number in *errnum* to an error message string.

### Return Value

The `strerror()` function returns a pointer to the string. It does not return a NULL value. The value of `errno` may be set to `ECONVERT` (conversion error).

### Example that uses `strerror()`

This example opens a file and prints a runtime error message if an error occurs.

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

int main(void)
{
    FILE *stream;

    if ((stream = fopen("mylib/myfile", "r")) == NULL)
        printf(" %s \n", strerror(errno));
}

/* This is a program fragment and not a complete function example */
```

### Related Information

- “`clearerr()` — Reset Error Indicators” on page 62
- “`ferror()` — Test for Read/Write Errors” on page 95
- “`perror()` — Print Error Message” on page 227
- “`<string.h>`” on page 17

---

## strfmon() — Convert Monetary Value to String

### Format

```
#include <monetary.h>
int strfmon(char *s, size_t maxsize, const char *format, argument_list);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the `LC_CTYPE` and `LC_MONETARY`  
| categories of the current locale. This function is not available when `LOCALETYPE(*CLD)` is specified on  
| the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `strfmon()` function places characters into the array pointed to by `s` as controlled by the string pointed to by `format`. No more than `maxsize` characters are placed into the array.

The character string `format` contains two types of objects: plain characters, which are copied to the output stream, and directives, each of which results in the fetching of zero or more arguments, which are converted and formatted. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are simply ignored. Only 15 significant digits are guaranteed on conversions involving double values.

A directive consists of a % character, optional conversion specifications, and an ending character that determines the directive’s behavior.

A directive consists of the following sequence:

- A % character.
- Optional flags.
- Optional field width.
- Optional left precision.
- Optional right precision.
- A required conversion character indicating the type of conversion to be performed.

Table 7. Flags

Flag	Meaning
=f	An = followed by a single character <i>f</i> which is used as the numeric fill character. By default the numeric fill character is a space character. This flag does not affect field width filling, which always uses a space character. This flag is ignored unless left precision is specified.
^	Do not use grouping characters when formatting the currency value. Default is to insert grouping characters as defined in the current locale.
+ or (	Specify the style representing positive and negative currency amounts. If + is specified, the locale's equivalent of + and - for monetary quantities will be used. If ( is specified, negative amounts are enclosed within parenthesis. Default is +.
!	Do not output the currency symbol. Default is to output the currency symbol.
-	Use left justification for double arguments. Default is right justification.

### Field Width

**w** A decimal digit string *w* specifying a minimum field width in bytes in which the result of the conversion is right-justified (or left-justified if the flag - is specified). The default is 0.

### Left Precision

**#n** A # followed by a decimal digit string *n* specifying a 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 than *n* digit positions are required, this conversion specification is ignored. Digit positions in excess of those actually required are filled with the numeric fill character (see the =f flag above).

If grouping has not been suppressed with the ^ flag, and it is defined for the current locale, grouping separators are inserted before the fill characters (if any) are added. Grouping separators are not applied to fill characters even if the fill character is a digit. 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.

### Right Precision

**.p** A period followed by a decimal digit string *p* specifies the number of digits after the radix character. If the value of the right precision *p* is 0, no radix character appears. If a right precision is not specified, a default specified by the current locale is used. The amount being formatted is rounded to the specified number of digits prior to formatting.

Table 8. Conversion Characters

Specifier	Meaning
%i	The double argument is formatted according to the locale's international currency format.
%n	The double argument is formatted according to the locale's national currency format.
%%	Is replaced by %. No argument is converted.

## Return Value

If the total number of resulting bytes including the ending null character is not more than *maxsize*, the `strfmmon()` function returns the number of bytes placed into the array pointed to by *s*, but excludes the ending null character. Otherwise, zero is returned, and the contents of the array are undefined.

The value of `errno` may be set to:

**E2BIG** Conversion stopped due to lack of space in the buffer.

## Example that uses `strfmmon()`

```
#include <stdio.h>
#include <monetary.h>
#include <locale.h>

int main(void)
{
    char string[100];
    double money = 1234.56;
    if (setlocale(LC_ALL, "/qsys.lib/en_us.locale") == NULL) {
        printf("Unable to setlocale().\n");
        exit(1);
    }

    strfmmon(string, 100, "%i", money); /* USD 1,234.56 */
    printf("%s\n", string);
    strfmmon(string, 100, "%n", money); /* $1,234.56 */
    printf("%s\n", string);
}

/*****
    The output should be similar to:
    USD 1,234.56
    $1,234.56
*****/
```

## Related Information

- “`localeconv()` — Retrieve Information from the Environment” on page 181
- “`<monetary.h>`” on page 8

---

## strftime() — Convert Date/Time to String

### Format

```
#include <time.h>
size_t strftime(char *s, size_t maxsize, const char *format,
                const struct tm *timeptr);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

- | **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE, LC\_TIME, and
- | LC\_TOD categories of the current locale. For more information, see “Understanding CCSIDs and Locales”
- | on page 524.

## Description

The `strftime()` function places bytes into the array pointed to by *s* as controlled by the string pointed to by *format*. The format string consists of zero or more conversion specifications and ordinary characters. A conversion specification consists of a % character and a terminating conversion character that determines the behavior of the conversion. All ordinary characters (including the terminating null byte, and multi-byte chars) are copied unchanged into the array. If copying takes place between objects that overlap, then the behavior is undefined. No more than *maxsize* bytes are placed in the array. The appropriate characters are determined by the values contained in the structure pointed to by *timeptr*, and by the values stored in the current locale.

Each standard conversion specification is replaced by appropriate characters as described in the following table:

Specifier	Meaning
%a	Abbreviated weekday name.
%A	Full weekday name.
%b	Abbreviated month name.
%B	Full month name.
%c	Date/Time in the format of the locale.
%C	Century number [00-99], the year divided by 100 and truncated to an integer.
%d	Day of the month [01-31].
%D	Date Format, same as %m/%d/%y.
%e	Same as %d, except single digit is preceded by a space [1-31].
%g	2 digit year portion of ISO week date [00,99].
%G	4 digit year portion of ISO week date. Can be negative.
%h	Same as %b.
%H	Hour in 24-hour format [00-23].
%I	Hour in 12-hour format [01-12].
%j	Day of the year [001-366].
%m	Month [01-12].
%M	Minute [00-59].
%n	Newline character.
%p	AM or PM string.
%r	Time in AM/PM format of the locale. If not available in the locale time format, defaults to the POSIX time AM/PM format: %I:%M:%S %p.
%R	24-hour time format without seconds, same as %H:%M.
%S	Second [00-61]. The range for seconds allows for a leap second and a double leap second.
%t	Tab character.
%T	24-hour time format with seconds, same as %H:%M:%S.
%u	Weekday [1,7]. Monday is 1 and Sunday is 7.
%U	Week number of the year [00-53]. Sunday is the first day of the week.

Specifier	Meaning
%V	ISO week number of the year [01-53]. Monday is the first day of the week. If the week containing January 1st has four or more days in the new year then it is considered week 1. Otherwise, it is the last week of the previous year, and the next year is week 1 of the new year.
%w	Weekday [0,6], Sunday is 0.
%W	Week number of the year [00-53]. Monday is the first day of the week.
%x	Date in the format of the locale.
%X	Time in the format of the locale.
%y	2 digit year [00,99].
%Y	4-digit year. Can be negative.
%z	UTC offset. Output is a string with format +HHMM or -HHMM, where + indicates east of GMT, - indicates west of GMT, HH indicates the number of hours from GMT, and MM indicates the number of minutes from GMT.
%Z	Time zone name.
%%	% character.

### Modified Conversion Specifiers

Some conversion specifiers can be modified by the E or O modifier characters to indicate that an alternate format or specification should be used rather than the one normally used by the unmodified conversion specifier. If a modified conversion specifier uses a field in the current locale that is unavailable, then the behavior will be as if the unmodified conversion specification were used. For example, if the *era* string is the empty string "", which means that the string is unavailable, then %EY would act like %Y.

Specifier	Meaning
%Ec	Date/time for current era.
%EC	Era name.
%Ex	Date for current era.
%EX	Time for current era.
%Ey	Era year. This is the offset from the base year.
%EY	Year for current era.
%Od	Day of the month using alternate digits.
%Oe	Same as %Od.
%OH	Hour in 24 hour format using alternate digits.
%OI	Hour in 12 hour format using alternate digits.
%Om	Month using alternate digits.
%OM	Minutes using alternate digits.
%OS	Seconds using alternate digits.
%Ou	Weekday using alternate digits. Monday is 1 and Sunday is 7.
%OU	Week number of the year using alternate digits. Sunday is the first day of the week.
%OV	ISO week number of the year using alternate digits. See %V for explanation of ISO week number.
%Ow	Weekday using alternate digits. Sunday is 0.
%OW	Week number of the year using alternate digits. Monday is the first day of the week.

Specifier	Meaning
%Oy	2-digit year using alternate digits.
%OZ	If the time zone name exists in the current locale, this is the same as %Z; otherwise, the abbreviated time zone name of the current job is returned.

**Note:** %C, %D, %e, %h, %n, %r, %R, %t, %T, %u, %V, and the modified conversion specifiers are not available when LOCALETYPE(\*CLD) is specified on the compilation command.

### Return Value

If the total number of resulting bytes including the terminating null byte is not more than maxsize, strftime() returns the number of bytes placed into the array pointed to by s, not including the terminating null byte. Otherwise, 0 is returned and the contents of the array are indeterminate.

If a conversion error occurs, errno may be set to ECONVERT.

### Example that uses strftime()

```
#include <stdio.h>
#include <time.h>

int main(void)
{
    char s[100];
    int rc;
    time_t temp;
    struct tm *timeptr;

    temp = time(NULL);
    timeptr = localtime(&temp);

    rc = strftime(s, sizeof(s), "Today is %A, %b %d.\nTime: %r", timeptr);
    printf("%d characters written.\n%s\n", rc, s);

    return 0;
}

/*****
    The output should be similar to:
    46 characters written
    Today is Wednesday, Oct 24.
    Time: 01:01:15 PM
*****/
```

### Related Information

- “asctime() — Convert Time to Character String” on page 39
- “asctime\_r() — Convert Time to Character String (Restartable)” on page 41
- “ctime() — Convert Time to Character String” on page 71
- “ctime64() — Convert Time to Character String” on page 73
- “ctime64\_r() — Convert Time to Character String (Restartable)” on page 76
- “ctime\_r() — Convert Time to Character String (Restartable)” on page 74
- “gmtime() — Convert Time” on page 161
- “gmtime64() — Convert Time” on page 163
- “gmtime64\_r() — Convert Time (Restartable)” on page 167
- “gmtime\_r() — Convert Time (Restartable)” on page 165
- “localtime() — Convert Time” on page 185

- | • “localtime64() — Convert Time” on page 187
- | • “localtime64\_r() — Convert Time (Restartable)” on page 189
- “localtime\_r() — Convert Time (Restartable)” on page 188
- “setlocale() — Set Locale” on page 339
- “strptime()— Convert String to Date/Time” on page 385
- “time() — Determine Current Time” on page 411
- | • “time64() — Determine Current Time” on page 412
- “<time.h>” on page 18

---

## stricmp() - Compare Strings without Case Sensitivity

### Format

```
#include <string.h>
int stricmp(const char *string1, const char *string2);
```

**Note:** The `stricmp` function is available for C++ programs. It is available for C only when the program defines the `__cplusplus__strings__` macro.

**Language Level:** Extension

**Threadsafe:** Yes.

- | **Locale Sensitive:** The behavior of this function might be affected by the `LC_CTYPE` category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

`stricmp` compares *string1* and *string2* without sensitivity to case. All alphabetic characters in the two arguments *string1* and *string2* are converted to lowercase before the comparison.

The function operates on null-ended strings. The string arguments to the function are expected to contain a null character (`\0`) marking the end of the string.

### Return Value

`stricmp` returns a value indicating the relationship between the two strings, as follows:

Value	Meaning
Less than 0	<i>string1</i> less than <i>string2</i>
0	<i>string1</i> equivalent to <i>string2</i>
Greater than 0	<i>string1</i> greater than <i>string2</i>

### Example that uses `stricmp()`

This example uses `stricmp` to compare two strings.

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    /* Compare two strings as lowercase */
    if (0 == stricmp("hello", "HELLO"))
        printf("The strings are equivalent.\n");
}
```

```

else
    printf("The strings are not equivalent.\n");
return 0;
}

```

The output should be:

```
The strings are equivalent.
```

#### Related Information:

- “strcmpi() - Compare Strings Without Case Sensitivity” on page 362
- “strcoll() — Compare Strings” on page 363
- “strcspn() — Find Offset of First Character Match” on page 365
- “strdup - Duplicate String” on page 366
- “strncmp() — Compare Strings” on page 379
- “strcasemp() — Compare Strings without Case Sensitivity” on page 357
- “strncasemp() — Compare Strings without Case Sensitivity” on page 376
- “strnicmp - Compare Substrings Without Case Sensitivity” on page 382
- “wcsmp() — Compare Wide-Character Strings” on page 453
- “wcsncmp() — Compare Wide-Character Strings” on page 464
- “<string.h>” on page 17

---

## strlen() — Determine String Length

### Format

```
#include <string.h>
size_t strlen(const char *string);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `strlen()` function determines the length of *string* excluding the ending null character.

### Return Value

The `strlen()` function returns the length of *string*.

### Example that uses `strlen()`

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] ) );
}
/***** 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

- “mblen() — Determine Length of a Multibyte Character” on page 197
- “strncat() — Concatenate Strings” on page 377
- “strncmp() — Compare Strings” on page 379
- “strncpy() — Copy Strings” on page 380
- “wcslen() — Calculate Length of Wide-Character String” on page 461
- “<string.h>” on page 17

---

## strncasecmp() — Compare Strings without Case Sensitivity

### Format

```

#include <strings.h>
int strncasecmp(const char *string1, const char *string2, size_t count);

```

**Language Level:** XPG4

**Threadsafe:** Yes.

**Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The strncasecmp() function compares up to *count* characters of *string1* and *string2* without sensitivity to case. All alphabetic characters in *string1* and *string2* are converted to lowercase before comparison.

The strncasecmp() function operates on null terminated strings. The string arguments to the function are expected to contain a null character ('\0') marking the end of the string.

### Return Value

The strncasecmp() function returns a value indicating the relationship between the two strings, as follows:

Table 9. Return values of strncasecmp()

Value	Meaning
Less than 0	<i>string1</i> less than <i>string2</i>
0	<i>string1</i> equivalent to <i>string2</i>

Table 9. Return values of `strncasecmp()` (continued)

Greater than 0	<i>string1</i> greater than <i>string2</i>
----------------	--

### Example that uses `strncasecmp()`

This example uses `strncasecmp()` to compare two strings.

```
#include <stdio.h>
#include <strings.h>

int main(void)
{
    char_t *str1 = "STRING ONE";
    char_t *str2 = "string TWO";
    int result;

    result = strncasecmp(str1, str2, 6);

    if (result == 0)
        printf("Strings compared equal.\n");
    else if (result < 0)
        printf("\'%s\' is less than \''%s\'".\n", str1, str2);
    else
        printf("\'%s\' is greater than \''%s\'".\n", str1, str2);

    return 0;
}

/***** The output should be similar to: *****/

Strings compared equal.

*****/
```

### Related Information

- “`strcasecmp()` — Compare Strings without Case Sensitivity” on page 357
- “`strncmp()` — Compare Strings” on page 379
- “`stricmp()` - Compare Strings without Case Sensitivity” on page 374
- “`wscmp()` — Compare Wide-Character Strings” on page 453
- “`wcsncmp()` — Compare Wide-Character Strings” on page 464
- “`__wcsicmp()` — Compare Wide Character Strings without Case Sensitivity” on page 460
- “`__wcsnicmp()` — Compare Wide Character Strings without Case Sensitivity” on page 467
- “`<strings.h>`” on page 17

---

## `strncat()` — Concatenate Strings

### Format

```
#include <string.h>
char *strncat(char *string1, const char *string2, size_t count);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `strncat()` function appends the first *count* characters of *string2* to *string1* and ends the resulting string with a null character (`\0`). If *count* is greater than the length of *string2*, the length of *string2* is used in place of *count*.

The `strncat()` function operates on null-ended strings. The string argument to the function should contain a null character (`\0`) marking the end of the string.

### Return Value

The `strncat()` function returns a pointer to the joined string (*string1*).

### Example that uses `strncat()`

This example demonstrates the difference between `strcat()` and `strncat()`. The `strcat()` function 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 should be similar to: *****/

strcat : buffer1 = "computer program"
strncat: buffer1 = "computer pr"
*/
```

### Related Information

- “`strcat()` — Concatenate Strings” on page 358
- “`strncmp()` — Compare Strings” on page 379
- “`strncpy()` — Copy Strings” on page 380
- “`strpbrk()` — Find Characters in String” on page 384
- “`strrchr()` — Locate Last Occurrence of Character in String” on page 389
- “`strspn()` — Find Offset of First Non-matching Character” on page 390
- “`wscat()` — Concatenate Wide-Character Strings” on page 451
- “`wcsncat()` — Concatenate Wide-Character Strings” on page 463
- “`<string.h>`” on page 17

---

## strncmp() — Compare Strings

### Format

```
#include <string.h>
int strncmp(const char *string1, const char *string2, size_t count);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `strncmp()` function compares *string1* and *string2* to the maximum of *count*.

### Return Value

The `strncmp()` function returns a value indicating the relationship between the strings, as follows:

Value	Meaning
Less than 0	<i>string1</i> less than <i>string2</i>
0	<i>string1</i> equivalent to <i>string2</i>
Greater than 0	<i>string1</i> greater than <i>string2</i>

### Example that uses `strncmp()`

This example demonstrates the difference between the `strcmp()` function and the `strncmp()` function.

```

#include <stdio.h>
#include <string.h>

#define SIZE 10

int main(void)
{
    int result;
    int index = 3;
    char buffer1[SIZE] = "abcdefg";
    char buffer2[SIZE] = "abcfg";
    void print_result( int, char *, char * );

    result = strcmp( buffer1, buffer2 );
    printf( "Comparison of each character\n" );
    printf( "  strcmp: " );
    print_result( result, buffer1, buffer2 );

    result = strncmp( buffer1, buffer2, index);
    printf( "\nComparison of only the first %i characters\n", index );
    printf( "  strncmp: " );
    print_result( result, buffer1, buffer2 );
}

void print_result( int res, char * p_buffer1, char * p_buffer2 )
{
    if ( res == 0 )
        printf( "\"%s\" is identical to \"%s\"\n", 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 should be similar to: *****/

Comparison of each character
  strcmp: "abcdefg" is less than "abcfg"

Comparison of only the first 3 characters
  strncmp: "abcdefg" is identical to "abcfg"
*/

```

## Related Information

- “strcmp() — Compare Strings” on page 360
- “strcspn() — Find Offset of First Character Match” on page 365
- “strncat() — Concatenate Strings” on page 377
- “strncpy() — Copy Strings”
- “strpbrk() — Find Characters in String” on page 384
- “strrchr() — Locate Last Occurrence of Character in String” on page 389
- “strspn() — Find Offset of First Non-matching Character” on page 390
- “wscmp() — Compare Wide-Character Strings” on page 453
- “wcsncmp() — Compare Wide-Character Strings” on page 464
- “<string.h>” on page 17
- “\_\_wcsicmp() — Compare Wide Character Strings without Case Sensitivity” on page 460
- “\_\_wcsnicmp() — Compare Wide Character Strings without Case Sensitivity” on page 467

---

## strncpy() — Copy Strings

### Format

```
#include <string.h>
char *strncpy(char *string1, const char *string2, size_t count);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `strncpy()` function copies *count* characters of *string2* to *string1*. If *count* is less than or equal to the length of *string2*, a null character (`\0`) is *not* appended to the copied string. If *count* is greater than the length of *string2*, the *string1* result is padded with null characters (`\0`) up to length *count*.

### Return Value

The `strncpy()` function returns a pointer to *string1*.

### Example that uses `strncpy()`

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 should be similar to: *****/

destination is originally = 'abcdefg'
After strcpy, destination becomes '123456789'

destination1 is originally = 'abcdefg'
After strncpy, destination1 becomes '12345fg'
*/
```

### Related Information

- “`strcpy()` — Copy Strings” on page 364
- “`strcspn()` — Find Offset of First Character Match” on page 365
- “`strncat()` — Concatenate Strings” on page 377
- “`strncmp()` — Compare Strings” on page 379

- “strpbrk() — Find Characters in String” on page 384
- “strrchr() — Locate Last Occurrence of Character in String” on page 389
- “strspn() — Find Offset of First Non-matching Character” on page 390
- “<string.h>” on page 17

---

## strnicmp - Compare Substrings Without Case Sensitivity

### Format

```
#include <string.h>
int strnicmp(const char *string1, const char *string2, int n);
```

**Note:** The `strnset` and `strset` functions are available for C++ programs. They are available for C only when the program defines the `__cplusplus__strings__` macro.

**Language Level:** Extension

**Threadsafe:** Yes.

**Locale Sensitive:** The behavior of this function might be affected by the `LC_CTYPE` category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

`strnicmp` compares, at most, the first *n* characters of *string1* and *string2* without sensitivity to case.

The function operates on null terminated strings. The string arguments to the function are expected to contain a null character (`\0`) marking the end of the string.

### Return Value

`strnicmp` returns a value indicating the relationship between the substrings, as follows:

Value	Meaning
Less than 0	<i>substring1</i> less than <i>substring2</i>
0	<i>substring1</i> equivalent to <i>substring2</i>
Greater than 0	<i>substring1</i> greater than <i>substring2</i>

### Example that uses `strnicmp()`

This example uses `strnicmp` to compare two strings.

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char *str1 = "THIS IS THE FIRST STRING";
    char *str2 = "This is the second string";
    int numresult;
    /* Compare the first 11 characters of str1 and str2
       without regard to case */
    numresult = strnicmp(str1, str2, 11);
    if (numresult < 0)
        printf("String 1 is less than string2.\n");
    else
        if (numresult > 0)
            printf("String 1 is greater than string2.\n");
```

```

        else
            printf("The two strings are equivalent.\n");
        return 0;
    }

```

The output should be:

```
The two strings are equivalent.
```

### Related Information:

- “strcmp() — Compare Strings” on page 360
- “strncmpi() - Compare Strings Without Case Sensitivity” on page 362
- “stricmp() - Compare Strings without Case Sensitivity” on page 374
- “strncmp() — Compare Strings” on page 379
- “wcscmp() — Compare Wide-Character Strings” on page 453
- “wcsncmp() — Compare Wide-Character Strings” on page 464
- “<string.h>” on page 17

## strnset - strset - Set Characters in String

### Format

```

#include <string.h>
char *strnset(char *string, int c, size_t n);
char *strset(char *string, int c);

```

**Note:** The `strnset` and `strset` functions are available for C++ programs. They are available for C only when the program defines the `__cplusplus_strings__` macro.

**Language Level:** Extension

**Threadsafe:** Yes.

### Description

`strnset` sets, at most, the first  $n$  characters of *string* to  $c$  (converted to a char). If  $n$  is greater than the length of *string*, the length of *string* is used in place of  $n$ . `strset` sets all characters of *string*, except the ending null character (`\0`), to  $c$  (converted to a char). For both functions, the string is a null-terminated string.

### Return Value

Both `strset` and `strnset` return a pointer to the altered string. There is no error return value.

### Example that uses `strnset()` and `strset()`

In this example, `strnset` sets not more than four characters of a string to the character 'x'. Then the `strset` function changes any non-null characters of the string to the character 'k'.

```

#include <stdio.h>
#include <string.h>
int main(void)
{
    char str[] = "abcdefghi";
    printf("This is the string: %s\n", str);
    printf("This is the string after strnset: %s\n", strnset((char*)str, 'x', 4));
    printf("This is the string after strset: %s\n", strset((char*)str, 'k'));
    return 0;
}

```

The output should be:

```
This is the string: abcdefghi
This is the string after strnset: xxxxefghi
This is the string after strset: kkkkkkkkk
```

#### Related Information:

- “strchr() — Search for Character” on page 359
- “strpbrk() — Find Characters in String”
- “wcschr() — Search for Wide Character” on page 452
- “wcpbrk() — Locate Wide Characters in String” on page 468
- “<string.h>” on page 17

---

## strpbrk() — Find Characters in String

### Format

```
#include <string.h>
char *strpbrk(const char *string1, const char *string2);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the  
| current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The strpbrk() function locates the first occurrence in the string pointed to by *string1* of any character from the string pointed to by *string2*.

### Return Value

The strpbrk() function returns a pointer to the character. If *string1* and *string2* have no characters in common, a NULL pointer is returned.

### Example that uses strpbrk()

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 should be similar to: \*\*\*\*\*/

```
The first occurrence of any of the characters "ab" in "The Blue Danube"
is "anube"
*/
```

## Related Information

- “strchr() — Search for Character” on page 359
- “strcmp() — Compare Strings” on page 360
- “strcspn() — Find Offset of First Character Match” on page 365
- “strncmp() — Compare Strings” on page 379
- “strrchr() — Locate Last Occurrence of Character in String” on page 389
- “strspn() — Find Offset of First Non-matching Character” on page 390
- “wcschr() — Search for Wide Character” on page 452
- “wcscspn() — Find Offset of First Wide-Character Match” on page 457
- “wcsspbrk() — Locate Wide Characters in String” on page 468
- “wcsrchr() — Locate Last Occurrence of Wide Character in String” on page 471
- “wscswcs() — Locate Wide-Character Substring” on page 488
- “<string.h>” on page 17

---

## strptime()— Convert String to Date/Time

### Format

```
#include <time.h>
char *strptime(const char *buf, const char *format, struct tm *tm);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE, LC\_TIME, and  
| LC\_TOD categories of the current locale. This function is not available when LOCALETYPE(\*CLD) is  
| specified on the compilation command. For more information, see “Understanding CCSIDs and Locales”  
| on page 524.

### Description

The `strptime()` function 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 *format*.

The *format* contains zero or more directives. A directive contains either an ordinary character (not % or a white space), or a conversion specification. Each conversion specification is composed of a % character followed by one or more conversion characters, which specify the replacement required. There must be a white space or other delimiter in both *buf* and *format* to be guaranteed that the function will behave as expected. There must be a delimiter between two string-to-number conversions, or the first number conversion may convert characters that belong to the second conversion specifier.

Any whitespace (as specified by `isspace()`) encountered before a directive is scanned in either the format string or the input string will be ignored. A directive that is an ordinary character must exactly match the next scanned character in the input string. Case is relevant when matching ordinary character directives. If the ordinary character directive in the format string does not match the character in the input string, `strptime` is not successful. No more characters will be scanned.

Any other conversion specification is matched by scanning characters in the input string until a character that is not a possible character for that specification is found or until no more characters can be scanned. If the specification was string-to-number, the possible character range is +,- or a character specified by `isdigit()`. Number specifiers do not require leading zeros. If the specification needs to match a field in the current locale, scanning is repeated until a match is found. Case is ignored when matching fields in

the locale. If a match is found, the structure pointed to by *tm* will be updated with the corresponding locale information. If no match is found, *strptime* is not successful. No more characters will be scanned.

Missing fields in the *tm* structure may be filled in by *strftime* if given enough information. For example, if a date is given, *tm\_yday* can be calculated.

Each standard conversion specification is replaced by appropriate characters as described in the following table:

Specifier	Meaning
%a	Name of day of the week, can be either the full name or an abbreviation.
%A	Same as %a.
%b	Month name, can be either the full name or an abbreviation.
%B	Same as %b.
%c	Date/time, in the format of the locale.
%C	Century number [00–99]. Calculates the year if a two-digit year is used.
%d	Day of the month [1–31].
%D	Date format, same as %m/%d/%y.
%e	Same as %d.
%g	2 digit year portion of ISO week date [00–99].
%G	4 digit year portion of ISO week date. Can be negative.
%h	Same as %b.
%H	Hour in 24-hour format [0–23].
%I	Hour in 12-hour format [1–12].
%j	Day of the year [1–366].
%m	Month [1–12].
%M	Minute [0–59].
%n	Skip all whitespaces until a newline character is found.
%p	AM or PM string, used for calculating the hour if 12-hour format is used.
%r	Time in AM/PM format of the locale. If not available in the locale time format, defaults to the POSIX time AM/PM format: %I:%M:%S %p.
%R	24-hour time format without seconds, same as %H:%M.
%S	Second [00–61]. The range for seconds allows for a leap second and a double leap second.
%t	Skip all whitespaces until a tab character is found.
%T	24 hour time format with seconds, same as %H:%M:%S .
%u	Weekday [1–7]. Monday is 1 and Sunday is 7.
%U	Week number of the year [0–53], Sunday is the first day of the week. Used in calculating the day of the year.
%V	ISO week number of the year [1–53]. Monday is the first day of the week. If the week containing January 1st has four or more days in the new year, it is considered week 1. Otherwise, it is the last week of the previous year, and the next week is week 1 of the new year. Used in calculating the day of the year.
%w	Weekday [0 –6]. Sunday is 0.
%W	Week number of the year [0–53]. Monday is the first day of the week. Used in calculating the day of the year.
%x	Date in the format of the locale.

Specifier	Meaning
%X	Time in the format of the locale.
%y	2-digit year [0-99].
%Y	4-digit year. Can be negative.
%z	UTC offset. Output is a string with format +HHMM or -HHMM, where + indicates east of GMT, - indicates west of GMT, HH indicates the number of hours from GMT, and MM indicates the number of minutes from GMT.
%Z	Time zone name.
%%	% character.

### Modified Conversion Specifiers

Some conversion specifiers can be modified by the E or O modifier characters to indicate that an alternate format or specification should be used. If a modified conversion specifier uses a field in the current locale that is unavailable, then the behavior will be as if the unmodified conversion specification were used. For example, if the *era* string is the empty string "", which means that *era* is unavailable, then %EY would act like %Y.

Specifier	Meaning
%Ec	Date/time for current era.
%EC	Era name.
%Ex	Date for current era.
%EX	Time for current era.
%Ey	Era year. This is the offset from the base year.
%EY	Year for the current era.
%Od	Day of the month using alternate digits.
%Oe	Same as %Od.
%OH	Hour in 24-hour format using alternate digits.
%OI	Hour in 12-hour format using alternate digits.
%Om	Month using alternate digits.
%OM	Minutes using alternate digits.
%OS	Seconds using alternate digits.
%Ou	Day of the week using alternate digits. Monday is 1 and Sunday is 7.
%OU	Week number of the year using alternate digits. Sunday is the first day of the week.
%OV	ISO week number of the year using alternate digits. See %V for explanation of ISO week number.
%Ow	Weekday using alternate digit. Sunday is 0 and Saturday is 6.
%OW	Week number of the year using alternate digits. Monday is the first day of the week.
%Oy	2-digit year using alternate digits.
%OZ	Abbreviated time zone name.

### Return Value

On successful completion, the `strptime()` function returns a pointer to the character following the last character parsed. Otherwise, a null pointer is returned. The value of `errno` may be set to **ECONVERT** (conversion error).

### Example that uses `strptime()`

```
#include <stdio.h>
#include <locale.h>
#include <time.h>

int main(void)
{
    char buf[100];
    time_t t;
    struct tm *timeptr,result;

    setlocale(LC_ALL,"/QSYS.LIB/EN_US.LOCALE");
    t = time(NULL);
    timeptr = localtime(&t);
    strftime(buf,sizeof(buf), "%a %m/%d/%Y %r", timeptr);

    if(strptime(buf, "%a %m/%d/%Y %r",&result) == NULL)
        printf("\nstrptime failed\n");
    else
    {
        printf("tm_hour:  %d\n",result.tm_hour);
        printf("tm_min:   %d\n",result.tm_min);
        printf("tm_sec:   %d\n",result.tm_sec);
        printf("tm_mon:   %d\n",result.tm_mon);
        printf("tm_mday:  %d\n",result.tm_mday);
        printf("tm_year:  %d\n",result.tm_year);
        printf("tm_yday:  %d\n",result.tm_yday);
        printf("tm_wday:  %d\n",result.tm_wday);
    }

    return 0;
}

/*****
The output should be similar to:
Tue 10/30/2001 10:59:10 AM
tm_hour:  10
tm_min:   59
tm_sec:   10
tm_mon:   9
tm_mday:  30
tm_year:  101
tm_yday:  302
tm_wday:  2
*****/
```

### Related Information

- “`asctime()` — Convert Time to Character String” on page 39
- “`asctime_r()` — Convert Time to Character String (Restartable)” on page 41
- “`ctime()` — Convert Time to Character String” on page 71
- | • “`ctime64()` — Convert Time to Character String” on page 73
- | • “`ctime64_r()` — Convert Time to Character String (Restartable)” on page 76
- “`ctime_r()` — Convert Time to Character String (Restartable)” on page 74
- “`gmtime()` — Convert Time” on page 161
- | • “`gmtime64()` — Convert Time” on page 163
- | • “`gmtime64_r()` — Convert Time (Restartable)” on page 167
- “`gmtime_r()` — Convert Time (Restartable)” on page 165
- “`localtime()` — Convert Time” on page 185
- | • “`localtime64()` — Convert Time” on page 187
- | • “`localtime64_r()` — Convert Time (Restartable)” on page 189

- “localtime\_r() — Convert Time (Restartable)” on page 188
- “setlocale() — Set Locale” on page 339
- “strftime() — Convert Date/Time to String” on page 370
- “time() — Determine Current Time” on page 411
- | • “time64() — Determine Current Time” on page 412
- “<time.h>” on page 18
- “wcsptime()— Convert Wide Character String to Date/Time” on page 469

---

## strrchr() — Locate Last Occurrence of Character in String

### Format

```
#include <string.h>
char *strrchr(const char *string, int c);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

- | **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `strrchr()` function finds the last occurrence of `c` (converted to a character) in `string`. The ending null character is considered part of the `string`.

### Return Value

The `strrchr()` function returns a pointer to the last occurrence of `c` in `string`. If the given character is not found, a NULL pointer is returned.

### Example that uses `strrchr()`

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';

    /* This illustrates strchr */
    ptr = strchr( buf, ch );
    printf( "The first occurrence of %c in '%s' is '%s'\n", ch, buf, ptr );

    /* This illustrates strrchr */
    ptr = strrchr( buf, ch );
    printf( "The last occurrence of %c in '%s' is '%s'\n", ch, buf, ptr );
}

/***** Output should be similar to: *****/

The first occurrence of p in 'computer program' is 'puter program'
The last occurrence of p in 'computer program' is 'program'
*/

```

### Related Information

- “strchr() — Search for Character” on page 359
- “strcmp() — Compare Strings” on page 360
- “strcspn() — Find Offset of First Character Match” on page 365
- “strncmp() — Compare Strings” on page 379
- “strpbrk() — Find Characters in String” on page 384
- “strspn() — Find Offset of First Non-matching Character”
- “<string.h>” on page 17

---

## strspn() — Find Offset of First Non-matching Character

### Format

```

#include <string.h>
size_t strspn(const char *string1, const char *string2);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `strspn()` function finds the first occurrence of a character in *string1* that is not contained in the set of characters that is specified by *string2*. The null character (`\0`) that ends *string2* is not considered in the matching process.

### Return Value

The `strspn()` function returns the index of the first character found. This value is equal to the length of the initial substring of *string1* that consists entirely of characters from *string2*. If *string1* begins with a character not in *string2*, the `strspn()` function returns 0. If all the characters in *string1* are found in *string2*, the length of *string1* is returned.

### Example that uses `strspn()`

This example finds the first occurrence in the array *string* of a character that is not an a, b, or c. Because the string in this example is *cabbage*, the `strspn()` function 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 should be similar to: *****/

The first 5 characters of "cabbage" are found in "abc"
*/
```

### Related Information

- “`strcat()` — Concatenate Strings” on page 358
- “`strchr()` — Search for Character” on page 359
- “`strcmp()` — Compare Strings” on page 360
- “`strcpy()` — Copy Strings” on page 364
- “`strcspn()` — Find Offset of First Character Match” on page 365
- “`strpbrk()` — Find Characters in String” on page 384
- “`strrchr()` — Locate Last Occurrence of Character in String” on page 389
- “`wcschr()` — Search for Wide Character” on page 452
- “`wcscspn()` — Find Offset of First Wide-Character Match” on page 457
- “`wcspbrk()` — Locate Wide Characters in String” on page 468
- “`wcsspn()` — Find Offset of First Non-matching Wide Character” on page 474
- “`wcswcs()` — Locate Wide-Character Substring” on page 488
- “`wcsrchr()` — Locate Last Occurrence of Wide Character in String” on page 471
- “`<string.h>`” on page 17

---

## **strstr() — Locate Substring**

### Format

```
#include <string.h>
char *strstr(const char *string1, const char *string2);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

## Description

The `strstr()` function finds the first occurrence of *string2* in *string1*. The function ignores the null character (`\0`) that ends *string2* in the matching process.

## Return Value

The `strstr()` function returns a pointer to the beginning of the first occurrence of *string2* in *string1*. If *string2* does not appear in *string1*, the `strstr()` function returns `NULL`. If *string2* points to a string with zero length, the `strstr()` function returns *string1*.

## Example that uses `strstr()`

This example locates the string "haystack" in the string "needle in a haystack".

```
#include <string.h>
#include <stdio.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 should be similar to: *****/

haystack
*/
```

## Related Information

- “`strchr()` — Search for Character” on page 359
- “`strcmp()` — Compare Strings” on page 360
- “`strcspn()` — Find Offset of First Character Match” on page 365
- “`strncmp()` — Compare Strings” on page 379
- “`strpbrk()` — Find Characters in String” on page 384
- “`strrchr()` — Locate Last Occurrence of Character in String” on page 389
- “`strspn()` — Find Offset of First Non-matching Character” on page 390
- “`wcschr()` — Search for Wide Character” on page 452
- “`wcscspn()` — Find Offset of First Wide-Character Match” on page 457
- “`wcspbrk()` — Locate Wide Characters in String” on page 468
- “`wcsrchr()` — Locate Last Occurrence of Wide Character in String” on page 471
- “`wcsspn()` — Find Offset of First Non-matching Wide Character” on page 474
- “`wcswcs()` — Locate Wide-Character Substring” on page 488
- “`<string.h>`” on page 17

---

## | **strtod() — strtodf() — strtold — Convert Character String to Double, Float, and Long Double**

### | Format

```

| #include <stdlib.h>
| double strtod(const char *nptr, char **endptr);
| float strtof(const char *nptr, char **endptr);
| long double strtold(const char *nptr, char **endptr);

```

| **Language Level:** ANSI

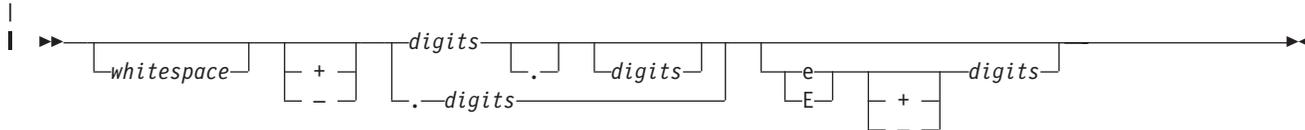
| **Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of these functions might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Description**

| The strtod(), strtof(), and strtold() functions convert a character string to a double, float, or long double value. The parameter *nptr* points to a sequence of characters that can be interpreted as a numeric binary floating-point value. These functions stop reading the string at the first character that is not recognized as part of a number. This character can be the null character at the end of the string.

| The strtod(), strtof(), and strtold() functions expect *nptr* to point to a string with the following form:



| The first character that does not fit this form stops the scan. In addition, a sequence of INFINITY or NAN (ignoring case) is allowed.

| **Return Value**

| The strtod(), strtof(), and strtold() functions return the value of the floating-point number, except when the representation causes an underflow or overflow. For an overflow, strtof() returns HUGE\_VALF or -HUGE\_VALF; strtod() and strtold() return HUGE\_VAL or -HUGE\_VAL. For an underflow, all functions return 0.

| In both cases, errno is set to ERANGE. If the string pointed to by *nptr* does not have the expected form, no conversion is performed and the value of *nptr* is stored in the object pointed to by *endptr*, provided that *endptr* is not a NULL pointer.

| The strtod(), strtof(), and strtold() functions do not fail if a character other than a digit follows an E or e that is read as an exponent. For example, 100elf is converted to the floating-point value 100.0.

| A character sequence of INFINITY (ignoring case) yields a value of INFINITY. A character value of NAN yields a Quiet Not-A-Number (NAN) value.

| **Example that uses strtod(), strtof(), and strtold()**

| This example converts the strings to double, float, and long double values. It prints the converted values and the substring that stopped the conversion.

```

| #include <stdlib.h>
| #include <stdio.h>
|
| int main(void)
| {
|     char *string, *stopstring;
|     double x;
|     float f;
|     long double ld;
|
|     string = "3.1415926This stopped it";
|     f = strttof(string, &stopstring);
|     printf("string = %s\n", string);
|     printf("  strttof = %f\n", f);
|     printf("Stopped scan at \"%s\"\n\n", stopstring);
|
|     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);
|
|     string = "3.1415926This stopped it";
|     ld = strtold(string, &stopstring);
|     printf("string = %s\n", string);
|     printf("  strtold = %lf\n", ld);
|     printf("  Stopped scan at %s\n\n", stopstring);
|     string = "100ergs";
|     ld = strtold(string, &stopstring);
|     printf("string = \"%s\"\n", string);
|     printf("  strtold = %lf\n", ld);
|     printf("  Stopped scan at \"%s\"\n\n", stopstring);
| }
|
| /***** Output should be similar to: *****/
| string = 3.1415926This stopped it
|   strttof = 3.141593
|   Stopped scan at This stopped it
|
| string = 100ergs
|   strttof = 100.000000
|   Stopped scan at "erg"
|
| string = 3.1415926This stopped it
|   strtod = 3.141593
|   Stopped scan at This stopped it
|
| string = 100ergs
|   strtod = 100.000000
|   Stopped scan at "erg"
|
| string = 3.1415926This stopped it
|   strtold = 3.141593
|   Stopped scan at This stopped it
|
| string = 100ergs
|   strtold = 100.000000
|   Stopped scan at "erg"
|
| */

```

## | Related Information

- | • “atof() — Convert Character String to Float” on page 46
- | • “atoi() — Convert Character String to Integer” on page 48
- | • “atol() — atoll() — Convert Character String to Long or Long Long Integer” on page 49
- | • “strtod32() — strtod64() — strtod128() — Convert Character String to Decimal Floating-Point”
- | • “strtol() — strtoll() — Convert Character String to Long and Long Long Integer” on page 400
- | • “strtoul() — strtoull() — Convert Character String to Unsigned Long and Unsigned Long Long Integer” on page 403
- | • “wcstod() — Convert Wide-Character String to Double” on page 476
- | • “wcstod32() — wcstod64() — wcstod128()— Convert Wide-Character String to Decimal Floating-Point” on page 478
- | • “<stdlib.h>” on page 17

## | **strtod32() — strtod64() — strtod128() — Convert Character String to Decimal Floating-Point**

### | **Format**

```
| #include <stdlib.h>
| _Decimal32 strtod32(const char *nptr, char **endptr);
| _Decimal64 strtod64(const char *nptr, char **endptr);
| _Decimal128 strtod128(const char *nptr, char **endptr);
```

| **Language Level:** ANSI

| **Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of these functions might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### | **Description**

| The strtod32(), strtod64(), and strtod128() functions convert a character string to a single-precision, double-precision, or quad-precision decimal floating-point value. The parameter *nptr* points to a sequence of characters that can be interpreted as a numeric decimal floating-point value. These functions stop reading the string at the first character that is not recognized as part of a number. This character can be the null character at the end of the string. The *endptr* parameter is updated to point to this character, provided that *endptr* is not a NULL pointer.

| The strtod32(), strtod64(), and strtod128() functions expect *nptr* to point to a string with the following form:



| The first character that does not fit this form stops the scan. In addition, a sequence of INFINITY or NAN (ignoring case) is allowed.

### | **Return Value**

| The `strtod32()`, `strtod64()`, and `strtod128()` functions return the value of the floating-point number, except when the representation causes an underflow or overflow. For an overflow, `strtod32()` returns `HUGE_VAL_D32` or `-HUGE_VAL_D32`; `strtod64()` returns `HUGE_VAL_D64` or `-HUGE_VAL_D64`; `strtod128()` returns `HUGE_VAL_D128` or `-HUGE_VAL_D128`. For an underflow, all functions return `+0.E0`.

| In both the overflow and underflow cases, `errno` is set to `ERANGE`. If the string pointed to by `nptr` does not have the expected form, a value of `+0.E0` is returned and the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a `NULL` pointer.

| The `strtod32()`, `strtod64()`, and `strtod128()` functions do not fail if a character other than a digit follows an `E` or `e` that is read as an exponent. For example, `100elf` is converted to the floating-point value `100.0`.

| A character sequence of `INFINITY` (ignoring case) yields a value of `INFINITY`. A character value of `NAN` yields a Quiet Not-A-Number (NaN) value.

| If necessary, the return value is rounded using the rounding mode `Round to Nearest, Ties to Even`.

| **Example that uses `strtod32()`, `strtod64()`, and `strtod128()`**

| This example converts the strings to single-precision, double-precision, and quad-precision decimal floating-point values. It prints the converted values and the substring that stopped the conversion.

```

| #include <stdlib.h>
| #include <stdio.h>
|
| int main(void)
| {
|     char *string, *stopstring;
|     _Decimal32 d32;
|     _Decimal64 d64;
|     _Decimal128 d128;
|
|     string = "3.1415926This stopped it";
|     d32 = strtod32(string, &stopstring);
|     printf("string = %s\n", string);
|     printf(" strtod32 = %Hf\n", d32);
|     printf(" Stopped scan at %s\n\n", stopstring);
|     string = "100ergs";
|     d32 = strtod32(string, &stopstring);
|     printf("string = \"%s\"\n", string);
|     printf(" strtod = %Hf\n", d32);
|     printf(" Stopped scan at \"%s\"\n\n", stopstring);
|
|     string = "3.1415926This stopped it";
|     d64 = strtod64(string, &stopstring);
|     printf("string = %s\n", string);
|     printf(" strtod = %Df\n", d64);
|     printf(" Stopped scan at %s\n\n", stopstring);
|     string = "100ergs";
|     d64 = strtod64(string, &stopstring);
|     printf("string = \"%s\"\n", string);
|     printf(" strtod = %Df\n", d64);
|     printf(" Stopped scan at \"%s\"\n\n", stopstring);
|
|     string = "3.1415926This stopped it";
|     d128 = strtod128(string, &stopstring);
|     printf("string = %s\n", string);
|     printf(" strtold = %DDf\n", d128);
|     printf(" Stopped scan at %s\n\n", stopstring);
|     string = "100ergs";
|     d128 = strtod128(string, &stopstring);
|     printf("string = \"%s\"\n", string);
|     printf(" strtold = %DDf\n", d128);
|     printf(" Stopped scan at \"%s\"\n\n", stopstring);
| }
|
| /***** Output should be similar to: *****/
|
| string = 3.1415926This stopped it
| strtod = 3.141593
| Stopped scan at This stopped it
|
| string = "100ergs"
| strtod = 100.000000
| Stopped scan at "ergs"
|
| string = 3.1415926This stopped it
| strtod= 3.141593
| Stopped scan at This stopped it
|
| string = "100ergs"
| strtod = 100.000000
| Stopped scan at "ergs"
|
|
|
|

```

```

| string = 3.1415926This stopped it
| strtold = 3.141593
| Stopped scan at This stopped it
|
| string = "100ergs"
| strtold = 100.000000
| Stopped scan at "ergs"
|
| */
|
|

```

### Related Information

- | • “atof() — Convert Character String to Float” on page 46
- | • “atoi() — Convert Character String to Integer” on page 48
- | • “atol() — atoll() — Convert Character String to Long or Long Long Integer” on page 49
- | • “strtod() — strtodf() — strtold — Convert Character String to Double, Float, and Long Double” on page 392
- | • “strtol() — strtoll() — Convert Character String to Long and Long Long Integer” on page 400
- | • “strtoul() — strtoull() — Convert Character String to Unsigned Long and Unsigned Long Long Integer” on page 403
- | • “wcstod() — Convert Wide-Character String to Double” on page 476
- | • “wcstod32() — wcstod64() — wcstod128()— Convert Wide-Character String to Decimal Floating-Point” on page 478
- | • “<stdlib.h>” on page 17

---

## strtok() — Tokenize String

### Format

```

#include <string.h>
char *strtok(char *string1, const char *string2);

```

**Language Level:** ANSI

**Threadsafe:** No. Use strtok\_r() instead.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The strtok() function reads *string1* as a series of zero or more tokens, and *string2* as the set of characters serving as delimiters of the tokens in *string1*. The tokens in *string1* can be separated by one or more of the delimiters from *string2*. The tokens in *string1* can be located by a series of calls to the strtok() function.

In the first call to the strtok() function for a given *string1*, the strtok() function searches for the first token in *string1*, skipping over leading delimiters. A pointer to the first token is returned.

When the strtok() function is called with a NULL *string1* argument, the next token is read from a stored copy of the last non-null *string1* parameter. Each delimiter is replaced by a null character. The set of delimiters can vary from call to call, so *string2* can take any value. Note that the initial value of *string1* is not preserved after the call to the strtok() function.

Note that the `strtok()` function writes data into the buffer. The function should be passed to a non-critical buffer containing the string to be tokenized because the buffer will be damaged by the `strtok()` function.

### Return Value

The first time the `strtok()` function is called, it returns a pointer to the first token in *string1*. In later calls with the same token string, the `strtok()` function returns a pointer to the next token in the string. A NULL pointer is returned when there are no more tokens. All tokens are null-ended.

**Note:** The `strtok()` function uses an internal static pointer to point to the next token in the string being tokenized. A reentrant version of the `strtok()` function, `strtok_r()`, which does not use any internal static storage, can be used in place of the `strtok()` function.

### Example that uses `strtok()`

Using a loop, this example gathers tokens, separated by commas, from a string until no tokens are left. The example prints the tokens, a string, of, and tokens.

```
#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 should be similar to: *****/

token: a string
token: of
token:
token: tokens
*/
```

### Related Information

- “`strcat()` — Concatenate Strings” on page 358
- “`strchr()` — Search for Character” on page 359
- “`strcmp()` — Compare Strings” on page 360
- “`strcpy()` — Copy Strings” on page 364
- “`strcspn()` — Find Offset of First Character Match” on page 365
- “`strspn()` — Find Offset of First Non-matching Character” on page 390
- “`strtok_r()` — Tokenize String (Restartable)”
- “`<string.h>`” on page 17

---

## `strtok_r()` — Tokenize String (Restartable)

### Format

```
#include <string.h>
char *strtok_r(char *string, const char *seps,
               char **lasts);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

- | **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

## Description

This function is the restartable version of `strtok()`.

The `strtok_r()` function reads *string* as a series of zero or more tokens, and *seps* as the set of characters serving as delimiters of the tokens in *string*. The tokens in *string* can be separated by one or more of the delimiters from *seps*. The argument *lasts* points to a user-provided pointer, which points to stored information necessary for the `strtok_r()` function to continue scanning the same string.

In the first call to the `strtok_r()` function for a given null-ended *string*, it searches for the first token in *string*, skipping over leading delimiters. It returns a pointer to the first character of the first token, writes a null character into *string* immediately following the returned token, and updates the pointer to which *lasts* points.

To read the next token from *string*, call the `strtok_r()` function with a NULL *string* argument. This causes the `strtok_r()` function to search for the next token in the previous token string. Each delimiter in the original *string* is replaced by a null character, and the pointer to which *lasts* points is updated. The set of delimiters in *seps* can vary from call to call, but *lasts* must remain unchanged from the previous call. When no tokens remain in *string*, a NULL pointer is returned.

## Return Value

The first time the `strtok_r()` function is called, it returns a pointer to the first token in *string*. In later calls with the same token string, the `strtok_r()` function returns a pointer to the next token in the string. A NULL pointer is returned when there are no more tokens. All tokens are null-ended.

## Related Information

- “`strcat()` — Concatenate Strings” on page 358
- “`strchr()` — Search for Character” on page 359
- “`strcmp()` — Compare Strings” on page 360
- “`strcpy()` — Copy Strings” on page 364
- “`strcspn()` — Find Offset of First Character Match” on page 365
- “`strspn()` — Find Offset of First Non-matching Character” on page 390
- “`strtok()` — Tokenize String” on page 398
- “`<string.h>`” on page 17

---

## strtol() — strtoll() — Convert Character String to Long and Long Long Integer

Format (`strtol()`)

```
#include <stdlib.h>
long int strtol(const char *nptr, char **endptr, int base);
```

### Format (strtoll())

```
#include <stdlib.h>
long long int strtoll(char *string, char **endptr, int base);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of these functions might be affected by the LC\_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `strtol()` function converts a character string to a long integer value. The parameter *nptr* points to a sequence of characters that can be interpreted as a numeric value of type long int.

The `strtoll()` function converts a character string to a long long integer value. The parameter *nptr* points to a sequence of characters that can be interpreted as a numeric value of type long long int.

When you use these functions, the *nptr* parameter should point to a string with the following form:



If the *base* parameter is a value between 2 and 36, the subject sequence’s expected form is a sequence of letters and digits representing an integer whose radix is specified by the *base* parameter. This sequence is optionally preceded by a positive (+) or negative (-) sign. Letters from a to z inclusive (either upper or lower case) are ascribed the values 10 to 35; only letters whose ascribed values are less than that of the base parameter are permitted. If the base parameter has a value of 16, the characters 0x or 0X optionally precede the sequence of letters and digits, following the positive (+) or negative (-) sign, if present.

If the value of the base parameter is 0, the string determines the base. After an optional leading sign a leading 0 indicates octal conversion, a leading 0x or 0X indicates hexadecimal conversion, and all other leading characters result in decimal conversion.

These functions scan the string up to the first character that is inconsistent with the *base* parameter. This character may be the null character ('\0') at the end of the string. Leading white-space characters are ignored, and an optional sign may precede the digits.

If the value of the *endptr* parameter is not null a pointer, a pointer to the character that ended the scan is stored in the value pointed to by *endptr*. If a value cannot be formed, the value pointed to by *endptr* is set to the *nptr* parameter

### Return Value

If *base* has an invalid value (less than 0, 1, or greater than 36), `errno` is set to `EINVAL` and 0 is returned. The value pointed to by the *endptr* parameter is set to the value of the *nptr* parameter.

If the value is outside the range of representable values, `errno` is set to `ERANGE`. If the value is positive, the `strtol()` function will return `LONG_MAX`, and the `strtoll()` function will return `LONGLONG_MAX`. If the value is negative, the `strtol()` function will return `LONG_MIN`, and the `strtoll()` function will return `LONGLONG_MIN`.

If no characters are converted, the `strtoll()` and `strtol()` functions will set `errno` to `EINVAL` and 0 is returned. For both functions, the value pointed to by `endptr` is set to the value of the `nptr` parameter. Upon successful completion, both functions return the converted value.

### Example that uses `strtol()`

This example converts the strings to a long 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;
    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", stopstring);
    }
}

/***** Output should be similar to: *****/

string = 10110134932
  strtol = 45 (base 2)
  Stopped scan at 34932

  strtol = 4423 (base 4)
  Stopped scan at 4932
```

### Related Information

- “`atof()` — Convert Character String to Float” on page 46
- “`atoi()` — Convert Character String to Integer” on page 48
- “`atol()` — `atoll()` — Convert Character String to Long or Long Long Integer” on page 49
- “`strtod()` — `strtof()` — `strtold()` — Convert Character String to Double, Float, and Long Double” on page 392
- “`strtod32()` — `strtod64()` — `strtod128()` — Convert Character String to Decimal Floating-Point” on page 395
- “`strtoul()` — `strtoull()` — Convert Character String to Unsigned Long and Unsigned Long Long Integer” on page 403
- “`wcstol()` — `wcstoll()` — Convert Wide Character String to Long and Long Long Integer” on page 481
- “`<stdlib.h>`” on page 17

---

## strtoul() — strtoull() — Convert Character String to Unsigned Long and Unsigned Long Long Integer

**Format** (strtoul())

```
#include <stdlib.h>
unsigned long int strtoul(const char *nptr, char **endptr, int base);
```

**Format** (strtoull())

```
#include <stdlib.h>
unsigned long long int strtoull(char *string, char **endptr, int base);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

- | **Locale Sensitive:** The behavior of these functions might be affected by the LC\_CTYPE category of the current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `strtoul()` function converts a character string to an unsigned long integer value. The parameter *nptr* points to a sequence of characters that can be interpreted as a numeric value of type unsigned long int.

The `strtoull()` function converts a character string to an unsigned long long integer value. The parameter *nptr* points to a sequence of characters that can be interpreted as a numeric value of type unsigned long long int.

When you use these functions, the *nptr* parameter should point to a string with the following form:



If the *base* parameter is a value between 2 and 36, the subject sequence’s expected form is a sequence of letters and digits representing an integer whose radix is specified by the base parameter. This sequence is optionally preceded by a positive (+) or negative (-) sign. Letters from a to z inclusive (either upper or lower case) are ascribed the values 10 to 35. Only letters whose ascribed values are less than that of the base parameter are permitted. If the *base* parameter has a value of 16 the characters `0x` or `0X` optionally precede the sequence of letters and digits, following the positive (+) or negative (-) sign, if present.

If the value of the *base* parameter is 0, the string determines the *base*. After an optional leading sign a leading 0 indicates octal conversion, a leading `0x` or `0X` indicates hexadecimal conversion, and all other leading characters result in decimal conversion.

These functions scan the string up to the first character that is inconsistent with the base parameter. This character may be the null character (`'\0'`) at the end of the string. Leading white-space characters are ignored, and an optional sign may precede the digits.

If the value of the *endptr* parameter is not null a pointer, a pointer to the character that ended the scan is stored in the value pointed to by *endptr*. If a value cannot be formed, the value pointed to by *endptr* is set to the *nptr* parameter.

**Return Value**

If *base* has an invalid value (less than 0, 1, or greater than 36), *errno* is set to *EINVAL* and 0 is returned. The value pointed to by the *endptr* parameter is set to the value of the *nptr* parameter.

If the value is outside the range of representable values, *errno* is set to *ERANGE*. The `strtoul()` function will return `ULONG_MAX` and the `strtoull()` function will return `ULLONG_MAX`.

If no characters are converted, the `strtoull()` function will set *errno* to *EINVAL* and 0 is returned. The `strtoul()` function will return 0 but will not set *errno* to *EINVAL*. In both cases the value pointed to by *endptr* is set to the value of the *nptr* parameter. Upon successful completion, both functions return the converted value.

### Example that uses `strtoul()`

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 should be similar to: *****/

string = 1000e13 e
    strtoul = 8 (base 2)
    Stopped scan at e13 e
*/
```

### Related Information

- “`atof()` — Convert Character String to Float” on page 46
- “`atoi()` — Convert Character String to Integer” on page 48
- “`atol()` — `atoll()` — Convert Character String to Long or Long Long Integer” on page 49
- “`strtod()` — `strtof()` — `strtold` — Convert Character String to Double, Float, and Long Double” on page 392
- “`strtod32()` — `strtod64()` — `strtod128()` — Convert Character String to Decimal Floating-Point” on page 395
- “`strtol()` — `strtoll()` — Convert Character String to Long and Long Long Integer” on page 400
- “`wcstoul()` — `wcstoull()` — Convert Wide Character String to Unsigned Long and Unsigned Long Long Integer” on page 486
- “`<stdlib.h>`” on page 17

---

## strxfrm() — Transform String

### Format

```
#include <string.h>
size_t strxfrm(char *string1, const char *string2, size_t count);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_COLLATE  
| categories of the current locale. For more information, see “Understanding CCSIDs and Locales” on page  
| 524.

## Description

The `strxfrm()` function transforms the string pointed to by `string2` and places the result into the string pointed to by `string1`. The transformation is determined by the program’s current locale. The transformed string is not necessarily readable, but can be used with the `strcmp()` or the `strncmp()` functions.

## Return Value

The `strxfrm()` function returns the length of the transformed string, excluding the ending null character. If the returned value is greater than or equal to `count`, the contents of the transformed string are indeterminate.

If `strxfrm()` is unsuccessful, `errno` is changed. The value of `errno` may be set to `EINVAL` (the `string1` or `string2` arguments contain characters which are not available in the current locale).

## Example that uses `strxfrm()`

This example prompts the user to enter a string of characters, then uses `strxfrm()` to transform the string and return its length.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
    char *string1, buffer[80];
    int length;

    printf("Type in a string of characters.\n ");
    string1 = gets(buffer);
    length = strxfrm(NULL, string1, 0);
    printf("You would need a %d element array to hold the string\n",length);
    printf("\n\n%s\n\n transformed according",string1);
    printf(" to this program's locale. \n");
}
```

## Related Information

- “`localeconv()` — Retrieve Information from the Environment” on page 181
- “`setlocale()` — Set Locale” on page 339
- “`strcmp()` — Compare Strings” on page 360
- “`strcoll()` — Compare Strings” on page 363
- “`strncmp()` — Compare Strings” on page 379
- “`<string.h>`” on page 17

---

## swprintf() — Format and Write Wide Characters to Buffer

### Format

```
#include <wchar.h>
int swprintf(wchar_t *wcsbuffer, size_t n,
             const wchar_t *format, argument-list);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

- | **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. The behavior might also be affected by the LC\_UNI\_CTYPE and LC\_UNI\_NUMERIC categories of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.
- | **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `swprintf()` function formats and stores a series of wide characters and values into the wide-character buffer `wcsbuffer`. The `swprintf()` function is equivalent to the `sprintf()` function, except that it operates on wide characters.

The value `n` specifies the maximum number of wide characters to be written, including the ending null character. The `swprintf()` function converts each entry in the `argument-list` according to the corresponding wide-character format specifier in `format`. The format has the same form and function as the format string for the `printf()` function, with the following exceptions:

- `%c` (without an `l` prefix) converts a character argument to `wchar_t`, as if by calling the `mbtowc()` function.
- `%lc` and `%C` copy a `wchar_t` to `wchar_t`. `%#lc` and `%#C` are equivalent to `%lc` and `%C`, respectively.
- `%s` (without an `l` prefix) converts an array of multibyte characters to an array of `wchar_t`, as if by calling the `mbstowcs()` function. The array is written up to, but not including, the ending null character, unless the precision specifies a shorter output.
- `%ls` and `%S` copy an array of `wchar_t` (no conversion). The array is written up to, but not including, the ending NULL character, unless the precision specifies a shorter output. `%#ls` and `%#S` are equivalent to `%ls` and `%S`, respectively.

Width and precision always are wide characters.

A null wide character is added to the end of the wide characters written; the null wide character is not counted as part of the returned value. If copying takes place between objects that overlap, the behavior is undefined.

### Return Value

- | The `swprintf()` function returns the number of wide characters that are written to the output buffer, not counting the ending null wide character or a negative value if an error is encountered. If `n` or more wide characters are requested to be written, a negative value is returned.

The value of `errno` may be set to `EINVAL`, invalid argument.

### Example that uses `swprintf()`

This example uses the `swprintf()` function to format and print several values to buffer.

```
#include <wchar.h>
#include <stdio.h>

#define BUF_SIZE 100

int main(void)
{
    wchar_t wcsbuf[BUF_SIZE];
    wchar_t wstring[] = L"ABCDE";
    int     num;

    num = swprintf(wcsbuf, BUF_SIZE, L"%s", "xyz");
    num += swprintf(wcsbuf + num, BUF_SIZE - num, L"%ls", wstring);
    num += swprintf(wcsbuf + num, BUF_SIZE - num, L"%i", 100);
    printf("The array wcsbuf contains: \"%ls\"\n", wcsbuf);
    return 0;

    /*****
    The output should be similar to :

    The array wcsbuf contains: "xyzABCDE100"
    *****/
}
```

### Related Information

- “`printf()` — Print Formatted Characters” on page 229
- “`sprintf()` — Print Formatted Data to Buffer” on page 352
- “`vswprintf()` — Format and Write Wide Characters to Buffer” on page 439
- “`<wchar.h>`” on page 18

---

## swscanf() — Read Wide Character Data

### Format

```
#include <wchar.h>
int swscanf(const wchar_t *buffer, const wchar_t *format, argument-list);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the `LC_CTYPE` and `LC_NUMERIC` categories of the current locale. The behavior might also be affected by the `LC_UNI_CTYPE` and `LC_UNI_NUMERIC` categories of the current locale if `LOCALETYPE(*LOCALEUCS2)` or `LOCALETYPE(*LOCALEUTF)` is specified on the compilation command. This function is not available when `LOCALETYPE(*CLD)` is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `swscanf()` function is equivalent of the `fwscanf()` function, except that the argument buffer specifies a wide string from which the input is to be obtained, rather than from a stream. Reaching the end of the wide string is equivalent to encountering end-of-file for the `fwscanf()` function.

### Return Value

The `swscanf()` function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is EOF when the end of the string is encountered before anything is converted.

The value of `errno` may be set `EINVAL`, invalid argument.

### Example that uses `swscanf()`

This example uses the `swscanf()` function to read various data from the string *ltokenstring*, and then displays that data.

```
#include <wchar.h>
#include <stdio.h>

wchar_t *ltokenstring = L"15 12 14";
int i;
float fp;
char s[10];
char c;

int main(void)
{
    /* Input various data */
    swscanf(ltokenstring, L"%s %c%d%f", s, &c, &i, &fp);

    /* If there were no space between %s and %c,
     * swscanf would read the first character following
     * the string, which is a blank space. */

    printf("string = %s\n",s);
    printf("character = %c\n",c);
    printf("integer = %d\n",i);
    printf("floating-point number = %f\n",fp);
}
```

### Related Information

- “`fscanf()` — Read Formatted Data” on page 132
- “`scanf()` — Read Data” on page 330
- “`fwscanf()` — Read Data from Stream Using Wide Character” on page 147
- “`wscanf()` — Read Data Using Wide-Character Format String” on page 504
- “`sscanf()` — Read Data” on page 355
- “`sprintf()` — Print Formatted Data to Buffer” on page 352
- “`<wchar.h>`” on page 18

---

## system() — Execute a Command

### Format

```
#include <stdlib.h>
int system(const char *string);
```

**Language Level:** ANSI

**Threadsafe:** Yes. However, the CL command processor and all CL commands are NOT threadsafe. Use this function with caution.

### Description

The `system()` function passes the given *string* to the CL command processor for processing.

### Return Value

If passed a non-NULL pointer to a string, the `system()` function passes the argument to the CL command processor. The `system()` function returns zero if the command is successful. If passed a NULL pointer to a string, `system()` returns -1, and the command processor is not called. If the command fails, `system()` returns 1. If the `system()` function fails, the global variable `_EXCP_MSGID` in `<stddef.h>` is set with the exception message ID. The exception message ID set within the `_EXCP_MSGID` variable is in job CCSID.

### Example that uses `system()`

```
#include <stdlib.h>

int main(void)
{
    int result;

    /* A data area is created, displayed and deleted: */

    result = system("CRTDTAARA QTEMP/TEST TYPE(*CHAR) VALUE('Test')");
    result = system("DSPDTAARA TEST");
    result = system("DLTDTAARA TEST");

}
```

### Related Information

- “`exit()` — End Program” on page 88
- “`<stdlib.h>`” on page 17

---

## `tan()` — Calculate Tangent

### Format

```
#include <math.h>
double tan(double x);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `tan()` function calculates the tangent of  $x$ , where  $x$  is expressed in radians. If  $x$  is too large, a partial loss of significance in the result can occur and sets `errno` to `ERANGE`. The value of `errno` may also be set to `EDOM`.

### Return Value

The `tan()` function returns the value of the tangent of  $x$ .

### Example that uses `tan()`

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 should be similar to: *****/

tan( 0.785398 ) is 1.000000
*/

```

### Related Information

- “acos() — Calculate Arccosine” on page 38
- “asin() — Calculate Arcsine” on page 42
- “atan() – atan2() — Calculate Arctangent” on page 44
- “cos() — Calculate Cosine” on page 64
- “cosh() — Calculate Hyperbolic Cosine” on page 65
- “sin() — Calculate Sine” on page 348
- “sinh() — Calculate Hyperbolic Sine” on page 349
- “tanh() — Calculate Hyperbolic Tangent”
- “<math.h>” on page 8

---

## tanh() — Calculate Hyperbolic Tangent

### Format

```

#include <math.h>
double tanh(double x);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `tanh()` function calculates the hyperbolic tangent of  $x$ , where  $x$  is expressed in radians.

### Return Value

The `tanh()` function returns the value of the hyperbolic tangent of  $x$ . The result of `tanh()` cannot have a range error.

### Example that uses `tanh()`

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 should be similar to: *****/

tanh( 0.785398 ) = 0.655794
*/

```

### Related Information

- “acos() — Calculate Arccosine” on page 38
- “asin() — Calculate Arcsine” on page 42
- “atan() – atan2() — Calculate Arctangent” on page 44
- “cos() — Calculate Cosine” on page 64
- “cosh() — Calculate Hyperbolic Cosine” on page 65
- “sin() — Calculate Sine” on page 348
- “sinh() — Calculate Hyperbolic Sine” on page 349
- “tan() — Calculate Tangent” on page 409
- “<math.h>” on page 8

---

## time() — Determine Current Time

### Format

```

#include <time.h>
time_t time(time_t *timeptr);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `time()` function determines the current calendar time, in seconds.

**Note:** Calendar time is the number of seconds that have elapsed since EPOCH, which is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).

### Return Value

The `time()` function returns the current calendar time. The return value is also stored in the location that is given by `timeptr`. If `timeptr` is NULL, the return value is not stored. If the calendar time is not available, the value `(time_t)(-1)` is returned.

### Example that uses `time()`

This example gets the time and assigns it to *ltime*. The `ctime()` function then converts the number of seconds to the current date and time. This example then prints a message giving the current time.

```
#include <time.h>
#include <stdio.h>

int main(void)
{
    time_t ltime;
    if(time(&ltime) == -1)
    {
        printf("Calendar time not available.\n");
        exit(1);
    }
    printf("The time is %s\n", ctime(&ltime));
}

/***** Output should be similar to: *****/

The time is Mon Mar 22 19:01:41 2004
*/
```

### Related Information

- “`asctime()` — Convert Time to Character String” on page 39
- “`asctime_r()` — Convert Time to Character String (Restartable)” on page 41
- “`ctime()` — Convert Time to Character String” on page 71
- | • “`ctime64()` — Convert Time to Character String” on page 73
- | • “`ctime64_r()` — Convert Time to Character String (Restartable)” on page 76
- “`ctime_r()` — Convert Time to Character String (Restartable)” on page 74
- “`gmtime()` — Convert Time” on page 161
- | • “`gmtime64()` — Convert Time” on page 163
- | • “`gmtime64_r()` — Convert Time (Restartable)” on page 167
- “`gmtime_r()` — Convert Time (Restartable)” on page 165
- “`localtime()` — Convert Time” on page 185
- | • “`localtime64()` — Convert Time” on page 187
- | • “`localtime64_r()` — Convert Time (Restartable)” on page 189
- “`localtime_r()` — Convert Time (Restartable)” on page 188
- “`mktime()` — Convert Local Time” on page 218
- | • “`mktime64()` — Convert Local Time” on page 220
- | • “`time64()` — Determine Current Time”
- “`<time.h>`” on page 18

---

## | `time64()` — Determine Current Time

### | Format

```
| #include <time.h>
| time64_t time64(time64_t *timeptr);
```

| **Language Level:** ILE C Extension.

| **Threadsafe:** Yes.

### | Description

| The `time64()` function determines the current calendar time, in seconds.

| **Note:** Calendar time is the number of seconds that have elapsed since EPOCH, which is 00:00:00, January 1, 1970 Universal Coordinate Time (UTC).

### | **Return Value**

| The `time64()` function returns the current calendar time. The return value is also stored in the location that is given by `timeptr`. If `timeptr` is NULL, the return value is not stored. If the calendar time is not available, the value `(time_t)(-1)` is returned.

### | **Example that uses `time64()`**

| This example gets the time and assigns it to `ltime`. The `ctime64()` function then converts the number of seconds to the current date and time. This example then prints a message giving the current time.

```
| #include <time.h>
| #include <stdio.h>
|
| int main(void)
| {
|     time64_t ltime;
|
|     if(time64(&ltime) == -1)
|     {
|         printf("Calendar time not available.\n");
|         exit(1);
|     }
|     printf("The time is %s", ctime64(&ltime));
| }
|
| /***** Output should be similar to: *****/
|
| The time is Mon Mar 22 19:01:41 2004
| */
```

### | **Related Information**

- | • “`asctime()` — Convert Time to Character String” on page 39
- | • “`asctime_r()` — Convert Time to Character String (Restartable)” on page 41
- | • “`ctime()` — Convert Time to Character String” on page 71
- | • “`ctime64()` — Convert Time to Character String” on page 73
- | • “`ctime_r()` — Convert Time to Character String (Restartable)” on page 74
- | • “`ctime()` — Convert Time to Character String” on page 71
- | • “`gmtime()` — Convert Time” on page 161
- | • “`gmtime64()` — Convert Time” on page 163
- | • “`gmtime64_r()` — Convert Time (Restartable)” on page 167
- | • “`gmtime_r()` — Convert Time (Restartable)” on page 165
- | • “`localtime()` — Convert Time” on page 185
- | • “`localtime64()` — Convert Time” on page 187
- | • “`localtime64_r()` — Convert Time (Restartable)” on page 189
- | • “`localtime_r()` — Convert Time (Restartable)” on page 188
- | • “`mktime()` — Convert Local Time” on page 218
- | • “`mktime64()` — Convert Local Time” on page 220
- | • “`time()` — Determine Current Time” on page 411
- | • “`<time.h>`” on page 18

---

## tmpfile() — Create Temporary File

### Format

```
#include <stdio.h>
FILE *tmpfile(void);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

The `tmpfile()` function creates a temporary binary file. The file is automatically removed when it is closed or when the program is ended.

The `tmpfile()` function opens the temporary file in `wb+` mode.

### Return Value

The `tmpfile()` function returns a stream pointer, if successful. If it cannot open the file, it returns a NULL pointer. On normal end (`exit()`), these temporary files are removed.

On the i5/OS Data Management system, the `tmpfile()` function creates a new file that is named `QTEMP/QACXxxxx`. If you specify the `SYSIFCOPT(*IFSIO)` option on the compilation command, the `tmpfile()` function creates a new file that is named `/tmp/QACXaaaaaaaa`. At the end of the job, the file that is created with the filename from the `tmpfile()` function is discarded. You can use the `remove()` function to remove files.

### Example that uses tmpfile()

This example creates a temporary file, and if successful, writes `tmpstring` to it. At program end, the file is removed.

```
#include <stdio.h>

FILE *stream;
char tmpstring[ ] = "This is the string to be temporarily written";

int main(void)
{
    if((stream = tmpfile( )) == NULL)
        perror("Cannot make a temporary file");
    else
        fprintf(stream, "%s", tmpstring);
}
```

### Related Information

- “`fopen()` — Open Files” on page 109
- “`<stdio.h>`” on page 15

---

## tmpnam() — Produce Temporary File Name

### Format

```
#include <stdio.h>
char *tmpnam(char *string);
```

**Language Level:** ANSI

**Threadsafe:** Yes. However, using tmpnam(NULL) is NOT threadsafe.

### Description

The tmpnam() function produces a valid file name that is not the same as the name of any existing file. It stores this name in *string*. If *string* is NULL, the tmpnam() function leaves the result in an internal static buffer. Any subsequent calls destroy this value. If *string* is not NULL, it must point to an array of at least L\_tmpnam bytes. The value of L\_tmpnam is defined in <stdio.h>.

The tmpnam() function produces a different name each time it is called within an activation group up to at least TMP\_MAX names. For ILE C, TMP\_MAX is 32 767. This is a theoretical limit; the actual number of files that can be opened at the same time depends on the available space in the system.

### Return Value

The tmpnam() function returns a pointer to the name. If it cannot create a unique name then it returns NULL.

### Example that uses tmpnam()

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

- “fopen() — Open Files” on page 109
- “remove() — Delete File” on page 274
- “<stdio.h>” on page 15

---

## toascii() — Convert Character to Character Representable by ASCII

### Format

```
#include <ctype.h>
int toascii(int c);
```

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the  
| current locale. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation  
| command. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `toascii()` function determines to what character *c* would be mapped to in a 7-bit US-ASCII locale and returns the corresponding character encoding in the current locale.

### Return Value

The `toascii()` function maps the character *c* according to a 7-bit US-ASCII locale and returns the corresponding character encoding in the current locale.

### Example that uses `toascii()`

This example prints encodings of the 7-bit US-ASCII characters 0x7c to 0x82 are mapped to by `toascii()`.

```
#include <stdio.h>
#include <ctype.h>

void main(void)
{
    int ch;

    for (ch=0x7c; ch<=0x82; ch++) {
        printf("toascii(0x%04x) = %c\n", ch, toascii(ch));
    }
}

/*****And the output should be:*****/
toascii(0x7c) = @
toascii(0x7d) = '
toascii(0x7e) = =
toascii(0x7f) = "
toascii(0x80) = X
toascii(0x81) = a
toascii(0x82) = b
*****/
```

### Related Information

- “`isascii()` — Test for Character Representable as ASCII Value” on page 171
- “`<ctype.h>`” on page 3

---

## tolower() – toupper() — Convert Character Case

### Format

```
#include <ctype.h>
int tolower(int C);
int toupper(int c);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of these functions might be affected by the `LC_CTYPE` category of the  
| current locale. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `tolower()` function converts the uppercase letter *C* to the corresponding lowercase letter.

The `toupper()` function converts the lowercase letter *c* to the corresponding uppercase letter.

## Return Value

Both functions return the converted character. If the character *c* does not have a corresponding lowercase or uppercase character, the functions return *c* unchanged.

### Example that uses toupper() and tolower()

This example uses the toupper() and tolower() functions to change characters between code 0 and code 7f.

```
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    int ch;

    for (ch = 0; ch <= 0x7f; ch++)
    {
        printf("toupper=%#04x\n", toupper(ch));
        printf("tolower=%#04x\n", tolower(ch));
        putchar('\n');
    }
}
```

## Related Information

- “isalnum() - isxdigit() — Test Integer Value” on page 169
- “tolower() -toupper() — Convert Wide Character Case” on page 418
- “<ctype.h>” on page 3

---

## towctrans() — Translate Wide Character

### Format

```
#include <wctype.h>
wint_t towctrans(wint_t wc, wctrans_t desc);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the  
| current locale if LOCALETYPE(\*LOCALE) is specified on the compilation command. It might also be  
| affected by the LC\_UNI\_CTYPE category of the current locale if either the  
| LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) option is specified on the compilation  
| command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation  
| command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

| The towctrans() function maps the wide character *wc* using the mapping that is described by *desc*.

A towctrans(wc, wctrans("tolower")) behaves in the same way as the call to the wide-character, case-mapping function tolower().

A `towctrans(wc, wctrans("toupper"))` behaves in the same way as the call to the wide-character, case-mapping function `toupper()`.

### Return Value

The `towctrans()` function returns the mapped value of `wc` using the mapping that is described by *desc*.

### Example that uses `towctrans()`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wchar.h>
#include <wctype.h>

int main()
{
    char *alpha = "abcdefghijklmnopqrstuvwxyz";
    char *tocase[2] = {"toupper", "tolower"};
    wchar_t *wcalpha;
    int i, j;
    size_t alphalen;

    alphalen = strlen(alpha)+1;
    wcalpha = (wchar_t *)malloc(sizeof(wchar_t)*alphalen);

    mbstowcs(wcalpha, alpha, 2*alphalen);

    for (i=0; i<2; ++i) {
        printf("Input string: %ls\n", wcalpha);
        for (j=0; j
            for (j=0; j
```

### Related Information

- “`wctrans()` —Get Handle for Character Mapping” on page 493
- “`<wchar.h>`” on page 18

---

## towlower() –toupper() — Convert Wide Character Case

### Format

```
#include <wctype.h>
wint_t tolower(wint_t wc);
wint_t toupper(wint_t wc);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of these functions might be affected by the `LC_CTYPE` category of the  
| current locale if `LOCALETYPE(*LOCALE)` is specified on the compilation command. The behavior of  
| these functions might also be affected by the `LC_UNI_CTYPE` category of the current locale if either the  
| `LOCALETYPE(*LOCALEUCS2)` or `LOCALETYPE(*LOCALEUTF)` option is specified on the compilation  
| command. These functions are not available when `LOCALETYPE(*CLD)` is specified on the compilation  
| command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `toupper()` function converts the lowercase character *wc* to the corresponding uppercase letter. The `tolower()` function converts the uppercase character *wc* to the corresponding lowercase letter.

### Return Value

If *wc* is a wide character for which `iswupper()` (or `iswlower()`) is true and there is a corresponding wide character for which `iswlower()` (or `iswupper()`) is true, `tolower()` (or `toupper()`) returns the corresponding wide character. Otherwise, the argument is returned unchanged.

### Example that uses `tolower()` and `toupper()`

This example uses `tolower()` and `toupper()` to convert characters between 0 and 0x7f.

```
#include <wctype.h>
#include <stdio.h>

int main(void)
{
    wint_t w_ch;

    for (w_ch = 0; w_ch <= 0xff; w_ch++) {
        printf ("toupper : %#04x %#04x, ", w_ch, toupper(w_ch));
        printf ("tolower : %#04x %#04x\n", w_ch, tolower(w_ch));
    }
    return 0;
}
/*****
The output should be similar to:
```

```
:
toupper : 0xc1 0xc1, tolower : 0xc1 0x81
toupper : 0xc2 0xc2, tolower : 0xc2 0x82
toupper : 0xc3 0xc3, tolower : 0xc3 0x83
toupper : 0xc4 0xc4, tolower : 0xc4 0x84
toupper : 0xc5 0xc5, tolower : 0xc5 0x85
:
toupper : 0x81 0xc1, tolower : 0x81 0x81
toupper : 0x82 0xc2, tolower : 0x82 0x82
toupper : 0x83 0xc3, tolower : 0x83 0x83
toupper : 0x84 0xc4, tolower : 0x84 0x84
toupper : 0x85 0xc5, tolower : 0x85 0x85
:
*****/
}
```

### Related Information

- “`iswalnum()` to `iswxdigit()` — Test Wide Integer Value” on page 173
- “`tolower()` – `toupper()` — Convert Character Case” on page 416
- “`<wctype.h>`” on page 19

---

## **`_ultoa` - Convert Unsigned Long Integer to String**

### Format

```
#include <stdlib.h>
char *_ultoa(unsigned long value, char *string, int radix);
```

**Note:** The `_ultoa` function is supported only for C++, not for C.

**Language Level:** Extension

**Threadsafe:** Yes.

## Description

`_ultoa` converts the digits of the given unsigned long *value* to a character string that ends with a null character and stores the result in *string*. The *radix* argument specifies the base of *value*; it must be in the range 2 to 36.

**Note:** The space allocated for *string* must be large enough to hold the returned string. The function can return up to 33 bytes including the null character (`\0`).

## Return Value

`_ultoa` returns a pointer to *string*. There is no error return value.

When the string argument is NULL or the *radix* is outside the range 2 to 36, `errno` will be set to `EINVAL`.

## Example that uses `_ultoa()`

This example converts the integer value 255 to a decimal, binary, and hexadecimal representation.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    char buffer[35];
    char *p;
    p = _ultoa(255UL, buffer, 10);
    printf("The result of _ultoa(255) with radix of 10 is %s\n", p);
    p = _ultoa(255UL, buffer, 2);
    printf("The result of _ultoa(255) with radix of 2\n    is %s\n", p);
    p = _ultoa(255UL, buffer, 16);
    printf("The result of _ultoa(255) with radix of 16 is %s\n", p);
    return 0;
}
```

The output should be:

```
The result of _ultoa(255) with radix of 10 is 255
The result of _ultoa(255) with radix of 2
    is 11111111
The result of _ultoa(255) with radix of 16 is ff
```

## Related Information

- “`_gcvt` - Convert Floating-Point to String” on page 151
- “`_itoa` - Convert Integer to String” on page 176
- “`_ltoa` - Convert Long Integer to String” on page 192
- “`<stdlib.h>`” on page 17

---

## `ungetc()` — Push Character onto Input Stream

### Format

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

## Description

The `ungetc()` function pushes the unsigned character *c* back onto the given input *stream*. However, only one consecutive character is guaranteed to be pushed back onto the input stream if you call `ungetc()` consecutively. The *stream* must be open for reading. A subsequent read operation on the *stream* starts with *c*. The character *c* cannot be the EOF character.

Characters placed on the stream by `ungetc()` will be erased if `fseek()`, `fsetpos()`, `rewind()`, or `fflush()` is called before the character is read from the *stream*.

### Return Value

The `ungetc()` function returns the integer argument *c* converted to an unsigned char, or EOF if *c* cannot be pushed back.

The value of `errno` may be set to:

#### Value Meaning

##### ENOTREAD

The file is not open for read operations.

##### EIOERROR

A non-recoverable I/O error occurred.

##### EIORECEERR

A recoverable I/O error occurred.

The `ungetc()` function is not supported for files opened with `type=record`.

### Example that uses `ungetc()`

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 non-digit character appears before the end of the file, `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;
    while ((ch = getc(stream)) != EOF && isdigit(ch))
        result = result * 10 + ch - '0';
    if (ch != EOF)
        ungetc(ch, stream);
    /* Put the nondigit character back */
    printf("The result is: %d\n", result);
    if ((ch = getc(stream)) != EOF)
        printf("The character is: %c\n", ch);
}
```

### Related Information

- “`getc()` – `getchar()` — Read a Character” on page 152
- “`fflush()` — Write Buffer to File” on page 96
- “`fseek()` — `fseeko()` — Reposition File Position” on page 134
- “`fsetpos()` — Set File Position” on page 136
- “`putc()` – `putchar()` — Write a Character” on page 239

- “rewind() — Adjust Current File Position” on page 276
- “<stdio.h>” on page 15

---

## ungetwc() — Push Wide Character onto Input Stream

### Format

```
#include <wchar.h>
#include <stdio.h>
wint_t ungetwc(wint_t wc, FILE *stream);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

- | **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. The behavior might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.
- | **Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified on the compilation command.
- | **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `ungetwc()` function pushes the wide character `wc` back onto the input stream. The pushed-back wide characters will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call (on the 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` is `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. However, only one consecutive wide character is guaranteed to be pushed back onto the input stream if you call `ungetwc()` consecutively.

For a text stream, the file position indicator is backed up by one wide character. This affects the `ftell()`, `fflush()`, `fseek()` (with `SEEK_CUR`), and `fgetpos()` function. 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 the `ftell()`, `fseek()` (with `SEEK_CUR`), `fgetpos()`, and `fflush()` function.

### Return Value

The `ungetwc()` function returns the wide character pushed back after conversion, or `WEOF` if the operation fails.

### Example that uses `ungetwc()`

```
#include <wchar.h>
#include <wctype.h>
#include <stdio.h>
#include <stdlib.h>
```

```

int main(void)
{
    FILE          *stream;
    wint_t        wc;
    wint_t        wc2;
    unsigned int  result = 0;

    if (NULL == (stream = fopen("ungetwc.dat", "r+"))) {
        printf("Unable to open file.\n");
        exit(EXIT_FAILURE);
    }

    while (WEOF != (wc = fgetwc(stream)) && iswdigit(wc))
        result = result * 10 + wc - L'0';

    if (WEOF != wc)
        ungetwc(wc, stream);    /* Push the nondigit wide character back */

    /* get the pushed back character */
    if (WEOF != (wc2 = fgetwc(stream))) {
        if (wc != wc2) {
            printf("Subsequent fgetwc does not get the pushed back character.\n");
            exit(EXIT_FAILURE);
        }
        printf("The digits read are '%i'\n"
            "The character being pushed back is '%lc'", result, wc2);
    }
    return 0;

    /*****
    Assuming the file ungetwc.dat contains:

    12345ABCDE67890XYZ

    The output should be similar to :

    The digits read are '12345'
    The character being pushed back is 'A'
    *****/
}

```

## Related Information

- “fflush() — Write Buffer to File” on page 96
- “fseek() — fseeko() — Reposition File Position” on page 134
- “fsetpos() — Set File Position” on page 136
- “getwc() — Read Wide Character from Stream” on page 157
- “putwc() — Write Wide Character” on page 242
- “ungetc() — Push Character onto Input Stream” on page 420
- “<wchar.h>” on page 18

---

## va\_arg() – va\_end() – va\_start() — Access Function Arguments

### 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);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

## Description

The `va_arg()`, `va_end()`, and `va_start()` functions 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.

`va_start()` initializes the *arg\_ptr* pointer for subsequent calls to `va_arg()` and `va_end()`.

The argument *variable\_name* is the identifier of the rightmost named parameter in the parameter list (preceding `,` ...). Use `va_start()` before `va_arg()`. Corresponding `va_start()` and `va_end()` macros must be in the same function.

The `va_arg()` function 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()` function can retrieve arguments from the list any number of times within the function. The *var\_type* argument must be one of `int`, `long`, `decimal`, `double`, `struct`, `union`, or `pointer`, or a typedef of one of these types.

The `va_end()` function is needed to indicate the end of parameter scanning.

- | Because it is not always possible for the called routine to determine how many arguments there are, the
- | calling routine should communicate the number of arguments to the called routine. To determine the
- | number of arguments, a routine can use a null pointer to signal the end of the list or pass the count of
- | the optional arguments as one of the required arguments. The `printf()` function, for instance, can tell
- | how many arguments there are through the *format-string* argument.

## Return Value

The `va_arg()` function returns the current argument. The `va_end` and `va_start()` functions do not return a value.

### Example that uses `va_arg()` – `va_end()` – `va_start()`

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>

int vout(int max, ...);

int main(void)
{
    vout(3, "Sat", "Sun", "Mon");
    printf("\n");
    vout(5, "Mon", "Tues", "Wed", "Thurs", "Fri");
}

int 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 should be similar to: *****/

Day: Sat
Day: Sun
Day: Mon

Day: Mon
Day: Tues
Day: Wed
Day: Thurs
Day: Fri
*/

```

### Related Information

- “`vfprintf()` — Print Argument Data to Stream”
- “`vprintf()` — Print Argument Data” on page 432
- “`vwprintf()` — Format Argument Data as Wide Characters and Write to a Stream” on page 428
- “`vsprintf()` — Print Argument Data to Buffer” on page 436
- “`<stdarg.h>`” on page 13

---

## vfprintf() — Print Argument Data to Stream

### Format

```

#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE *stream, const char *format, va_list arg_ptr);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the `LC_CTYPE` and `LC_NUMERIC` categories of the current locale. The behavior might also be affected by the `LC_UNI_CTYPE` category of the current locale if `LOCALETYPE(*LOCALEUCS2)` or `LOCALETYPE(*LOCALEUTF)` is specified on the

| compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the  
| compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

## Description

The `vfprintf()` function formats and writes a series of characters and values to the output *stream*. The `vfprintf()` function works just like the `fprintf()` function, except that *arg\_ptr* points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by `va_start` for each call. In contrast, the `fprintf()` function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

The `vfprintf()` function converts each entry in the argument list according to the corresponding format specifier in *format*. The *format* has the same form and function as the format string for the `printf()` function.

## Return Value

If successful, `vfprintf()` returns the number of bytes written to *stream*. If an error occurs, the function returns a negative value.

## Example that uses `vfprintf()`

This example prints out a variable number of strings to the file `myfile`.

```
#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("mylib/myfile", "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 should be similar to: *****/

Sat Sun Mon
*/
```

## Related Information

- “`fprintf()` — Write Formatted Data to a Stream” on page 116
- “`printf()` — Print Formatted Characters” on page 229
- “`va_arg()` – `va_end()` – `va_start()` — Access Function Arguments” on page 423
- “`vprintf()` — Print Argument Data” on page 432
- “`vsprintf()` — Print Argument Data to Buffer” on page 436

- “vwprintf() — Format Argument Data as Wide Characters and Print” on page 443
- “<stdarg.h>” on page 13
- “<stdio.h>” on page 15

---

## vfscanf() — Read Formatted Data

### Format

```
#include <stdarg.h>
#include <stdio.h>
int vfscanf(FILE *stream, const char *format, va_list arg_ptr);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. The behavior might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `vfscanf()` function reads data from a stream into locations specified by a variable number of arguments. The `vfscanf()` function works just like the `fscanf()` function, except that `arg_ptr` points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by `va_start` for each call. In contrast, the `fscanf()` function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

Each argument must be a pointer to a variable with a type that corresponds to a type specifier in format-string. The *format* has the same form and function as the format string for the `scanf()` function.

### Return Value

The `vfscanf()` function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is EOF for an attempt to read at end-of-file if no conversion was performed. A return value of 0 means that no fields were assigned.

### Example that uses `vfscanf()`

This example opens the file *myfile* for input, and then scans this file for a string, a long integer value, and a floating-point value.

```

#include <stdio.h>
#include <stdarg.h>

int vread(FILE *stream, char *fmt, ...)
{
    int rc;
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    rc = vfscanf(stream, fmt, arg_ptr);
    va_end(arg_ptr);
    return(rc);
}

#define MAX_LEN 80
int main(void)
{
    FILE *stream;
    long l;
    float fp;
    char s[MAX_LEN + 1];
    char c;
    stream = fopen("mylib/myfile", "r");
    /* Put in various data. */
    vread(stream, "%s", &s[0]);
    vread(stream, "%ld", &l);
    vread(stream, "%c", &c);
    vread(stream, "%f", &fp);
    printf("string = %s\n", s);
    printf("long double = %ld\n", l);
    printf("char = %c\n", c);
    printf("float = %f\n", fp);
}
/***** If myfile contains *****/
/***** abcdefghijklmnopqrstuvwxyz 343.2 *****/
/***** expected output is: *****/
string = abcdefghijklmnopqrstuvwxyz
long double = 343
char = .
float = 2.000000
*/

```

## Related Information

- “[fprintf\(\)](#) — Write Formatted Data to a Stream” on page 116
- “[fscanf\(\)](#) — Read Formatted Data” on page 132
- “[fwscanf\(\)](#) — Read Data from Stream Using Wide Character” on page 147
- “[scanf\(\)](#) — Read Data” on page 330
- “[sscanf\(\)](#) — Read Data” on page 355
- “[swscanf\(\)](#) — Read Wide Character Data” on page 407
- “[wscanf\(\)](#) — Read Data Using Wide-Character Format String” on page 504
- “[<stdio.h>](#)” on page 15

---

## vfwprintf() — Format Argument Data as Wide Characters and Write to a Stream

### Format

```

#include <stdarg.h>
#include <stdio.h>
#include <wchar.h>
int vfwprintf(FILE *stream, const wchar_t *format, va_list arg);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. The behavior might also be affected by the LC\_UNI\_CTYPE and LC\_UNI\_NUMERIC categories of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified on the compilation command.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `vwprintf()` function is equivalent to the `fprintf()` function, except that the variable argument list is replaced by `arg`, which the `va_start` macro (and possibly subsequent `va_arg` calls) will have initialized. The `vwprintf()` function does not invoke the `va_end` macro.

Because the functions `vwprintf()`, `vswprintf()`, and `wprintf()` invoke the `va_arg` macro, the value of `arg` after the return is unspecified.

### Return Value

| The `vwprintf()` function returns the number of wide characters that are written to the output buffer, not counting the ending null wide character or a negative value if an error was encountered. If `n` or more wide characters are requested to be written, a negative value is returned.

### Example that uses `vwprintf()`

This example prints the wide character `a` to a file. The printing is done from the `vout()` function, which takes a variable number of arguments and uses `vwprintf()` to print them to a file.

```
#include <wchar.h>
#include <stdarg.h>
#include <locale.h>

void vout (FILE *stream, wchar_t *fmt, ...);

const char ifs_path [] = "tmp/myfile";

int main(void) {
    FILE *stream;
    wchar_t format [] = L"%lc";

    setlocale(LC_ALL, "POSIX");
    if ((stream = fopen (ifs_path, "w")) == NULL) {
        printf("Could not open file.\n");
        return (-1);
    }
    vout (stream, format, L'a');
    fclose (stream);

    /*****
    The contents of output file tmp/myfile.dat should
    be a wide char 'a' which in the "POSIX" locale
    is '0081'x.
    *****/
}
```

```

*/
return (0);

}

void vout (FILE *stream, wchar_t *fmt, ...)
{
    va_list arg_ptr;
    va_start (arg_ptr, fmt);
    vfwprintf (stream, fmt, arg_ptr);
    va_end (arg_ptr);
}

```

### Related Information

- “printf() — Print Formatted Characters” on page 229
- “fprintf() — Write Formatted Data to a Stream” on page 116
- “vfprintf() — Print Argument Data to Stream” on page 425
- “vprintf() — Print Argument Data” on page 432
- “btowc() — Convert Single Byte to Wide Character” on page 53
- “mbrtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 201
- “fwprintf() — Format Data as Wide Characters and Write to a Stream” on page 143
- “vswprintf() — Format and Write Wide Characters to Buffer” on page 439
- “vwprintf() — Format Argument Data as Wide Characters and Print” on page 443
- “<stdarg.h>” on page 13
- “<stdio.h>” on page 15
- “<wchar.h>” on page 18

---

## vwscanf() — Read Formatted Wide Character Data

### Format

```

#include <stdarg.h>
#include <stdio.h>
int vwscanf(FILE *stream, const wchar_t *format, va_list arg_ptr);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

- | **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. The behavior might also be affected by the LC\_UNI\_CTYPE and LC\_UNI\_NUMERIC categories of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.
- | **Integrated File System Interface:** See “Wide Characters” on page 527 for more information.
- | **Wide Character Function:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified on the compilation command.

### Description

The `vfwscanf()` function reads wide data from a stream into locations specified by a variable number of arguments. The `vfwscanf()` function works just like the `fscanf()` function, except that *arg\_ptr* points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by *va\_start* for each call. In contrast, the `fscanf()` function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

Each argument must be a pointer to a variable with a type that corresponds to a type specifier in format-string. The *format* has the same form and function as the format string for the `fscanf()` function.

## Return Value

The `vfwscanf()` function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is EOF for an attempt to read at end-of-file if no conversion was performed. A return value of 0 means that no fields were assigned.

## Example that uses `vfwscanf()`

This example opens the file *myfile* for input, and then scans this file for a string, a long integer value, and a floating-point value.

```
#include <stdio.h>
#include <stdarg.h>
#include <wchar.h>

int vread(FILE *stream, wchar_t *fmt, ...)
{
    int rc;
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    rc = vfwscanf(stream, fmt, arg_ptr);
    va_end(arg_ptr);
    return(rc);
}

#define MAX_LEN 80
int main(void)
{
    FILE *stream;
    long l;
    float fp;
    char s[MAX_LEN + 1];
    char c;
    stream = fopen("mylib/myfile", "r");
    /* Put in various data. */
    vread(stream, L"%s", &s [0]);
    vread(stream, L"%ld", &l);
    vread(stream, L"%c", &c);
    vread(stream, L"%f", &fp);
    printf("string = %s\n", s);
    printf("long double = %ld\n", l);
    printf("char = %c\n", c);
    printf("float = %f\n", fp);
}
/***** If myfile contains *****/
/***** abcdefghijklmnopqrstuvwxyz 343.2 *****/
/***** expected output is: *****/
string = abcdefghijklmnopqrstuvwxyz
long double = 343
char = .
float = 2.000000
*/
```

## Related Information

- “`fscanf()` — Read Formatted Data” on page 132
- “`fwprintf()` — Format Data as Wide Characters and Write to a Stream” on page 143
- “`fwscanf()` — Read Data from Stream Using Wide Character” on page 147
- “`scanf()` — Read Data” on page 330
- “`sscanf()` — Read Data” on page 355
- “`swprintf()` — Format and Write Wide Characters to Buffer” on page 406
- “`swscanf()` — Read Wide Character Data” on page 407
- “`vfprintf()` — Read Formatted Data” on page 427
- “`vfwscanf()` — Read Formatted Wide Character Data” on page 430
- “`vscanf()` — Read Formatted Data” on page 433
- “`vsscanf()` — Read Formatted Data” on page 437
- “`vswscanf()` — Read Formatted Wide Character Data” on page 441
- “`vwscanf()` — Read Formatted Wide Character Data” on page 445
- “`wprintf()` — Format Data as Wide Characters and Print” on page 503
- “`wscanf()` — Read Data Using Wide-Character Format String” on page 504
- “`<wchar.h>`” on page 18

---

## `vprintf()` — Print Argument Data

### Format

```
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char *format, va_list arg_ptr);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Locale Sensitive:** The behavior of this function might be affected by the `LC_CTYPE` and `LC_NUMERIC` categories of the current locale. The behavior might also be affected by the `LC_UNI_CTYPE` category of the current locale if `LOCALETYPE(*LOCALEUCS2)` or `LOCALETYPE(*LOCALEUTF)` is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `vprintf()` function formats and prints a series of characters and values to `stdout`. The `vprintf()` function works just like the `printf()` function, except that `arg_ptr` points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by `va_start` for each call. In contrast, the `printf()` function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

The `vprintf()` function converts each entry in the argument list according to the corresponding format specifier in `format`. The `format` has the same form and function as the format string for the `printf()` function.

### Return Value

If successful, the `vprintf()` function returns the number of bytes written to `stdout`. If an error occurs, the `vprintf()` function returns a negative value. The value of `errno` may be set to `ETRUNC`.

### Example that uses `vprintf()`

This example prints out a variable number of strings to stdout.

```
#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("mylib/myfile", "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 should be similar to: *****/
Mon Tues Wed Thurs Fri
*/
```

### Related Information

- “printf() — Print Formatted Characters” on page 229
- “va\_arg() – va\_end() – va\_start() — Access Function Arguments” on page 423
- “vfprintf() — Print Argument Data to Stream” on page 425
- “vsprintf() — Print Argument Data to Buffer” on page 436
- “<stdarg.h>” on page 13
- “<stdio.h>” on page 15

---

## vscanf() — Read Formatted Data

### Format

```
#include <stdarg.h>
#include <stdio.h>
int vscanf(const char *format, va_list arg_ptr);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. The behavior might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The vscanf() function reads data from stdin into locations specified by a variable number of arguments. The vscanf() function works just like the scanf() function, except that *arg\_ptr* points to a list of arguments whose number can vary from call to call in the program. These arguments should be

initialized by `va_start` for each call. In contrast, the `scanf()` function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

Each argument must be a pointer to a variable with a type that corresponds to a type specifier in format-string. The *format* has the same form and function as the format string for the `scanf()` function.

## Return Value

The `vscanf()` function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is EOF for an attempt to read at end-of-file if no conversion was performed. A return value of 0 means that no fields were assigned.

## Example that uses `vscanf()`

This example uses the `vscanf()` function to read an integer, a floating-point value, a character, and a string from `stdin` and then displays these values.

```
#include <stdio.h>
#include <stdarg.h>
int vread(char *fmt, ...)
{
    int rc;
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    rc = vscanf(fmt, arg_ptr);
    va_end(arg_ptr);
    return(rc);
}

int main(void)
{
    int i, rc;
    float fp;
    char c, s[81];
    printf("Enter an integer, a real number, a character "
           "and a string : \n");
    rc = vread("%d %f %c %s", &i, &fp, &c, s);
    if (rc != 4)
        printf("Not all fields are assigned\n");
    else
    {
        printf("integer = %d\n", i);
        printf("real number = %f\n", fp);
        printf("character = %c\n", c);
        printf("string = %s\n",s);
    }
}
/***** If input is: 12 2.5 a yes, *****/
/***** then output should be similar to: *****/
Enter an integer, a real number, a character and a string :
integer = 12
real number = 2.500000
character = a
string = yes
*/
```

## Related Information

- “`fscanf()` — Read Formatted Data” on page 132
- “`fwprintf()` — Format Data as Wide Characters and Write to a Stream” on page 143
- “`fwscanf()` — Read Data from Stream Using Wide Character” on page 147

- “scanf() — Read Data” on page 330
- “sscanf() — Read Data” on page 355
- “swprintf() — Format and Write Wide Characters to Buffer” on page 406
- “swscanf() — Read Wide Character Data” on page 407
- “vfscanf() — Read Formatted Data” on page 427
- “vfwscanf() — Read Formatted Wide Character Data” on page 430
- “vscanf() — Read Formatted Data” on page 433
- “vsscanf() — Read Formatted Data” on page 437
- “vswscanf() — Read Formatted Wide Character Data” on page 441
- “vwscanf() — Read Formatted Wide Character Data” on page 445
- “wprintf() — Format Data as Wide Characters and Print” on page 503
- “wscanf() — Read Data Using Wide-Character Format String” on page 504
- “<wchar.h>” on page 18

---

## vsnprintf() — Print Argument Data to Buffer

### Format

```
#include <stdarg.h>
#include <stdio.h>
int vsnprintf(char *target-string, size_t n, const char *format, va_list arg_ptr);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. The behavior might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The vsnprintf() function formats and stores a series of characters and values in the buffer target-string. The vsnprintf() function works just like the sprintf() function, except that arg\_ptr points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by the va\_start function for each call. In contrast, the sprintf() function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

The vsnprintf() function converts each entry in the argument list according to the corresponding format specifier in format. The format has the same form and function as the format string for the printf() function.

### Return Value

The vsnprintf() function returns the number of bytes that are written in the array, not counting the ending null character.

### Example that uses vsnprintf()

This example assigns a variable number of strings to string and prints the resultant string.

```
#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);
    vsnprintf(string, 8, fmt, arg_ptr);
    va_end(arg_ptr);
}

/***** Output should be similar to: *****/

The string is: Sat Su
*/

```

### Related Information

- “printf() — Print Formatted Characters” on page 229
- “sprintf() — Print Formatted Data to Buffer” on page 352
- “snprintf() — Print Formatted Data to Buffer” on page 350
- “va\_arg() – va\_end() – va\_start() — Access Function Arguments” on page 423
- “vfprintf() — Print Argument Data to Stream” on page 425
- “vsprintf() — Print Argument Data to Buffer”
- “<stdarg.h>” on page 13
- “<stdio.h>” on page 15

---

## vsprintf() — Print Argument Data to Buffer

### Format

```

#include <stdarg.h>
#include <stdio.h>
int vsprintf(char *target-string, const char *format, va_list arg_ptr);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. The behavior might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The vsprintf() function formats and stores a series of characters and values in the buffer *target-string*. The vsprintf() function works just like the sprintf() function, except that *arg\_ptr* points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by the va\_start function for each call. In contrast, the sprintf() function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

The `vsprintf()` function converts each entry in the argument list according to the corresponding format specifier in *format*. The *format* has the same form and function as the format string for the `printf()` function.

### Return Value

If successful, the `vsprintf()` function returns the number of bytes written to *target-string*. If an error occurs, the `vsprintf()` function returns a negative value.

### Example that uses `vsprintf()`

This example assigns a variable number of strings to *string* and prints the resultant string.

```
#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 should be similar to: *****/

The string is: Sat Sun Mon
*/
```

### Related Information

- “`printf()` — Print Formatted Characters” on page 229
- “`sprintf()` — Print Formatted Data to Buffer” on page 352
- “`va_arg()` – `va_end()` – `va_start()` — Access Function Arguments” on page 423
- “`vfprintf()` — Print Argument Data to Stream” on page 425
- “`vprintf()` — Print Argument Data” on page 432
- “`vswprintf()` — Format and Write Wide Characters to Buffer” on page 439
- “`<stdarg.h>`” on page 13
- “`<stdio.h>`” on page 15

---

## vsscanf() — Read Formatted Data

### Format

```
#include <stdarg.h>
#include <stdio.h>
int vsscanf(const char *buffer, const char *format, va_list arg_ptr);
```

Language Level: ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. The behavior might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

### Description

The `vsscanf()` function reads data from a buffer into locations specified by a variable number of arguments. The `vsscanf()` function works just like the `sscanf()` function, except that *arg\_ptr* points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by *va\_start* for each call. In contrast, the `sscanf()` function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

Each argument must be a pointer to a variable with a type that corresponds to a type specifier in format-string. The *format* has the same form and function as the format string for the `scanf()` function.

### Return Value

The `vsscanf()` function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is EOF for an attempt to read at end-of-file if no conversion was performed. A return value of 0 means that no fields were assigned.

### Example that uses `vsscanf()`

This example uses `vsscanf()` to read various data from the string *tokenstring* and then displays that data.

```

#include <stdio.h>
#include <stdarg.h>
#include <stddef.h>

int vread(const char *buffer, char *fmt, ...)
{
    int rc;
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    rc = vsscanf(buffer, fmt, arg_ptr);
    va_end(arg_ptr);
    return(rc);
}

int main(void)
{
    char *tokenstring = "15 12 14";
    wchar_t * idestring = L"ABC Z";
    wchar_t ws[81];
    wchar_t wc;
    int i;
    float fp;
    char s[81];
    char c;
    /* Input various data */
    /* In the first invocation of vsscanf, the format string is */
    /* "%s %c%d%f". If there were no space between %s and %c, */
    /* vsscanf would read the first character following the */
    /* string, which is a blank space. */
    vread(tokenstring, "%s %c%d%f", s, &c, &i, &fp);
    vread((char *) idestring, "%S %C", ws,&wc);
    /* Display the data */
    printf("\nstring = %s\n",s);
    printf("character = %c\n",c);
    printf("integer = %d\n",i);
    printf("floating-point number = %f\n",fp);
    printf("wide-character string = %S\n", ws);
    printf("wide-character = %C\n", wc);
}

/***** Output should be similar to: *****/
string = 15
character = 1
integer = 2
floating-point number = 14.000000
wide-character string = ABC
wide-character = Z
*****/

```

## Related Information

- “`fscanf()` — Read Formatted Data” on page 132
- “`fwscanf()` — Read Data from Stream Using Wide Character” on page 147
- “`scanf()` — Read Data” on page 330
- “`sscanf()` — Read Data” on page 355
- “`sprintf()` — Print Formatted Data to Buffer” on page 352
- “`<stdio.h>`” on page 15
- “`swscanf()` — Read Wide Character Data” on page 407
- “`wscanf()` — Read Data Using Wide-Character Format String” on page 504

---

## vswprintf() — Format and Write Wide Characters to Buffer

### Format

```
#include <stdarg.h>
#include <wchar.h>
int vswprintf(wchar_t *wcsbuffer, size_t n, const wchar_t
             *format, va_list argptr);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

- | **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. It might also be affected by the LC\_UNI\_CTYPE and LC\_UNI\_NUMERIC categories of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.
- | **Wide Character Function:** See “Wide Characters” on page 527 for more information.

## Description

The `vswprintf()` function formats and stores a series of wide characters and values in the buffer `wcsbuffer`. The `vswprintf()` function works just like the `swprintf()` function, except that `argptr` points to a list of wide-character arguments whose number can vary from call to call. These arguments should be initialized by `va_start` for each call. In contrast, the `swprintf()` function can have a list of arguments, but the number of arguments in that list are fixed when you compile in the program.

The value `n` specifies the maximum number of wide characters to be written, including the ending null character. The `vswprintf()` function converts each entry in the argument list according to the corresponding wide-character format specifier in `format`. The format has the same form and function as the format string for the `printf()` function, with the following exceptions:

- `%c` (without an `l` prefix) converts an integer argument to `wchar_t`, as if by calling the `mbtowc()` function.
- `%lc` converts a `wint_t` to `wchar_t`.
- `%s` (without an `l` prefix) converts an array of multibyte characters to an array of `wchar_t`, as if by calling the `mbrtowc()` function. The array is written up to, but not including, the ending null character, unless the precision specifies a shorter output.
- `%ls` writes an array of `wchar_t`. The array is written up to, but not including, the ending null character, unless the precision specifies a shorter output.

A null wide character is added to the end of the wide characters written; the null wide character is not counted as part of the returned value. If copying takes place between objects that overlap, the behavior is undefined.

## Return Value

The `vswprintf()` function returns the number of bytes written in the array, not counting the ending null wide character.

## Example that uses `vswprintf()`

This example creates a function `vout()` that takes a variable number of wide-character arguments and uses `vswprintf()` to print them to `wcstr`.

```

#include <stdio.h>
#include <stdarg.h>
#include <wchar.h>

wchar_t *format3 = L"%1s %d %1s";
wchar_t *format5 = L"%1s %d %1s %d %1s";

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);
    return;
}

int main(void)
{
    wchar_t wcstr[100];

    vout(wcstr, 100, format3, L"ONE", 2L, L"THREE");
    printf("%1s\n", wcstr);
    vout(wcstr, 100, format5, L"ONE", 2L, L"THREE", 4L, L"FIVE");
    printf("%1s\n", wcstr);
    return 0;

    /*****
    The output should be similar to:

    ONE 2 THREE
    ONE 2 THREE 4 FIVE
    *****/
}

```

## Related Information

- “swprintf() — Format and Write Wide Characters to Buffer” on page 406
- “vfprintf() — Print Argument Data to Stream” on page 425
- “vprintf() — Print Argument Data” on page 432
- “vsprintf() — Print Argument Data to Buffer” on page 436
- “<stdarg.h>” on page 13
- “<wchar.h>” on page 18

---

## vswscanf() — Read Formatted Wide Character Data

### Format

```

#include <stdarg.h>
#include <wchar.h>

```

```

int vswscanf(const wchar_t *buffer, const wchar_t *format, va_list arg_ptr);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. It might also be affected by the LC\_UNI\_CTYPE and LC\_UNI\_NUMERIC categories of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is

| specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is  
| specified on the compilation command. For more information, see “Understanding CCSIDs and Locales”  
| on page 524.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `vswscanf()` function reads wide data from a buffer into locations specified by a variable number of arguments. The `vswscanf()` function works just like the `swscanf()` function, except that *arg\_ptr* points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by `va_start` for each call. In contrast, the `swscanf()` function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

Each argument must be a pointer to a variable with a type that corresponds to a type specifier in format-string. The *format* has the same form and function as the format string for the `swscanf()` function.

### Return Value

The `vswscanf()` function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is EOF for an attempt to read at end-of-file if no conversion was performed. A return value of 0 means that no fields were assigned.

### Example that uses `vswscanf()`

This example uses the `vswscanf()` function to read various data from the string *tokenstring* and then displays that data.

```

#include <stdio.h>
#include <stdarg.h>
#include <wchar.h>
int vread(const wchar_t *buffer, wchar_t *fmt, ...)
{
    int rc;
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    rc = vswscanf(buffer, fmt, arg_ptr);
    va_end(arg_ptr);
    return(rc);
}
int main(void)
{
    wchar_t *tokenstring = L"15 12 14";
    char s[81];
    char c;
    int i;
    float fp;

    /* Input various data */

    vread(tokenstring, L"%s %c%d%f", s, &c, &i, &fp);

    /* Display the data */
    printf("\nstring = %s\n",s);
    printf("character = %c\n",c);
    printf("integer = %d\n",i);
    printf("floating-point number = %f\n",fp);
}
/***** Output should be similar to: *****/
string = 15
character = 1
integer = 2
floating-point number = 14.000000
*****/

```

### Related Information

- “fscanf() — Read Formatted Data” on page 132
- “scanf() — Read Data” on page 330
- “fwscanf() — Read Data from Stream Using Wide Character” on page 147
- “wscanf() — Read Data Using Wide-Character Format String” on page 504
- “sscanf() — Read Data” on page 355
- “sprintf() — Print Formatted Data to Buffer” on page 352
- “swscanf() — Read Wide Character Data” on page 407
- “<wchar.h>” on page 18

---

## vwprintf() — Format Argument Data as Wide Characters and Print

### Format

```

#include <stdarg.h>
#include <wchar.h>
int vwprintf(const wchar_t *format, va_list arg);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. It might also be affected by the LC\_UNI\_CTYPE and LC\_UNI\_NUMERIC categories of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified on the compilation command.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

## Description

The `wvprintf()` function is equivalent to the `wprintf()` function, except that the variable argument list is replaced by *arg*, which the `va_start` macro (and possibly subsequent `va_arg` calls) will have initialized. The `wvprintf()` function does not invoke the `va_end` macro.

## Return Value

The `wvprintf()` function returns the number of wide characters transmitted. If an output error occurred, `wvprintf()` returns a negative value.

## Example that uses `wvprintf()`

This example prints the wide character *a*. The printing is done from the `vout()` function, which takes a variable number of arguments and uses the `wvprintf()` function to print them to `stdout`.

```
#include <wchar.h>
#include <stdarg.h>
#include <locale.h>

void vout (wchar_t *fmt, ...);

const char ifs_path[] = "tmp/mytest";

int main(void)    {
    FILE *stream;
    wchar_t format[] = L"%lc";
    setlocale(LC_ALL, "POSIX");
    vout (format, L'a');
    return(0);

/* A long a is written to stdout, if stdout is written to the screen
   it may get converted back to a single byte 'a'. */
}

void vout (wchar_t *fmt, ...) {

    va_list arg_ptr;
    va_start (arg_ptr, fmt);
    wvprintf (fmt, arg_ptr);
    va_end (arg_ptr);
}
```

## Related Information

- “`printf()` — Print Formatted Characters” on page 229
- “`vfprintf()` — Print Argument Data to Stream” on page 425
- “`vprintf()` — Print Argument Data” on page 432

- “btowc() — Convert Single Byte to Wide Character” on page 53
- “mbrtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 201
- “fwprintf() — Format Data as Wide Characters and Write to a Stream” on page 143
- “vswprintf() — Format and Write Wide Characters to Buffer” on page 439
- “vfwprintf() — Format Argument Data as Wide Characters and Write to a Stream” on page 428
- “<stdarg.h>” on page 13
- “<wchar.h>” on page 18

---

## vwscanf() — Read Formatted Wide Character Data

### Format

```
#include <stdarg.h>
#include <stdio.h>
```

```
int vwscanf(const wchar_t *format, va_list arg_ptr);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale. It might also be affected by the LC\_UNI\_CTYPE and LC\_UNI\_NUMERIC categories of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified on the compilation command.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `vwscanf()` function reads data from `stdin` into locations specified by a variable number of arguments. The `vwscanf()` function works just like the `wscanf()` function, except that `arg_ptr` points to a list of arguments whose number can vary from call to call in the program. These arguments should be initialized by `va_start` for each call. In contrast, the `wscanf()` function can have a list of arguments, but the number of arguments in that list is fixed when you compile the program.

Each argument must be a pointer to a variable with a type that corresponds to a type specifier in format-string. The *format* has the same form and function as the format string for the `wscanf()` function.

### Return Value

The `vwscanf()` function returns the number of fields that were successfully converted and assigned. The return value does not include fields that were read but not assigned. The return value is EOF for an attempt to read at end-of-file if no conversion was performed. A return value of 0 means that no fields were assigned.

### Example that uses `vwscanf()`

This example scans various types of data from `stdin`.

```

#include <stdio.h>
#include <stdarg.h>

int vread(wchar_t *fmt, ...)
{
    int rc;
    va_list arg_ptr;
    va_start(arg_ptr, fmt);
    rc = vwscanf(fmt, arg_ptr);
    va_end(arg_ptr);
    return(rc);
}

int main(void)
{
    int i, rc;
    float fp;
    char c, s[81];
    printf("Enter an integer, a real number, a character "
"and a string : \n");
    rc = vread(L"%d %f %c %s",&i,&fp,&c, s);
    if (rc != 4)
        printf("Not all fields are assigned\n");
    else
    {
        printf("integer = %d\n", i);
        printf("real number = %f\n", fp);
        printf("character = %c\n", c);
        printf("string = %s\n",s);
    }
}
/***** If input is: 12 2.5 a yes, *****/
/***** then output should be similar to: *****/
Enter an integer, a real number, a character and a string :
integer = 12
real number = 2.500000
character = a
string = yes
*/

```

### Related Information

- “fscanf() — Read Formatted Data” on page 132
- “scanf() — Read Data” on page 330
- “sscanf() — Read Data” on page 355
- “swscanf() — Read Wide Character Data” on page 407
- “fwscanf() — Read Data from Stream Using Wide Character” on page 147
- “wscanf() — Read Data Using Wide-Character Format String” on page 504
- “sprintf() — Print Formatted Data to Buffer” on page 352
- “<stdio.h>” on page 15

---

## wcrtomb() — Convert a Wide Character to a Multibyte Character (Restartable)

### Format

```

#include <wchar.h>
size_t wcrtomb (char *s, wchar_t wc, mbstate_t *ps);

```

**Language Level:** ANSI

**Threadsafe:** Yes, except when *ps* is NULL.

- | **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. The behavior might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.
- | **Wide Character Function:** See “Wide Characters” on page 527 for more information.

## Description

This function is the restartable version of the `wctomb()` function.

The `wcrtomb()` function converts a wide character to a multibyte character.

If *s* is a null pointer, the `wcrtomb()` function determines the number of bytes necessary to enter the initial shift state (zero if encodings are not state-dependent or if the initial conversion state is described). The resulting state described will be the initial conversion state.

If *s* is not a null pointer, the `wcrtomb()` function determines the number of bytes needed to represent the multibyte character that corresponds to the wide character given by *wc* (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 will be stored. If *wc* is a null wide character, the resulting state described will be the initial conversions state.

This function differs from its corresponding internal-state multibyte character function in that it has an extra parameter, *ps* of type pointer to `mbstate_t` that points to an object that can completely describe the current conversion state of the associated multibyte character sequence. If *ps* is `NULL`, an internal static variable will be used to keep track of the conversion state. Using the internal static variable is not threadsafe.

## Return Value

If *s* is a null pointer, the `wcrtomb()` function returns the number of bytes needed to enter the initial shift state. The value returned will not be greater than that of the `MB_CUR_MAX` macro.

If *s* is not a null pointer, the `wcrtomb()` function returns the number of bytes stored in the array object (including any shift sequences) when *wc* is a valid wide character; otherwise (when *wc* is not a valid wide character), an encoding error occurs, the value of the macro `EILSEQ` shall be stored in `errno` and `-1` will be returned, but the conversion state will be unchanged.

If a conversion error occurs, `errno` may be set to `ECONVERT`.

## Examples that use `wcrtomb()`

This program is compiled with `LOCALETYPE(*LOCALE)` and `SYSIFCOPT(*IFSIO)`:

```
#include <stdio.h>
#include <locale.h>
#include <wchar.h>
#include <errno.h>

#define STRLENGTH 10
#define LOCNAME "qsys.lib/JA_JP.locale"
#define LOCNAME_EN "qsys.lib/EN_US.locale"

int main(void)
{
    char    string[STRLENGTH];
    int length, sl = 0;
```

```

wchar_t wc = 0x4171;
wchar_t wc2 = 0x00C1;
wchar_t wc_string[10];
mbstate_t ps = 0;
memset(string, '\\0', STRLENGTH);
wc_string[0] = 0x00C1;
wc_string[1] = 0x4171;
wc_string[2] = 0x4172;
wc_string[3] = 0x00C2;
wc_string[4] = 0x0000;
/* In this first example we will convert a wide character */
/* to a single byte character. We first set the locale */
/* to a single byte locale. We choose a locale with */
/* CCSID 37. For single byte cases the state will always */
/* remain in the initial state 0 */

if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
    printf("setlocale failed.\n");

length = wctomb(string, wc, &ps);

/* In this case since wc > 256 hex, length is -1 and */
/* errno is set to EILSEQ (3492) */
printf("errno = %d, length = %d\n\n", errno, length);

length = wctomb(string, wc2, &ps);

/* In this case wc2 00C1 is converted to C1 */

printf("string = %s\n\n", string);

/* Now lets try a multibyte example. We first must set the */
/* locale to a multibyte locale. We choose a locale with */
/* CCSID 5026 */

if (setlocale(LC_ALL, LOCNAME) == NULL)
    printf("setlocale failed.\n");

length = wctomb(string, wc_string[0], &ps);

/* The first character is < 256 hex so is converted to */
/* single byte and the state is still the initial state 0 */

printf("length = %d, state = %d\n\n", length, ps);

s1 += length;

length = wctomb(&string[s1], wc_string[1], &ps);

/* The next character is > 256 hex so we get a shift out */
/* 0x0e followed by the double byte character. State is */
/* changed to double byte state. Length is 3. */

printf("length = %d, state = %d\n\n", length, ps);

s1 += length;

length = wctomb(&string[s1], wc_string[2], &ps);

/* The next character is > 256 hex so we get another */
/* double byte character. The state is left in */
/* double byte state. Length is 2. */

printf("length = %d, state = %d\n\n", length, ps);

s1 += length;

```

```

length = wctomb(&string[s1], wc_string[3], &ps);

/* The next character is < 256 hex so we close off the */
/* double byte characters with a shift in 0x0f and then */
/* get a single byte character. Length is 2. */
/* The hex look at string would now be: */
/* C10E417141720FC2 */
/* You would need a device capable of displaying multibyte */
/* characters to see this string. */

printf("length = %d, state = %d\n\n", length, ps);

/* In the last example we will show what happens if NULL */
/* is passed in for the state. */
memset(string, '\0', STRLENGTH);

length = wctomb(string, wc_string[1], NULL);

/* The second character is > 256 hex so a shift out */
/* followed by the double character is produced but since */
/* the state is NULL, the double byte character is closed */
/* off with a shift in right away. So string we look */
/* like this: 0E41710F and length is 4 and the state is */
/* left in the initial state. */

printf("length = %d, state = %d\n\n", length, ps);
}
/* The output should look like this:

errno = 3492, length = -1

string = A

length = 1, state = 0

length = 3, state = 2

length = 2, state = 2

length = 2, state = 0

length = 4, state = 0

*/

```

This program is compiled with `LOCALETYPE(*LOCALEUCS2)` and `SYSIFCOPT(*IFSIO)`:

```

#include <stdio.h>
#include <locale.h>
#include <wchar.h>
#include <errno.h>

#define STRLENGTH 10
#define LOCNAME "qsys.lib/JA_JP.locale"
#define LOCNAME_EN "qsys.lib/EN_US.locale"

int main(void)
{
    char    string[STRLENGTH];
    int length, s1 = 0;
    wchar_t wc = 0x4171;
    wchar_t wc2 = 0x0041;
    wchar_t wc_string[10];
    mbstate_t ps = 0;
    memset(string, '\0', STRLENGTH);
    wc_string[0] = 0x0041;
    wc_string[1] = 0xFF31;

```

```

wc_string[2] = 0xFF32;
wc_string[3] = 0x0042;
wc_string[4] = 0x0000;
/* In this first example we will convert a UNICODE character */
/* to a single byte character. We first set the locale */
/* to a single byte locale. We choose a locale with */
/* CCSID 37. For single byte cases the state will always */
/* remain in the initial state 0 */

if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
    printf("setlocale failed.\n");

length = wctomb(string, wc2, &ps);

/* In this case wc2 0041 is converted to C1 */
/* 0041 is UNICODE A, C1 is CCSID 37 A */

printf("string = %s\n\n", string);

/* Now lets try a multibyte example. We first must set the */
/* locale to a multibyte locale. We choose a locale with */
/* CCSID 5026 */

if (setlocale(LC_ALL, LOCNAME) == NULL)
    printf("setlocale failed.\n");

length = wctomb(string, wc_string[0], &ps);

/* The first character UNICODE character is converted to a */
/* single byte and the state is still the initial state 0 */

printf("length = %d, state = %d\n\n", length, ps);

s1 += length;

length = wctomb(&string[s1], wc_string[1], &ps);

/* The next UNICODE character is converted to a shift out */
/* 0x0e followed by the double byte character. State is */
/* changed to double byte state. Length is 3. */

printf("length = %d, state = %d\n\n", length, ps);

s1 += length;

length = wctomb(&string[s1], wc_string[2], &ps);

/* The UNICODE character is converted to another */
/* double byte character. The state is left in */
/* double byte state. Length is 2. */

printf("length = %d, state = %d\n\n", length, ps);

s1 += length;

length = wctomb(&string[s1], wc_string[3], &ps);

/* The next UNICODE character converts to single byte so */
/* we close off the */
/* double byte characters with a shiftin 0x0f and then */
/* get a single byte character. Length is 2. */
/* The hex look at string would now be: */
/* C10E42D842D90FC2 */
/* You would need a device capable of displaying multibyte */
/* characters to see this string. */

printf("length = %d, state = %d\n\n", length, ps);

```

```

}
/* The output should look like this:

string = A
length = 1, state = 0
length = 3, state = 2
length = 2, state = 2
length = 2, state = 0
*/

```

### Related Information

- “`mblen()` — Determine Length of a Multibyte Character” on page 197
- “`mbrlen()` — Determine Length of a Multibyte Character (Restartable)” on page 199
- “`mbrtowc()` — Convert a Multibyte Character to a Wide Character (Restartable)” on page 201
- “`mbsrtowcs()` — Convert a Multibyte String to a Wide Character String (Restartable)” on page 206
- “`wcsrtombs()` — Convert Wide Character String to Multibyte String (Restartable)” on page 473
- “`wctomb()` — Convert Wide Character to Multibyte Character” on page 492
- “`<wchar.h>`” on page 18

---

## wcscat() — Concatenate Wide-Character Strings

### Format

```

| #include <wchar.h>
| wchar_t *wcscat(wchar_t *string1, const wchar_t *string2);

```

**Language Level:** XPG4

**Threadsafe:** Yes.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `wcscat()` function 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-ended `wchar_t` strings. The string arguments to this function should contain a `wchar_t` null character marking the end of the string. Boundary checking is not performed.

### Return Value

The `wcscat()` function returns a pointer to the concatenated *string1*.

### Example that uses `wcscat()`

This example creates the wide character string “computer program” using the `wcscat()` function.

```

| #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 = wcschr( buffer1, string );
|     printf( "buffer1 = %ls\n", buffer1 );
|
| }
|
| /***** Output should be similar to: *****/
|
| buffer1 = computer program
| *****/

```

### Related Information

- “`strcat()` — Concatenate Strings” on page 358
- “`strncat()` — Concatenate Strings” on page 377
- “`wcschr()` — Search for Wide Character”
- “`wscmp()` — Compare Wide-Character Strings” on page 453
- “`wscpy()` — Copy Wide-Character Strings” on page 456
- “`wscspn()` — Find Offset of First Wide-Character Match” on page 457
- “`wcslen()` — Calculate Length of Wide-Character String” on page 461
- “`wcsncat()` — Concatenate Wide-Character Strings” on page 463
- “`<wchar.h>`” on page 18

---

## wcschr() — Search for Wide Character

### Format

```

| #include <wchar.h>
| wchar_t *wcschr(const wchar_t *string, wchar_t character);

```

**Language Level:** XPG4

**Threadsafe:** Yes.

**Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `wcschr()` function searches the wide-character *string* for the occurrence of *character*. The *character* can be a `wchar_t` null character (`\0`); the `wchar_t` null character at the end of *string* is included in the search.

The `wcschr()` function operates on null-ended `wchar_t` strings. The string argument to this function should contain a `wchar_t` null character marking the end of the string.

### Return Value

The `wcschr()` function returns a pointer to the first occurrence of *character* in *string*. If the character is not found, a NULL pointer is returned.

### Example that uses wcschr()

This example finds the first occurrence of the character "p" in the wide-character string "computer program".

```
| #include <stdio.h>
| #include <wchar.h>
|
| #define SIZE 40
|
| int main(void)
| {
|     wchar_t buffer1[SIZE] = L"computer program";
|     wchar_t * ptr;
|     wchar_t ch = L'p';
|
|     ptr = wcschr( buffer1, ch );
|     printf( "The first occurrence of %lc in '%ls' is '%ls'\n",
|            ch, buffer1, ptr );
|
| }
|
| /***** Output should be similar to: *****/
|
| The first occurrence of p in 'computer program' is 'puter program'
| */
```

### Related Information

- "strchr() — Search for Character" on page 359
- "strcspn() — Find Offset of First Character Match" on page 365
- "strpbrk() — Find Characters in String" on page 384
- "strrchr() — Locate Last Occurrence of Character in String" on page 389
- "strspn() — Find Offset of First Non-matching Character" on page 390
- "wscat() — Concatenate Wide-Character Strings" on page 451
- "wscmp() — Compare Wide-Character Strings"
- "wcscpy() — Copy Wide-Character Strings" on page 456
- "wscspn() — Find Offset of First Wide-Character Match" on page 457
- "wcslen() — Calculate Length of Wide-Character String" on page 461
- "wcsncmp() — Compare Wide-Character Strings" on page 464
- "wcpbrk() — Locate Wide Characters in String" on page 468
- "wcsrchr() — Locate Last Occurrence of Wide Character in String" on page 471
- "wcssp() — Find Offset of First Non-matching Wide Character" on page 474
- "wswcs() — Locate Wide-Character Substring" on page 488
- "<wchar.h>" on page 18

---

## wscmp() — Compare Wide-Character Strings

### Format

```
| #include <wchar.h>
| int wscmp(const wchar_t *string1, const wchar_t *string2);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

## Description

The `wscmp()` function compares two wide-character strings. The `wscmp()` function operates on null-ended `wchar_t` strings; string arguments to this function should contain a `wchar_t` null character marking the end of the string. Boundary checking is not performed when a string is added to or copied.

## Return Value

The `wscmp()` function returns a value indicating the relationship between the two strings, as follows:

### Value Meaning

#### Less than 0

*string1* less than *string2*

0 *string1* identical to *string2*

#### Greater than 0

*string1* greater than *string2*.

## Example that uses `wscmp()`

This example compares the wide-character string *string1* to *string2* using `wscmp()`.

```
| #include <stdio.h>
| #include <wchar.h>
|
| int main(void)
| {
|     int result;
|     wchar_t string1[] = L"abcdef";
|     wchar_t string2[] = L"abcdefg";
|
|     result = wscmp( 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 should be similar to: *****/
|
| "abcdef" is less than "abcdefg"
| */
```

## Related Information

- “`strcmp()` — Compare Strings” on page 360
- “`strncmp()` — Compare Strings” on page 379
- “`wscat()` — Concatenate Wide-Character Strings” on page 451
- “`wcschr()` — Search for Wide Character” on page 452
- “`wscpy()` — Copy Wide-Character Strings” on page 456
- “`wscspn()` — Find Offset of First Wide-Character Match” on page 457
- “`wcslen()` — Calculate Length of Wide-Character String” on page 461
- “`wcsncmp()` — Compare Wide-Character Strings” on page 464
- “`__wcsicmp()` — Compare Wide Character Strings without Case Sensitivity” on page 460

- “`__wcsnicmp()` — Compare Wide Character Strings without Case Sensitivity” on page 467
- “`<wchar.h>`” on page 18

---

## wscoll() —Language Collation String Comparison

### Format

```
#include <wchar.h>
int wscoll (const wchar_t *wcs1, const wchar_t *wcs2);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

**Locale Sensitive:** The behavior of this function might be affected by the `LC_COLLATE` category of the current locale if `LOCALETYPE(*LOCALE)` is specified on the compilation command. The behavior of this function might also be affected by the `LC_UNI_COLLATE` category of the current locale if `LOCALETYPE(*LOCALEUCS2)` or `LOCALETYPE(*LOCALEUTF)` is specified on the compilation command. This function is not available when `LOCALETYPE(*CLD)` is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

**Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `wscoll()` function compares the wide-character strings pointed to by `wcs1` and `wcs2`, both interpreted as appropriate to the `LC_COLLATE` category of the current locale (or the `LC_UNI_COLLATE` category if a UNICODE `LOCALETYPE` was specified).

### Return Value

The `wscoll()` function returns an integer value indicating the relationship between the strings, as follows:

#### Value Meaning

##### Less than 0

*wcs1* less than *wcs2*

**0** *wcs1* equivalent to *wcs2*

##### Greater than 0

*wcs1* greater than *wcs2*

If *wcs1* or *wcs2* contain characters outside the domain of the collating sequence, the `wscoll()` function sets `errno` to `EINVAL`. If an error occurs, the `wscoll()` function sets `errno` to a nonzero value. There is no error return value.

### Example that uses `wscoll()`

This example uses the default locale.

```

#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("%S\\n" is identical to %S\\n", wcs1, wcs2);
    else if ( result < 0)
        printf("%S\\n" is less than %S\\n", wcs1, wcs2);
    else
        printf("%S\\n" is greater than %S\\n", wcs1, wcs2);
}

```

### Related Information

- “strcoll() — Compare Strings” on page 363
- “setlocale() — Set Locale” on page 339
- “<wchar.h>” on page 18

---

## wcscpy() — Copy Wide-Character Strings

### Format

```

| #include <wchar.h>
| wchar_t *wcscpy(wchar_t *string1, const wchar_t *string2);

```

**Language Level:** XPG4

**Threadsafe:** Yes.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `wcscpy()` function copies the contents of *string2* (including the ending `wchar_t` null character) into *string1*.

The `wcscpy()` function operates on null-ended `wchar_t` strings; string arguments to this function should contain a `wchar_t` null character marking the end of the string. Only *string2* needs to contain a null character. Boundary checking is not performed.

### Return Value

The `wcscpy()` function returns a pointer to *string1*.

### Example that uses `wcscpy()`

This example copies the contents of source to destination.

```

| #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 = wcsncpy( destination, source );
|     printf( "After wcsncpy, destination becomes \"%ls\\n\"", destination );
| }
|
| /***** Output should be similar to: *****/
|
| destination is originally = "And this is the destination string"
| After wcsncpy, destination becomes "This is the source string"
| */

```

### Related Information

- “`strcpy()` — Copy Strings” on page 364
- “`strncpy()` — Copy Strings” on page 380
- “`wscat()` — Concatenate Wide-Character Strings” on page 451
- “`wcschr()` — Search for Wide Character” on page 452
- “`wscmp()` — Compare Wide-Character Strings” on page 453
- “`wscspn()` — Find Offset of First Wide-Character Match”
- “`wcslen()` — Calculate Length of Wide-Character String” on page 461
- “`wcncpy()` — Copy Wide-Character Strings” on page 466
- “`<wchar.h>`” on page 18

---

## wcscspn() — Find Offset of First Wide-Character Match

### Format

```

| #include <wchar.h>
| size_t wcscspn(const wchar_t *string1, const wchar_t *string2);

```

**Language Level:** XPG4

**Threadsafe:** Yes.

**Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `wcscspn()` function determines the number of `wchar_t` characters in the initial segment of the string pointed to by *string1* that do not appear in the string pointed to by *string2*.

The `wcscspn()` function operates on null-ended `wchar_t` strings; string arguments to this function should contain a `wchar_t` null character marking the end of the string.

### Return Value

The `wcscspn()` function returns the number of `wchar_t` characters in the segment.

## Example that uses wcsncpy()

This example uses wcsncpy() 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", wcsncpy( string, substring),
|           string, substring );
| }
|
| /***** Output should be similar to: *****/
|
| The first 10 characters in the string "This is the source string" are not
| in the string "axle"
| */
```

## Related Information

- “strncpy() — Find Offset of First Character Match” on page 365
- “strspn() — Find Offset of First Non-matching Character” on page 390
- “wscat() — Concatenate Wide-Character Strings” on page 451
- “wcschr() — Search for Wide Character” on page 452
- “wscmp() — Compare Wide-Character Strings” on page 453
- “wscpy() — Copy Wide-Character Strings” on page 456
- “wcslen() — Calculate Length of Wide-Character String” on page 461
- “wcsspn() — Find Offset of First Non-matching Wide Character” on page 474
- “wswcs() — Locate Wide-Character Substring” on page 488
- “<wchar.h>” on page 18

---

## wcsftime() — Convert to Formatted Date and Time

### Format

```
#include <wchar.h>
size_t wcsftime(wchar_t *wdest, size_t maxsize,
                const wchar_t *format, const struct tm *timeptr);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE, LC\_TIME, and LC\_TOD categories of the current locale if LOCALETYPE(\*LOCALE) is specified on the compilation command. The behavior of this function might also be affected by the LC\_UNI\_CTYPE, LC\_UNI\_TIME, and LC\_UNI\_TOD categories of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

**Wide Character Function:** See “Wide Characters” on page 527 for more information.

## Description

The `wcsftime()` function converts the time and date specification in the `timeptr` structure into a wide-character string. It then stores the null-ended string in the array pointed to by `wdest` according to the format string pointed to by `format`. The `maxsize` value specifies the maximum number of wide characters that can be copied into the array. This function is equivalent to `strftime()`, except that it uses wide characters.

The `wcsftime()` function works just like the `strftime()` function, except that it uses wide characters. The format string is a wide-character character string that contains:

- Conversion-specification characters.
- Ordinary wide characters, which are copied into the array unchanged.

This function uses the time structure pointed to by `timeptr`, and if the specifier is locale sensitive, then it will also use the `LC_TIME` category of the current locale to determine the appropriate replacement value of each valid specifier. The time structure pointed to by `timeptr` is usually obtained by calling the `gmtime()` or `localtime()` function.

## Return Value

If the total number of wide characters in the resulting string, including the ending null wide character, does not exceed `maxsize`, `wcsftime()` returns the number of wide characters placed into `wdest`, not including the ending null wide character. Otherwise, the `wcsftime()` function returns 0 and the contents of the array are indeterminate.

If a conversion error occurs, `errno` may be set to `ECONVERT`.

## Example that uses `wcsftime()`

This example obtains the date and time using `localtime()`, formats the information with the `wcsftime()`, and prints the date and time.

```
#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), L" Today is %A,"
                 L" %b %d.\n Time: %I:%M %p", timeptr);
    printf("%d characters placed in string to make:\n\n%ls\n", rc, dest);
    return 0;

    /*****
    The output should be similar to:

    43 characters placed in string to make:

    Today is Thursday, Nov 10.
    Time: 04:56 PM
    *****/
}
```

## Related Information

- | • “ctime() — Convert Time to Character String” on page 71
- | • “ctime64() — Convert Time to Character String” on page 73
- | • “ctime64\_r() — Convert Time to Character String (Restartable)” on page 76
- | • “ctime\_r() — Convert Time to Character String (Restartable)” on page 74
- “gmtime() — Convert Time” on page 161
- | • “gmtime64() — Convert Time” on page 163
- | • “gmtime64\_r() — Convert Time (Restartable)” on page 167
- | • “gmtime\_r() — Convert Time (Restartable)” on page 165
- “localtime() — Convert Time” on page 185
- | • “localtime64() — Convert Time” on page 187
- | • “localtime64\_r() — Convert Time (Restartable)” on page 189
- | • “localtime\_r() — Convert Time (Restartable)” on page 188
- “strftime() — Convert Date/Time to String” on page 370
- “strptime()— Convert String to Date/Time” on page 385
- | • “time() — Determine Current Time” on page 411
- | • “time64() — Determine Current Time” on page 412
- “<wchar.h>” on page 18

---

## **\_\_wcsicmp() — Compare Wide Character Strings without Case Sensitivity**

### **Format**

```
#include <wchar.h>
int __wcsicmp(const wchar_t *string1, const wchar_t *string2);
```

**Language Level:** Extension

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale if LOCALETYPE(\*LOCALE) is specified on the compilation command. The behavior of this function might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### **Description**

The `__wcsicmp()` function compares *string1* and *string2* without sensitivity to case. All alphabetic wide characters in *string1* and *string2* are converted to lowercase before comparison. The function operates on null terminated wide character strings. The string arguments to the function are expected to contain a `wchar_t` null character (`L'\0'`) marking the end of the string.

### **Return Value**

The `__wcsicmp()` function returns a value indicating the relationship between the two strings as follows:

Table 10. Return values of `__wcsicmp()`

Value	Meaning
-------	---------

Table 10. Return values of `__wcsicmp()` (continued)

Less than 0	<i>string1</i> less than <i>string2</i>
0	<i>string1</i> equivalent to <i>string2</i>
Greater than 0	<i>string1</i> greater than <i>string2</i>

### Example that uses `__wcsicmp()`

This example uses `__wcsicmp()` to compare two wide character strings.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *str1 = L"STRING";
    wchar_t *str2 = L"string";
    int result;

    result = __wcsicmp(str1, str2);

    if (result == 0)
        printf("Strings compared equal.\n");
    else if (result < 0)
        printf("\'%%ls\' is less than \'%ls\'\n", str1, str2);
    else
        printf("\'%%ls\' is greater than \'%ls\'\n", str1, str2);

    return 0;
}

/***** The output should be similar to: *****/

Strings compared equal.

*****/
```

### Related Information

- “`strcmp()` — Compare Strings” on page 360
- “`strncmp()` — Compare Strings” on page 379
- “`wscat()` — Concatenate Wide-Character Strings” on page 451
- “`wcschr()` — Search for Wide Character” on page 452
- “`wscspn()` — Find Offset of First Wide-Character Match” on page 457
- “`wcslen()` — Calculate Length of Wide-Character String”
- “`wcsncmp()` — Compare Wide-Character Strings” on page 464
- “`__wcsnicmp()` — Compare Wide Character Strings without Case Sensitivity” on page 467
- “`<wchar.h>`” on page 18

---

## wcslen() — Calculate Length of Wide-Character String

### Format

```
| #include <wchar.h>
| size_t wcslen(const wchar_t *string);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `wcslen()` function computes the number of wide characters in the string pointed to by *string*.

### Return Value

The `wcslen()` function returns the number of wide characters in *string*, excluding the ending `wchar_t` null character.

### Example that uses `wcslen()`

This example computes the length of the wide-character string `string`.

```
| #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 should be similar to: *****/
|
| Length of "abcdef" is 6
| */
```

### Related Information

- “`mblen()` — Determine Length of a Multibyte Character” on page 197
- “`strlen()` — Determine String Length” on page 375
- “`wcsncat()` — Concatenate Wide-Character Strings” on page 463
- “`wcsncmp()` — Compare Wide-Character Strings” on page 464
- “`wcsncpy()` — Copy Wide-Character Strings” on page 466
- | • “`<wchar.h>`” on page 18

---

## **wcslocaleconv() — Retrieve Wide Locale Information**

### Format

```
#include <locale.h>
struct wcslconv *wcslocaleconv(void);
```

**Language Level:** Extended

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the `LC_UNI_NUMERIC` and `LC_UNI_MONETARY` categories of the current locale. This function is only available when `LOCALETYPE(*LOCALEUCS2)` or `LOCALETYPE(*LOCALEUTF)` is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `wcslocaleconv()` function is the same as the `localeconv` function, except that it returns a pointer to a `wcslconv` structure, which is the wide version of a `lconv` structure. These elements are determined by the `LC_UNI_MONETARY` and `LC_UNI_NUMERIC` categories of the current locale.

### Return Value

The `wcslocaleconv()` function returns a pointer to a `wcslconv` structure.

### Example that uses `wcslocaleconv()`

This example prints out the Unicode currency symbol for a French locale.

```
/******  
This example prints out the Unicode currency symbol for a French  
locale. You first must create a Unicode French locale. You can do  
this with this command:  
CRTLOCALE LOCALE('QSYS.LIB/MYLIB.LIB/LC_UNI_FR.LOCALE') +  
SRCFILE('QSYS.LIB/QSYSLOCALE.LIB/QLOCALESRC.FILE/ +  
FR_FR.MBR') CCSID(13488)  
  
Then you must compile your c program with LOCALETYPE(*LOCALEUCS2)  
*****/  
#include <stdio.h>  
#include <locale.h>  
int main(void) {  
    char * string;  
    struct wcslconv * mylocale;  
    if (NULL != (string = setlocale(LC_UNI_ALL,  
    "QSYS.LIB/MYLIB.LIB/LC_UNI_FR.LOCALE"))) {  
        mylocale = wcslocaleconv();  
        /* Display the Unicode currency symbol in a French locale */  
        printf("French Unicode currency symbol is a %ls\n", mylocale->currency_symbol);  
    } else {  
        printf("setlocale(LC_UNI_ALL, \"QSYS.LIB/MYLIB.LIB/LC_UNI_FR.LOCALE\") \  
        returned <NULL>\n");  
    }  
  
    return 0;  
}
```

### Related Information

- “`setlocale()` — Set Locale” on page 339
- “`<locale.h>`” on page 7
- “`localeconv()` — Retrieve Information from the Environment” on page 181

---

## **wcsncat()** — Concatenate Wide-Character Strings

### Format

```
| #include <wchar.h>  
| wchar_t *wcsncat(wchar_t *string1, const wchar_t *string2, size_t count);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `wcsncat()` function appends up to *count* wide characters from *string2* to the end of *string1*, and appends a `wchar_t` null character to the result.

The `wcsncat()` function operates on null-ending wide-character strings; string arguments to this function should contain a `wchar_t` null character marking the end of the string.

### Return Value

The `wcsncat()` function returns *string1*.

### Example that uses `wcsncat()`

This example demonstrates the difference between the `wcscat()` and `wcsncat()` functions. The `wcscat()` function appends the entire second string to the first; the `wcsncat()` function 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 = wcsncpy( buffer1, L"computer" );
|
|     /* Call wcsncat with buffer1 and " program" */
|     ptr = wcsncat( buffer1, L" program", 3 );
|     printf( "wcsncat: buffer1 = \"%ls\"\n", buffer1 );
| }
| /***** Output should be similar to: *****/
|
| wcscat : buffer1 = "computer program"
| wcsncat: buffer1 = "computer pr"
| */
```

### Related Information

- “`strcat()` — Concatenate Strings” on page 358
- “`strncat()` — Concatenate Strings” on page 377
- “`wcscat()` — Concatenate Wide-Character Strings” on page 451
- “`wcsncmp()` — Compare Wide-Character Strings”
- “`wcsncpy()` — Copy Wide-Character Strings” on page 466
- “`<wchar.h>`” on page 18

---

## `wcsncmp()` — Compare Wide-Character Strings

### Format

```
| #include <wchar.h>
| int wcsncmp(const wchar_t *string1, const wchar_t *string2, size_t count);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `wcsncmp()` function compares up to *count* wide characters in *string1* to *string2*.

The `wcsncmp()` function operates on null-ended wide-character strings; string arguments to this function should contain a `wchar_t` null character marking the end of the string.

### Return Value

The `wcsncmp()` function returns a value indicating the relationship between the two strings, as follows:

#### Value    Meaning

##### Less than 0

*string1* less than *string2*

0        *string1* identical to *string2*

##### Greater than 0

*string1* greater than *string2*.

### Example that uses `wcsncmp()`

This example demonstrates the difference between the `wscmp()` function, which compares the entire strings, and the `wcsncmp()` function, which compares only a specified number of wide characters in the strings.

```

| #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 should be similar to: *****/
|
| 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

- “strcmp() — Compare Strings” on page 360
- “strcoll() — Compare Strings” on page 363
- “strncmp() — Compare Strings” on page 379
- “wcscmp() — Compare Wide-Character Strings” on page 453
- “wcsncat() — Concatenate Wide-Character Strings” on page 463
- “wcsncpy() — Copy Wide-Character Strings”
- “<wchar.h>” on page 18

---

## wcsncpy() — Copy Wide-Character Strings

### Format

```

| #include <wchar.h>
| wchar_t *wcsncpy(wchar_t *string1, const wchar_t *string2, size_t count);

```

**Language Level:** XPG4

**Threadsafe:** Yes

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `wcsncpy()` function copies up to *count* wide characters from *string2* to *string1*. If *string2* is shorter than *count* characters, *string1* is padded out to *count* characters with `wchar_t` null characters.

The `wcsncpy()` function operates on null-ended wide-character strings; string arguments to this function should contain a `wchar_t` null character marking the end of the string. Only *string2* needs to contain a null character.

### Return Value

The `wcsncpy()` returns a pointer to *string1*.

### Related Information

- “`strcpy()` — Copy Strings” on page 364
  - “`strncpy()` — Copy Strings” on page 380
  - “`wscpy()` — Copy Wide-Character Strings” on page 456
  - “`wcsncat()` — Concatenate Wide-Character Strings” on page 463
  - “`wcsncmp()` — Compare Wide-Character Strings” on page 464
- | • “<wchar.h>” on page 18

---

## **\_\_wcsnicmp() — Compare Wide Character Strings without Case Sensitivity**

### Format

```
#include <wchar.h>;
int __wcsnicmp(const wchar_t *string1, const wchar_t *string2, size_t count);
```

**Language Level:** Extension

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the `LC_CTYPE` category of the current locale if `LOCALETYPE(*LOCALE)` is specified on the compilation command. The behavior of this function might also be affected by the `LC_UNI_CTYPE` category of the current locale if `LOCALETYPE(*LOCALEUCS2)` or `LOCALETYPE(*LOCALEUTF)` is specified on the compilation command. This function is not available when `LOCALETYPE(*CLD)` is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `__wcsnicmp()` function compares up to *count* characters of *string1* and *string2* without sensitivity to case. All alphabetic wide characters in *string1* and *string2* are converted to lowercase before comparison.

The `__wcsnicmp()` function operates on null terminated wide character strings. The string arguments to the function are expected to contain a `wchar_t` null character (`L'\0'`) marking the end of the string.

### Return Value

The `__wcsnicmp()` function returns a value indicating the relationship between the two strings, as follows:

Table 11. Return values of `__wcsicmp()`

Value	Meaning
Less than 0	<i>string1</i> less than <i>string2</i>
0	<i>string1</i> equivalent to <i>string2</i>
Greater than 0	<i>string1</i> greater than <i>string2</i>

### Example that uses `__wcsnicmp()`

This example uses `__wcsnicmp()` to compare two wide character strings.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *str1 = L"STRING ONE";
    wchar_t *str2 = L"string TWO";
    int result;

    result = __wcsnicmp(str1, str2, 6);

    if (result == 0)
        printf("Strings compared equal.\n");
    else if (result < 0)
        printf("\'%s\' is less than \''s\'.\n", str1, str2);
    else
        printf("\'%s\' is greater than \''s\'.\n", str1, str2);

    return 0;
}
/***** The output should be similar to: *****/

Strings compared equal.

*****/
```

### Related Information

- “`strcmp()` — Compare Strings” on page 360
- “`strncmp()` — Compare Strings” on page 379
- “`wcscat()` — Concatenate Wide-Character Strings” on page 451
- “`wcschr()` — Search for Wide Character” on page 452
- “`wcspsn()` — Find Offset of First Wide-Character Match” on page 457
- “`wcslen()` — Calculate Length of Wide-Character String” on page 461
- “`wcsncmp()` — Compare Wide-Character Strings” on page 464
- “`__wcsicmp()` — Compare Wide Character Strings without Case Sensitivity” on page 460
- “`<wchar.h>`” on page 18

---

## wcspbrk() — Locate Wide Characters in String

### Format

```
| #include <wchar.h>
| wchar_t *wcspbrk(const wchar_t *string1, const wchar_t *string2);
```

Language Level: XPG4

**Threadsafe:** Yes.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `wcsprk()` function locates the first occurrence in the string pointed to by *string1* of any wide character from the string pointed to by *string2*.

### Return Value

The `wcsprk()` function returns a pointer to the character. If *string1* and *string2* have no wide characters in common, the `wcsprk()` function returns `NULL`.

### Example that uses `wcsprk()`

This example uses `wcsprk()` to find the first occurrence of either “a” or “b” in the array *string*.

```
| #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 = wcsprk( string, chars);
|     printf("The first occurrence of any of the characters \"%s\" in "
|           "\"%s\" is \"%s\"\\n", chars, string, result);
| }
|
| /***** Output should be similar to: *****/
|
| The first occurrence of any of the characters "ab" in "The Blue Danube"
| is "anube"
| *****/
```

### Related Information

- “`strchr()` — Search for Character” on page 359
- “`strcspn()` — Find Offset of First Character Match” on page 365
- “`strpbrk()` — Find Characters in String” on page 384
- “`strspn()` — Find Offset of First Non-matching Character” on page 390
- “`wcschr()` — Search for Wide Character” on page 452
- “`wcscmp()` — Compare Wide-Character Strings” on page 453
- “`wcscspn()` — Find Offset of First Wide-Character Match” on page 457
- “`wcscncmp()` — Compare Wide-Character Strings” on page 464
- “`wcsrchr()` — Locate Last Occurrence of Wide Character in String” on page 471
- “`wcswcs()` — Locate Wide-Character Substring” on page 488
- | • “`<wchar.h>`” on page 18

---

## **wcsptime()**— Convert Wide Character String to Date/Time

### Format

```
#include <wchar.h>
wchar_t *wcsptime(const wchar_t *buf, const wchar_t *format, struct tm *tm);
```

**Language Level:** Extended.

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_UNI\_CTYPE, LC\_UNI\_TIME, and LC\_UNI\_TOD categories of the current locale. This function is only available when LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

## Description

The `wcsptime()` function converts the wide character string pointed to by *buf* to values that are stored in the *tm* structure pointed to by *tm*, using the format specified by *format*. This function is equivalent to `strptime()`, except that it uses wide characters.

See “`strptime()`— Convert String to Date/Time” on page 385 for a description of the format string.

## Return Value

On successful completion, the `wcsptime()` function returns a pointer to the character following the last wide character parsed. Otherwise, a null pointer is returned. The value of `errno` may be set to `ECONVERT` (conversion error).

## Example that uses `wcsptime()`

```
#include <stdio.h>
#include <time.h>
#include <wchar.h>

int main(void)
{
    wchar_t buf[100];
    time_t t;
    struct tm *timeptr,result;

    t = time(NULL);
    timeptr = localtime(&t);
    wcsftime(buf, 100, L"%a %m/%d/%Y %r", timeptr);

    if (wcsptime(buf, L"%a %m/%d/%Y %r", &result) == NULL)
        printf("\nwcsptime failed\n");
    else
    {
        printf("tm_hour:  %d\n",result.tm_hour);
        printf("tm_min:  %d\n",result.tm_min);
        printf("tm_sec:  %d\n",result.tm_sec);
        printf("tm_mon:  %d\n",result.tm_mon);
        printf("tm_mday: %d\n",result.tm_mday);
        printf("tm_year: %d\n",result.tm_year);
        printf("tm_yday: %d\n",result.tm_yday);
        printf("tm_wday: %d\n",result.tm_wday);
    }

    return 0;
}

/*****
```

The output should be similar to:

```
tm_hour: 14
tm_min: 25
tm_sec: 34
tm_mon: 7
tm_mday: 19
tm_year: 103
tm_yday: 230
tm_wday: 2
```

\*\*\*\*\*/

### Related Information

- “asctime() — Convert Time to Character String” on page 39
- “asctime\_r() — Convert Time to Character String (Restartable)” on page 41
- “ctime() — Convert Time to Character String” on page 71
- | • “ctime64() — Convert Time to Character String” on page 73
- | • “ctime64\_r() — Convert Time to Character String (Restartable)” on page 76
- “ctime\_r() — Convert Time to Character String (Restartable)” on page 74
- “gmtime() — Convert Time” on page 161
- “gmtime64() — Convert Time” on page 163
- “gmtime64\_r() — Convert Time (Restartable)” on page 167
- “gmtime\_r() — Convert Time (Restartable)” on page 165
- “localtime() — Convert Time” on page 185
- “localtime64() — Convert Time” on page 187
- “localtime64\_r() — Convert Time (Restartable)” on page 189
- “localtime\_r() — Convert Time (Restartable)” on page 188
- “setlocale() — Set Locale” on page 339
- “strftime() — Convert Date/Time to String” on page 370
- “strptime() — Convert String to Date/Time” on page 385
- “time() — Determine Current Time” on page 411
- “time64() — Determine Current Time” on page 412
- “<time.h>” on page 18

---

## wcsrchr() — Locate Last Occurrence of Wide Character in String

### Format

```
| #include <wchar.h>
| wchar_t *wcsrchr(const wchar_t *string, wchar_t character);
```

**Language Level:** ANSI

**Threadsafe:** Yes

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `wcsrchr()` function locates the last occurrence of *character* in the string pointed to by *string*. The ending `wchar_t` null character is considered to be part of the string.

### Return Value

The `wcsrchr()` function returns a pointer to the character, or a NULL pointer if *character* does not occur in the string.

### Example that uses `wcsrchr()`

This example compares the use of `wcschr()` and `wcsrchr()`. It searches the string for the first and last occurrence of `p` in the wide character string.

```
| #include <stdio.h>
| #include <wchar.h>
|
| #define SIZE 40
|
| int main(void)
| {
|     wchar_t buf[SIZE] = L"computer program";
|     wchar_t * ptr;
|     int     ch = 'p';
|
|     /* This illustrates wcschr */
|     ptr = wcschr( buf, ch );
|     printf( "The first occurrence of %c in '%ls' is '%ls'\n", ch, buf, ptr );
|
|     /* This illustrates wcsrchr */
|     ptr = wcsrchr( buf, ch );
|     printf( "The last occurrence of %c in '%ls' is '%ls'\n",  ch, buf, ptr );
| }
|
| /***** Output should be similar to: *****/
|
| The first occurrence of p in 'computer program' is 'puter program'
| The last occurrence of p in 'computer program' is 'program'
| */
```

### Related Information

- “`strchr()` — Search for Character” on page 359
- “`strrchr()` — Locate Last Occurrence of Character in String” on page 389
- “`strcspn()` — Find Offset of First Character Match” on page 365
- “`strspn()` — Find Offset of First Non-matching Character” on page 390
- “`wcschr()` — Search for Wide Character” on page 452
- “`wcscmp()` — Compare Wide-Character Strings” on page 453
- “`wcscspn()` — Find Offset of First Wide-Character Match” on page 457
- “`wcscmp()` — Compare Wide-Character Strings” on page 464
- “`wcswcs()` — Locate Wide-Character Substring” on page 488
- “`wcspbrk()` — Locate Wide Characters in String” on page 468
- “`<wchar.h>`” on page 18

---

## wcsrtombs() — Convert Wide Character String to Multibyte String (Restartable)

### Format

```
#include <wchar.h>
size_t wcsrtombs (char *dst, const wchar_t **src, size_t len,
                 mbstate_t *ps);
```

**Language Level:** ANSI

**Threadsafe:** Yes, if the fourth parameter, *ps*, is not NULL.

- | **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. The behavior might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.
- | **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

This function is the restartable version of `wcstombs()`.

The `wcsrtombs()` function converts a sequence of wide characters from the array indirectly pointed to by *src* into a sequence of corresponding multibyte characters that begins 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 ending null wide character, which is also stored. Conversion will stop earlier in two cases: when a code is reached that does not correspond to a valid multibyte character, or (if *dst* is not a null pointer) 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 place as if by a call to `wcrtomb()`.

If *dst* is not a null pointer, the object pointed to by *src* will be assigned either a null pointer (if conversion stopped due to reaching a ending null character) or the address of the code just past the last wide character converted. If conversion stopped due to reaching a ending null wide character, the resulting state described will be the initial conversion state.

### Return Value

If the first code is not a valid wide character, an encoding error will occur. `wcsrtombs()` stores the value of the macro `EILSEQ` in `errno` and returns `(size_t) -1`, but the conversion state will be unchanged. Otherwise it returns the number of bytes in the resulting multibyte character sequence, which is the same as the number of array elements changed when *dst* is not a null pointer.

If a conversion error occurs, `errno` may be set to `ECONVERT`.

**Example that uses `wcsrtombs()`**

```

#include <stdio.h>
#include <wchar.h>
#include <string.h>

#define SIZE 20

int main(void)
{
    char    dest[SIZE];
    wchar_t *wcs = L"string";
    wchar_t *ptr;
    size_t  count = SIZE;
    size_t  length;
    mbstate_t ps = 0;

    ptr = (wchar_t *) wcs;
    length = wcsrtombs(dest, ptr, count, &ps);
    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, &ps);
    printf("%d characters were converted.\n", length);
    printf("The converted string is \"%s\"\n\n", dest);
}

/***** Output should be similar to: *****/
6 characters were converted.
The converted string is "string"

3 characters were converted.
The converted string is "str"
*/

```

### Related Information

- “mblen() — Determine Length of a Multibyte Character” on page 197
- “mbrlen() — Determine Length of a Multibyte Character (Restartable)” on page 199
- “mbrtowc() — Convert a Multibyte Character to a Wide Character (Restartable)” on page 201
- “mbsrtowcs() — Convert a Multibyte String to a Wide Character String (Restartable)” on page 206
- “wctomb() — Convert a Wide Character to a Multibyte Character (Restartable)” on page 446
- “wcstombs() — Convert Wide-Character String to Multibyte String” on page 483
- “<wchar.h>” on page 18

---

## wcsspn() — Find Offset of First Non-matching Wide Character

### Format

```

| #include <wchar.h>
| size_t wcsspn(const wchar_t *string1, const wchar_t *string2);

```

**Language Level:** ANSI

**Threadsafe:** Yes

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `wcsspn()` function computes the number of wide characters in the initial segment of the string pointed to by `string1`, which consists entirely of wide characters from the string pointed to by `string2`.

### Return Value

The `wcsspn()` function returns the number of wide characters in the segment.

### Example that uses `wcsspn()`

This example finds the first occurrence in the array `string` of a wide character that is not an a, b, or c. Because the string in this example is `cabbage`, the `wcsspn()` function 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 should be similar to: *****/
| The first 5 characters of "cabbage" are found in "abc"
| */
```

### Related Information

- “`strchr()` — Search for Character” on page 359
- “`strcspn()` — Find Offset of First Character Match” on page 365
- “`strpbrk()` — Find Characters in String” on page 384
- “`strrchr()` — Locate Last Occurrence of Character in String” on page 389
- “`strspn()` — Find Offset of First Non-matching Character” on page 390
- “`wscat()` — Concatenate Wide-Character Strings” on page 451
- “`wcschr()` — Search for Wide Character” on page 452
- “`wscmp()` — Compare Wide-Character Strings” on page 453
- “`wcscspn()` — Find Offset of First Wide-Character Match” on page 457
- “`wcscmp()` — Compare Wide-Character Strings” on page 464
- “`wcspbrk()` — Locate Wide Characters in String” on page 468
- “`wcsrchr()` — Locate Last Occurrence of Wide Character in String” on page 471
- “`wcsspn()` — Find Offset of First Non-matching Wide Character” on page 474
- “`wcswcs()` — Locate Wide-Character Substring” on page 488
- “`<wchar.h>`” on page 18

---

## **wcsstr() — Locate Wide-Character Substring**

### Format

```
#include <wchar.h>
wchar_t *wcsstr(const wchar_t *wcs1, const wchar_t *wcs2);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `wcsstr()` function locates the first occurrence of `wcs2` in `wcs1`.

### Return Value

The `wcsstr()` function returns a pointer to the beginning of the first occurrence of `wcs2` in `wcs1`. If `wcs2` does not appear in `wcs1`, the `wcsstr()` function returns `NULL`. If `wcs2` points to a wide-character string with zero length, it returns `wcs1`.

### Example that uses `wcsstr()`

This example uses the `wcsstr()` function to find the first occurrence of “hay” in the wide-character string “needle in a haystack”.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *wcs1 = L"needle in a haystack";
    wchar_t *wcs2 = L"hay";

    printf("result: \"%ls\"\n", wcsstr(wcs1, wcs2));
    return 0;

    /*****
    The output should be similar to:

    result: "haystack"
    *****/
}
```

### Related Information

- “`strstr()` — Locate Substring” on page 391
- “`wcschr()` — Search for Wide Character” on page 452
- “`wcsrchr()` — Locate Last Occurrence of Wide Character in String” on page 471
- “`wcswcs()` — Locate Wide-Character Substring” on page 488
- “`<wchar.h>`” on page 18

---

## **wcstod() — Convert Wide-Character String to Double**

### Format

```
#include <wchar.h>
double wcstod(const wchar_t *nptr, wchar_t **endptr);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

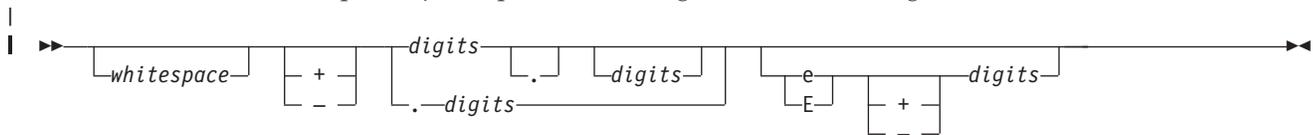
| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale if LOCALETYPE(\*LOCALE) is specified on the compilation command. The behavior of this function might also be affected by the LC\_UNI\_CTYPE and LC\_UNI\_NUMERIC categories of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

| The `wcstod()` function converts the initial portion of the wide-character string pointed to by `nptr` to a double value. The `nptr` parameter points to a sequence of characters that can be interpreted as a numeric binary floating-point value. The `wcstod()` function stops reading the string at the first character that it cannot recognize as part of a number. This character can be the `wchar_t` null character at the end of the string.

| The `wcstod()` function expects `nptr` to point to a string with the following form:



| The first character that does not fit this form stops the scan. In addition, a sequence of INFINITY or NAN (ignoring case) is allowed.

### Return Value

The `wcstod()` function returns the converted double value. If no conversion could be performed, the `wcstod()` function returns 0. If the correct value is outside the range of representable values, the `wcstod()` function returns `+HUGE_VAL` or `-HUGE_VAL` (according to the sign of the value), and sets `errno` to `ERANGE`. If the correct value would cause underflow, the `wcstod()` function returns 0 and sets `errno` to `ERANGE`. If the string `nptr` points to is empty or does not have the expected form, no conversion is performed, and the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

| The `wcstod()` function does not fail if a character other than a digit follows an E or e that is read as an exponent. For example, `100elf` is converted to the floating-point value 100.0.

The value of `errno` may be set to **ERANGE**, range error.

| A character sequence of INFINITY (ignoring case) yields a value of INFINITY. A character value of NAN yields a Quiet Not-A-Number (NAN) value.

### Example that uses wcstod()

| This example uses the `wcstod()` function to convert the string `wcs` to a binary floating-point value.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *wcs = L"3.1415926This stopped it";
    wchar_t *stopwcs;
```

```

printf("wcs = \"%ls\\n\"", wcs);
printf("   wctod = %f\\n", wctod(wcs, &stopwcs));
printf("   Stop scanning at \"%ls\\n\"", stopwcs);
return 0;

/*****
   The output should be similar to:

   wcs = "3.1415926This stopped it"
   wctod = 3.141593
   Stop scanning at "This stopped it"
*****/
}

```

## Related Information

- “strtod() — strtodf() — strtold — Convert Character String to Double, Float, and Long Double” on page 392
- | • “strtod32() — strtod64() — strtod128() — Convert Character String to Decimal Floating-Point” on page 395
- “strtol() — strtoll() — Convert Character String to Long and Long Long Integer” on page 400
- | • “wctod32() — wctod64() — wctod128()— Convert Wide-Character String to Decimal Floating-Point”
- “wcstol() — wcstoll() — Convert Wide Character String to Long and Long Long Integer” on page 481
- “wcstoul() — wcstoull() — Convert Wide Character String to Unsigned Long and Unsigned Long Long Integer” on page 486
- “<wchar.h>” on page 18

---

## | **wctod32() — wctod64() — wctod128()— Convert Wide-Character String to Decimal Floating-Point**

### | **Format**

```

#include <wchar.h>
| _Decimal32 wctod32(const wchar_t *nptr, wchar_t **endptr);
| _Decimal64 wctod64(const wchar_t *nptr, wchar_t **endptr);
| _Decimal128 wctod128(const wchar_t *nptr, wchar_t **endptr);

```

| **Language Level:** XPG4

| **Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of these functions might be affected by the LC\_CTYPE and LC\_NUMERIC categories of the current locale if LOCALETYPE(\*LOCALE) is specified on the compilation command. The behavior of these functions might also be affected by the LC\_UNI\_CTYPE and LC\_UNI\_NUMERIC categories of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. These functions are not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

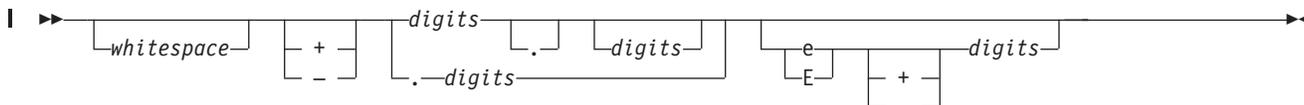
| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### | **Description**

| The wctod32(), wctod64(), and wctod128() functions convert the initial portion of the wide-character string pointed to by *nptr* to a single-precision, double-precision, or quad-precision decimal floating-point value. The parameter *nptr* points to a sequence of characters that can be interpreted as a numeric decimal floating-point value. The wctod32(), wctod64(), and wctod128() functions stop reading the string at

| the first character that is not recognized as part of a number. This character can be the `wchar_t` null character at the end of the string. The `endptr` parameter is updated to point to this character, provided that `endptr` is not a NULL pointer.

| The `wcstod32()`, `wcstod64()`, and `wcstod128()` functions expect `nptr` to point to a string with the following form:



| The first character that does not fit this form stops the scan. In addition, a sequence of INFINITY or NAN (ignoring case) is allowed.

### | Return Value

| The `wcstod32()`, `wcstod64()`, and `wcstod128()` functions return the value of the floating-point number, except when the representation causes an underflow or overflow. For an overflow, `wcstod32()` returns `HUGE_VAL_D32` or `-HUGE_VAL_D32`; `wcstod64()` returns `HUGE_VAL_D64` or `-HUGE_VAL_D64`; `wcstod128()` returns `HUGE_VAL_D128` or `-HUGE_VAL_D128`. For an underflow, all functions return `+0.E0`.

| In both the overflow and underflow cases, `errno` is set to `ERANGE`. If the string pointed to by `nptr` does not have the expected form, a value of `+0.E0` is returned and the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a NULL pointer.

| The `wcstod32()`, `wcstod64()`, and `wcstod128()` functions do not fail if a character other than a digit follows an E or e that is read as an exponent. For example, `100elf` is converted to the floating-point value `100.0`.

| A character sequence of INFINITY (ignoring case) yields a value of INFINITY. A character value of NAN (ignoring case) yields a Quiet Not-A-Number (NaN) value.

| If necessary, the return value is rounded using the rounding mode Round to Nearest, Ties to Even.

### | Example that uses `wcstod32()`, `wcstod64()`, and `wcstod128()`

| This example converts the string `wcs` to single-precision, double-precision, and quad-precision decimal floating-point values.

```
| #include <stdio.h>
| #include <wchar.h>
|
| int main(void)
| {
|     wchar_t *wcs = L"3.1415926This stopped it";
|     wchar_t *stopwcs;
|
|     printf("wcs = \"%ls\"\n", wcs);
|     printf("wcstod32 = %Hf\n", wcstod32(wcs, &stopwcs));
|     printf(" Stopped scan at \"%ls\"\n", stopwcs);
|     printf("wcs = \"%ls\"\n", wcs);
|     printf("wcstod64 = %Df\n", wcstod64(wcs, &stopwcs));
|     printf(" Stopped scan at \"%ls\"\n", stopwcs);
|     printf("wcs = \"%ls\"\n", wcs);
|     printf("wcstod128 = %DDf\n", wcstod128(wcs, &stopwcs));
|     printf(" Stopped scan at \"%ls\"\n", stopwcs);
| }
```

```

|
| /***** Output should be similar to: *****/
|
| wcs = "3.1415926This stopped it"
|   wctod = 3.141593
|   Stopped scan at "This stopped it"
| wcs = "3.1415926This stopped it"
|   wctod = 3.141593
|   Stopped scan at "This stopped it"
| wcs = "3.1415926This stopped it"
|   wctod = 3.141593
|   Stopped scan at "This stopped it"
|
| */
|

```

### Related Information

- | • “strtod() — strtodf() — strtold — Convert Character String to Double, Float, and Long Double” on page 392
- | • “strtod32() — strtod64() — strtod128() — Convert Character String to Decimal Floating-Point” on page 395
- | • “strtol() — strtoll() — Convert Character String to Long and Long Long Integer” on page 400
- | • “wctod() — Convert Wide-Character String to Double” on page 476
- | • “wcstol() — wcstoll() — Convert Wide Character String to Long and Long Long Integer” on page 481
- | • “wcstoul() — wcstoull() — Convert Wide Character String to Unsigned Long and Unsigned Long Long Integer” on page 486
- | • “<wchar.h>” on page 18

---

## wcstok() — Tokenize Wide-Character String

### Format

```

#include <wchar.h>
wchar_t *wcstok(wchar_t *wcs1, const wchar_t *wcs2, wchar_t **ptr);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `wcstok()` function reads `wcs1` as a series of zero or more tokens and `wcs2` as the set of wide characters serving as delimiters for the tokens in `wcs1`. A sequence of calls to the `wcstok()` function locates the tokens inside `wcs1`. The tokens can be separated by one or more of the delimiters from `wcs2`. The third argument points to a wide-character pointer that you provide where the `wcstok()` function stores information necessary for it to continue scanning the same string.

When the `wcstok()` function is first called for the wide-character string `wcs1`, it searches for the first token in `wcs1`, skipping over leading delimiters. The `wcstok()` function returns a pointer to the first token. To read the next token from `wcs1`, call the `wcstok()` function with NULL as the first parameter (`wcs1`). This NULL parameter causes the `wcstok()` function to search for the next token in the previous token string. Each delimiter is replaced by a null character to end the token.

The `wcstok()` function always stores enough information in the pointer `ptr` so that subsequent calls, with NULL as the first parameter and the unmodified pointer value as the third, will start searching right after the previously returned token. You can change the set of delimiters (`wcs2`) from call to call.

## Return Value

The `wcstok()` function returns a pointer to the first wide character of the token, or a null pointer if there is no token. In later calls with the same token string, the `wcstok()` function returns a pointer to the next token in the string. When there are no more tokens, the `wcstok()` function returns `NULL`.

## Example that uses `wcstok()`

This example uses the `wcstok()` function to locate the tokens in the wide-character string `str1`.

```
#include <stdio.h>
#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" */
    printf("t = '%ls'\n", t);
    t = wcstok(NULL, L",", &ptr1); /* t points to the token L"?b" */
    printf("t = '%ls'\n", t);
    t = wcstok(str2, L" \t", &ptr2); /* t is a null pointer */
    printf("t = '%ls'\n", t);
    t = wcstok(NULL, L"#", &ptr1); /* t points to the token L"c" */
    printf("t = '%ls'\n", t);
    t = wcstok(NULL, L"?", &ptr1); /* t is a null pointer */
    printf("t = '%ls'\n", t);
    return 0;

    /*****
    The output should be similar to:

        t = 'a'
        t = '?b'
        t = ''
        t = 'c'
        t = ''
    *****/
}
```

## Related Information

- “`strtok()` — Tokenize String” on page 398
- “`<wchar.h>`” on page 18

---

## `wcstol()` — `wcstoll()` — Convert Wide Character String to Long and Long Long Integer

### Format (`wcstol()`)

```
#include <wchar.h>
long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);
```

### Format (`wcstoll()`)

```
#include <wchar.h>
long long int wcstoll(const wchar_t *nptr, wchar_t **endptr, int base);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

- | **Locale Sensitive:** The behavior of these functions might be affected by the LC\_CTYPE category of the current locale if LOCALETYPE(\*LOCALE) is specified on the compilation command. The behavior of these functions might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. These functions are not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.
- | **Wide Character Function:** See “Wide Characters” on page 527 for more information.

## Description

The `wcstol()` function converts the initial portion of the wide-character string pointed to by `nptr` to a long integer value. The `nptr` parameter points to a sequence of wide characters that can be interpreted as a numerical value of type long int. The `wcstol()` function stops reading the string at the first wide character that it cannot recognize as part of a number. This character can be the `wchar_t` null character at the end of the string. The ending character can also be the first numeric character greater than or equal to the base.

The `wcstoll()` subroutine converts a wide-character string to a long long integer. The wide-character string is parsed to skip the initial space characters (as determined by the `iswspace` subroutine). Any non-space character signifies the start of a subject string that may form a long long int in the radix specified by the `base` parameter. The subject sequence is defined to be the longest initial substring that is a long long int of the expected form.

If the value of the `endptr` parameter is not null, then a pointer to the character that ended the scan is stored in `endptr`. If a long long integer cannot be formed, the value of the `endptr` parameter is set to that of the `nptr` parameter.

If the `base` parameter is a value between 2 and 36, the subject sequence’s expected form is a sequence of letters and digits representing a long long integer whose radix is specified by the `base` parameter. This sequence optionally is preceded by a positive (+) or negative (-) sign. Letters from a (or A) to z (or Z) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of the `base` parameter are permitted. If the `base` parameter has a value of 16, the characters 0x or 0X optionally precede the sequence of letters and digits, following the positive (+) or negative (-) sign, if present.

If the value of the `base` parameter is 0, the string determines the base. Therefore, after an optional leading sign, a leading 0 indicates octal conversion, and a leading 0x or 0X indicates hexadecimal conversion.

## Return Value

The `wcstol()` function returns the converted long integer value. If no conversion could be performed, the `wcstol()` function returns 0. If the correct value is outside the range of representable values, the `wcstol()` function returns `LONG_MAX` or `LONG_MIN` (according to the sign of the value), and sets `errno` to `ERANGE`. If the string `nptr` points to is empty or does not have the expected form, no conversion is performed, and the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Upon successful completion, the `wcstoll()` subroutine returns the converted value. If no conversion could be performed, 0 is returned, and the `errno` global variable is set to indicate the error. If the correct value is outside the range of representable values, the `wcstoll()` subroutine returns a value of `LONG_LONG_MAX` or `LONG_LONG_MIN`.

The value of `errno` may be set to `ERANGE` (range error), or `EINVAL` (invalid argument).

## Example that uses `wcstol()`

This example uses the `wcstol()` function to convert the wide-character string `wcs` to a long integer value.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wchar_t *wcs = L"10110134932";
    wchar_t *stopwcs;
    long    l;
    int     base;

    printf("wcs = \"%ls\\n\"", wcs);
    for (base=2; base<=8; base*=2) {
        l = wcstol(wcs, &stopwcs, base);
        printf("    wcstol = %ld\\n"
              "    Stopped scan at \"%ls\\n\\n\"", l, stopwcs);
    }
    return 0;

    /*****
    The output should be similar to:

    wcs = "10110134932"
    wcstol = 45
    Stopped scan at "34932"

    wcstol = 4423
    Stopped scan at "4932"

    wcstol = 2134108
    Stopped scan at "932"
    *****/
}
```

### Related Information

- “`strtod()` — `strtof()` — `strtold()` — Convert Character String to Double, Float, and Long Double” on page 392
- “`strtod32()` — `strtod64()` — `strtod128()` — Convert Character String to Decimal Floating-Point” on page 395
- “`strtol()` — `strtoll()` — Convert Character String to Long and Long Long Integer” on page 400
- “`strtoul()` — `strtoull()` — Convert Character String to Unsigned Long and Unsigned Long Long Integer” on page 403
- “`wcstod()` — Convert Wide-Character String to Double” on page 476
- “`wcstod32()` — `wcstod64()` — `wcstod128()` — Convert Wide-Character String to Decimal Floating-Point” on page 478
- “`wcstoul()` — `wcstoull()` — Convert Wide Character String to Unsigned Long and Unsigned Long Long Integer” on page 486
- “`<wchar.h>`” on page 18

---

## **wcstombs()** — Convert Wide-Character String to Multibyte String

### Format

```
#include <stdlib.h>
size_t wcstombs(char *dest, const wchar_t *string, size_t count);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the current locale. The behavior might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

## Description

The `wcstombs()` function 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 `wchar_t` null character is encountered.

Only complete multibyte characters are stored in *dest*. If the lack of space in *dest* would cause a partial multibyte character to be stored, `wcstombs()` stores fewer than *n* bytes and discards the invalid character.

## Return Value

The `wcstombs()` function returns the length in bytes of the multibyte character string, not including a ending null character. The value `(size_t)-1` is returned if an invalid multibyte character is encountered.

The value of `errno` may be set to `EILSEQ` (conversion stopped due to input character), or `ECONVERT` (conversion error).

## Examples that use `wcstombs()`

This program is compiled with `LOCALETYPE(*LOCALE)` and `SYSIFCOPT(*IFSIO)`:

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>

#define STRLENGTH 10
#define LOCNAME "qsys.lib/JA_JP.locale"
#define LOCNAME_EN "qsys.lib/EN_US.locale"

int main(void)
{
    char    string[STRLENGTH];
    int length, sl = 0;
    wchar_t wc2[] = L"ABC";
    wchar_t wc_string[10];
    mbstate_t ps = 0;
    memset(string, '\0', STRLENGTH);
    wc_string[0] = 0x00C1;
    wc_string[1] = 0x4171;
    wc_string[2] = 0x4172;
    wc_string[3] = 0x00C2;
    wc_string[4] = 0x0000;

    /* In this first example we will convert a wide character string */
    /* to a single byte character string. We first set the locale */
    /* to a single byte locale. We choose a locale with */
    /* CCSID 37. */

    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");

    length = wcstombs(string, wc2, 10);

    /* In this case wide characters ABC are converted to */
```

```

/* single byte characters ABC, length is 3. */
printf("string = %s, length = %d\n\n", string, length);

/* Now lets try a multibyte example. We first must set the */
/* locale to a multibyte locale. We choose a locale with */
/* CCSID 5026 */

if (setlocale(LC_ALL, LOCNAME) == NULL)
    printf("setlocale failed.\n");

length = wcstombs(string, wc_string, 10);

/* The hex look at string would now be: */
/* C10E417141720FC2 length will be 8 */
/* You would need a device capable of displaying multibyte */
/* characters to see this string. */

printf("length = %d\n\n", length);
}
/* The output should look like this:

string = ABC, length = 3

length = 8
*/

```

This program is compiled with LOCALETYPE(\*LOCALEUCS2) and SYSIFCOPT(\*IFSIO):

```

#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <wchar.h>

#define STRLENGTH 10
#define LOCNAME "qsys.lib/JA_JP.locale"
#define LOCNAME_EN "qsys.lib/EN_US.locale"

int main(void)
{
    char string[STRLENGTH];
    int length, sl = 0;
    wchar_t wc2[] = L"ABC";
    wchar_t wc_string[10];
    mbstate_t ps = 0;
    memset(string, '\0', STRLENGTH);
    wc_string[0] = 0x0041; /* UNICODE A */
    wc_string[1] = 0xFF41;
    wc_string[2] = 0xFF42;
    wc_string[3] = 0x0042; /* UNICODE B */
    wc_string[4] = 0x0000;
    /* In this first example we will convert a wide character string */
    /* to a single byte character string. We first set the locale */
    /* to a single byte locale. We choose a locale with */
    /* CCSID 37. */

    if (setlocale(LC_ALL, LOCNAME_EN) == NULL)
        printf("setlocale failed.\n");

    length = wcstombs(string, wc2, 10);

    /* In this case wide characters ABC are converted to */
    /* single byte characters ABC, length is 3. */

    printf("string = %s, length = %d\n\n", string, length);
}

```

```

/* Now lets try a multibyte example. We first must set the */
/* locale to a multibyte locale. We choose a locale with */
/* CCSID 5026 */

if (setlocale(LC_ALL, LOCNAME) == NULL)
    printf("setlocale failed.\n");

length = wcstombs(string, wc_string, 10);

/* The hex look at string would now be: */
/* C10E428142820FC2 length will be 8 */
/* You would need a device capable of displaying multibyte */
/* characters to see this string. */

printf("length = %d\n\n", length);
}
/* The output should look like this:
string = ABC, length = 3
length = 8
*/

```

### Related Information

- “mbstowcs() — Convert a Multibyte String to a Wide Character String” on page 207
- “wcslen() — Calculate Length of Wide-Character String” on page 461
- “wcsrtoombs() — Convert Wide Character String to Multibyte String (Restartable)” on page 473
- “wctomb() — Convert Wide Character to Multibyte Character” on page 492
- “<stdlib.h>” on page 17

---

## wcstoul() — wcstoull() — Convert Wide Character String to Unsigned Long and Unsigned Long Long Integer

### Format (wcstoul())

```

#include <wchar.h>
unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr, int base);

```

### Format (wcstoull())

```

#include <wchar.h>
unsigned long long int wcstoull(const wchar_t *nptr, wchar_t **endptr, int base);

```

**Language Level:** ANSI

**Threadsafe:** Yes.

**Locale Sensitive:** The behavior of these functions might be affected by the LC\_CTYPE category of the current locale if LOCALETYPE(\*LOCALE) is specified on the compilation command. The behavior of these functions might also be affected by the LC\_UNI\_CTYPE category of the current locale if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. These functions are not available when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

**Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `wcstoul()` function converts the initial portion of the wide-character string pointed to by `nptr` to an unsigned long integer value. The `nptr` parameter points to a sequence of wide characters that can be interpreted as a numerical value of type unsigned long int. The `wcstoul()` function stops reading the string at the first wide character that it cannot recognize as part of a number. This character can be the `wchar_t` null character at the end of the string. The ending character can also be the first numeric character greater than or equal to the base.

The `wcstoull()` subroutine converts a wide-character string to an unsigned long long integer. The wide-character string is parsed to skip the initial space characters (as determined by the `iswspace` subroutine). Any non-space character signifies the start of a subject string that may form an unsigned long long int in the radix specified by the `base` parameter. The subject sequence is defined to be the longest initial substring that is an unsigned long long int of the expected form.

If the value of the `endptr` parameter is not null, then a pointer to the character that ended the scan is stored in `endptr`. If an unsigned long long integer cannot be formed, the value of the `endptr` parameter is set to that of the `nptr` parameter.

If the `base` parameter is a value between 2 and 36, the subject sequence's expected form is a sequence of letters and digits representing an unsigned long long integer whose radix is specified by the `base` parameter. This sequence optionally is preceded by a positive (+) or negative (-) sign. Letters from a (or A) to z (or Z) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of the `base` parameter are permitted. If the `base` parameter has a value of 16, the characters 0x or 0X optionally precede the sequence of letters and digits, following the positive (+) or negative (-) sign, if present.

If the value of the `base` parameter is 0, the string determines the base. Therefore, after an optional leading sign, a leading 0 indicates octal conversion, and a leading 0x or 0X indicates hexadecimal conversion.

The value of `errno` may be set to `EINVAL` (`endptr` is null, no numbers are found, or base is invalid), or `ERANGE` (converted value is outside the range).

### Return Value

The `wcstoul()` function returns the converted unsigned long integer value. If no conversion could be performed, the `wcstoul()` function returns 0. If the correct value is outside the range of representable values, The `wcstoul()` function returns `ULONG_MAX` and sets `errno` to `ERANGE`. If the string `nptr` points to is empty or does not have the expected form, no conversion is performed, and the value of `nptr` is stored in the object pointed to by `endptr`, provided that `endptr` is not a null pointer.

Upon successful completion, the `wcstoull()` subroutine returns the converted value. If no conversion could be performed, 0 is returned, and the `errno` global variable is set to indicate the error. If the correct value is outside the range of representable values, `wcstoull()` subroutine returns a value of `ULONG_LONG_MAX`.

### Example that uses `wcstoul()`

This example uses the `wcstoul()` function to convert the string `wcs` to an unsigned long integer value.

```
#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"
       "The unsigned long int produced: %lu\n"
       "The substring of the input wide string that was not"
       " converted to unsigned long: `%ls`\n", wcs, answer, endptr);
return 0;

/*****
The output should be similar to:

The input wide string used: 1000e13 camels
The unsigned long int produced: 8
The substring of the input wide string that was not converted to
unsigned long: e13 camels
*****/
}

```

### Related Information

- “`strtod()` — `strtof()` — `strtold` — Convert Character String to Double, Float, and Long Double” on page 392
- “`strtod32()` — `strtod64()` — `strtod128()` — Convert Character String to Decimal Floating-Point” on page 395
- “`strtol()` — `strtoll()` — Convert Character String to Long and Long Long Integer” on page 400
- “`wcstod()` — Convert Wide-Character String to Double” on page 476
- “`wcstod32()` — `wcstod64()` — `wcstod128()`— Convert Wide-Character String to Decimal Floating-Point” on page 478
- “`wcstol()` — `wcstoll()` — Convert Wide Character String to Long and Long Long Integer” on page 481
- “<wchar.h>” on page 18

---

## wcswcs() — Locate Wide-Character Substring

### Format

```

#include <wchar.h>
wchar_t *wcswcs(const wchar_t *string1, const wchar_t *string2);

```

**Language Level:** XPG4

**Threadsafe:** Yes.

**Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `wcswcs()` function locates the first occurrence of *string2* in the wide-character string pointed to by *string1*. In the matching process, the `wcswcs()` function ignores the `wchar_t` null character that ends *string2*.

### Return Value

The `wcswcs()` function returns a pointer to the located string or `NULL` if the string is not found. If *string2* points to a string with zero length, `wcswcs()` returns *string1*.

### Example that uses `wcswcs()`

This example finds the first occurrence of the wide character string `pr` in `buffer1`.

```
| #include <stdio.h>
| #include <wchar.h>
|
| #define SIZE 40
|
| int main(void)
| {
|     wchar_t buffer1[SIZE] = L"computer program";
|     wchar_t * ptr;
|     wchar_t * wch = L"pr";
|
|     ptr = wcs wcs( buffer1, wch );
|     printf( "The first occurrence of %ls in '%ls' is '%ls'\n",
|            wch, buffer1, ptr );
| }
|
| /***** Output should be similar to: *****/
|
| The first occurrence of pr in 'computer program' is 'program'
| */
```

### Related Information

- “`strchr()` — Search for Character” on page 359
- “`strcspn()` — Find Offset of First Character Match” on page 365
- “`strpbrk()` — Find Characters in String” on page 384
- “`strrchr()` — Locate Last Occurrence of Character in String” on page 389
- “`strspn()` — Find Offset of First Non-matching Character” on page 390
- “`strstr()` — Locate Substring” on page 391
- “`wcschr()` — Search for Wide Character” on page 452
- “`wscmp()` — Compare Wide-Character Strings” on page 453
- “`wscspn()` — Find Offset of First Wide-Character Match” on page 457
- “`wcspbrk()` — Locate Wide Characters in String” on page 468
- “`wcsrchr()` — Locate Last Occurrence of Wide Character in String” on page 471
- “`wcsspn()` — Find Offset of First Non-matching Wide Character” on page 474
- “`<wchar.h>`” on page 18

---

## wcswidth() — Determine the Display Width of a Wide Character String

### Format

```
#include <wchar.h>
int wcswidth (const wchar_t *wcs, size_t n);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

**Locale Sensitive:** The behavior of this function might be affected by the `LC_CTYPE` category of the current locale if `LOCALETYPE(*LOCALE)` is specified on the compilation command. The behavior of this function might also be affected by the `LC_UNI_CTYPE` category of the current locale if `LOCALETYPE(*LOCALEUCS2)` or `LOCALETYPE(*LOCALEUTF)` is specified on the compilation command. This function is not available when `LOCALETYPE(*CLD)` is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

**Wide Character Function:** See “Wide Characters” on page 527 for more information.

## Description

The `wcswidth()` function 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 string pointed to by `wcs` occupies on a display device. The number is independent of its location on the device.

The value of `errno` may be set to `EINVAL` (non-printing wide character).

## Return Value

The `wcswidth()` function either returns:

- 0, if `wcs` points to a null wide character; or
- the number of printing positions occupied by the wide string pointed to by `wcs`; or
- -1, if any wide character in the wide string pointed to by `wcs` is not a printing wide character.

## Example that uses `wcswidth()`

```
#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));
}

/*****The output is as follows*****/
/*
*/
/*          wcs has a width of: 3          */
/*
*/
/*****/
```

## Related Information

- “`wcswidth()` — Determine the Display Width of a Wide Character String” on page 489
- “`<wchar.h>`” on page 18

---

## `wcsxfrm()` — Transform a Wide-Character String

### Format

```
#include <wchar.h>
size_t wcsxfrm (wchar_t *wcs1, const wchar_t *wcs2, size_t n);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the `LC_COLLATE` category of the  
| current locale if `LOCALETYPE(*LOCALE)` is specified on the compilation command. The behavior of this  
| function might also be affected by the `LC_UNI_COLLATE` category of the current locale if  
| `LOCALETYPE(*LOCALEUTF)` is specified on the compilation command. This function is not supported  
| when `LOCALETYPE(*LOCALEUCS2)` is specified on the compilation command. This function is not  
| available when `LOCALETYPE(*CLD)` is specified on the compilation command. For more information,  
| see “Understanding CCSIDs and Locales” on page 524.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `wcsxfrm()` function transforms the wide-character string pointed to by `wcs2` to values which represent character collating weights and places the resulting wide-character string into the array pointed to by `wcs1`.

### Return Value

The `wcsxfrm()` function returns the length of the transformed wide-character string (not including the ending null wide character code). If the value returned is *n* or more, the contents of the array pointed to by `wcs1` are indeterminate.

If `wcsxfrm()` is unsuccessful, `errno` is changed. The value of `errno` may be set to `EINVAL` (the `wcs1` or `wcs2` arguments contain characters which are not available in the current locale).

### Example that uses `wcsxfrm()`

```
#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\n", length);
    printf("\n\n%S\n\n transformed according", wcs);
    printf(" to this program's locale. \n");
}
```

### Related Information

- “`strxfrm()` — Transform String” on page 404
- “`<wchar.h>`” on page 18

---

## wctob() — Convert Wide Character to Byte

### Format

```
#include <stdio.h>
#include <wchar.h>
int wctob(wint_t wc);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the `LC_CTYPE` category of the current locale. The behavior might also be affected by the `LC_UNI_CTYPE` category of the current locale if `LOCALETYPE(*LOCALEUCS2)` or `LOCALETYPE(*LOCALEUTF)` is specified on the compilation command. This function is not available when `LOCALETYPE(*CLD)` is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

## Description

| The `wctob()` function determines whether `wc` corresponds to a member of the extended character set,  
| whose multibyte character has a length of 1 byte when in the initial shift state.

## Return Value

If `c` corresponds to a multibyte character with a length of 1 byte, the `wctob()` function returns the single-byte representation. Otherwise, it returns EOF.

If a conversion error occurs, `errno` may be set to `ECONVERT`.

## Example that uses `wctob()`

This example uses the `wctob()` function to test if the wide character A is a valid single-byte character.

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wint_t wc = L'A';

    if (wctob(wc) == wc)
        printf("%lc is a valid single byte character\n", wc);
    else
        printf("%lc is not a valid single byte character\n", wc);
    return 0;

    /*****
    The output should be similar to:

    A is a valid single byte character
    *****/
}
```

## Related Information

- “`mbtowc()` — Convert Multibyte Character to a Wide Character” on page 211
- “`wctomb()` — Convert Wide Character to Multibyte Character”
- “`wcstombs()` — Convert Wide-Character String to Multibyte String” on page 483
- “`<wchar.h>`” on page 18

---

## `wctomb()` — Convert Wide Character to Multibyte Character

### Format

```
#include <stdlib.h>
int wctomb(char *string, wchar_t character);
```

**Language Level:** ANSI

**Threadsafe:** No. Use `wcrtomb()` instead.

| **Locale Sensitive:** The behavior of this function might be affected by the `LC_CTYPE` category of the  
| current locale. The behavior might also be affected by the `LC_UNI_CTYPE` category of the current locale  
| if `LOCALETYPE(*LOCALEUCS2)` or `LOCALETYPE(*LOCALEUTF)` is specified on the compilation  
| command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

## Description

The `wctomb()` function 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, the `wctomb()` function stores `MB_CUR_MAX` characters in *string*.

The conversion of the wide character is the same as described in `wcstombs()`. See this function for a Unicode example.

## Return Value

The `wctomb()` function returns the length in bytes of the multibyte character. The value -1 is returned if *character* is not a valid multibyte character. If *string* is a NULL pointer, the `wctomb()` function returns nonzero if shift-dependent encoding is used, or 0 otherwise.

If a conversion error occurs, `errno` may be set to `ECONVERT`.

## Example that uses `wctomb()`

This example converts the wide character `c` to a multibyte character.

```
| #include <stdio.h>
| #include <stdlib.h>
| #include <wchar.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 should be similar to: *****/
|
| The number of bytes that comprise the multibyte character is 1
| And the converted string is "c"
| */
```

## Related Information

- “`mbtowc()` — Convert Multibyte Character to a Wide Character” on page 211
- “`wcslen()` — Calculate Length of Wide-Character String” on page 461
- “`wcrtomb()` — Convert a Wide Character to a Multibyte Character (Restartable)” on page 446
- “`wcstombs()` — Convert Wide-Character String to Multibyte String” on page 483
- “`wcsrombs()` — Convert Wide Character String to Multibyte String (Restartable)” on page 473
- “`<stdlib.h>`” on page 17

---

## `wctrans()` — Get Handle for Character Mapping

### Format

```
#include <wctype.h>
wctrans_t wctrans(const char *property);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

### Description

| The `wctrans()` function returns a value with type `wctrans_t`. This value describes a mapping between wide characters. The string argument *property* is a wide character mapping name. The `wctrans_t` equivalent of the wide character mapping name is returned by this function. The `tolower` and `toupper` wide character mapping names are defined in all locales.

### Return Value

| If *property* is a valid wide character mapping name, the `wctrans()` function returns a nonzero value that is valid as the second argument to the `towctrans()` function. Otherwise, it returns 0.

### Example that uses `wctrans()`

This example translates the lowercase alphabet to uppercase, and back to lowercase.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wchar.h>
#include <wctype.h>

int main()
{
    char *alpha = "abcdefghijklmnopqrstuvwxy";
    char *tocase[2] = {"toupper", "tolower"};
    wchar_t *walpha;
    int i, j;
    size_t alphalen;

    alphalen = strlen(alpha)+1;
    walpha = (wchar_t *)malloc(sizeof(wchar_t)*alphalen);

    mbstowcs(walpha, alpha, 2*alphalen);

    for (i=0; i<2; ++i) {
        printf("Input string: %ls\n", walpha);
        for (j=0; j<strlen(alpha); ++j) {
            walpha[j] = (wchar_t)towctrans((wint_t)walpha[j], wctrans(tocase[i]));
        }
        printf("Output string: %ls\n", walpha);
        printf("\n");
    }
    return 0;
}
```

/\*\*\*\*\*\* Output should be similar to: \*\*\*\*\*/

```
Input string: abcdefghijklmnopqrstuvwxy
Output string: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Input string: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Output string: abcdefghijklmnopqrstuvwxy
```

\*\*\*\*\*/

### Related Information

- “towctrans() — Translate Wide Character” on page 417
- “<wctype.h>” on page 19

---

## wctype() — Get Handle for Character Property Classification

### Format

```
#include <wctype.h>
wctype_t wctype(const char *property);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

### Description

The `wctype()` function is defined for valid character class names. The *property* is a string that identifies a generic character class. These character class names are defined in all locales: `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `xdigit`. The function returns a value of type `wctype_t`, which can be used as the second argument to a call of the `iswctype()` function.

The `wctype()` function determines values of `wctype_t` according to rules of the coded character set that are defined by character type information in the program’s locale (category `LC_CTYPE`). Values that are returned by the `wctype()` are valid until a call to `setlocale()` that changes the category `LC_CTYPE`.

### Return Value

The `wctype()` function returns zero if the given property name is not valid. Otherwise, it returns a value of type `wctype_t` that can be used in calls to `iswctype()`.

**Example that uses `wctype()`**

```

#include <wchar.h>

#define UPPER_LIMIT 0xFF
int main(void)
{
    int wc;
    for (wc = 0; wc <= UPPER_LIMIT; wc++) {
        printf("%#4x ", wc);
        printf("%c", iswctype(wc, wctype("print")) ? wc : "
");
        printf("%s", iswctype(wc, wctype("alnum")) ? "AN" : "
");
        printf("%s", iswctype(wc, wctype("alpha")) ? "A" : "
");
        printf("%s", iswctype(wc, wctype("blank")) ? "B" : "
");
        printf("%s", iswctype(wc, wctype("cntrl")) ? "C" : "
");
        printf("%s", iswctype(wc, wctype("digit")) ? "D" : "
");
        printf("%s", iswctype(wc, wctype("graph")) ? "G" : "
");
        printf("%s", iswctype(wc, wctype("lower")) ? "L" : "
");
        printf("%s", iswctype(wc, wctype("punct")) ? "PU" : "
");
        printf("%s", iswctype(wc, wctype("space")) ? "S" : "
");
        printf("%s", iswctype(wc, wctype("print")) ? "PR" : "
");
        printf("%s", iswctype(wc, wctype("upper")) ? "U" : "
");
        printf("%s", iswctype(wc, wctype("xdigit")) ? "X" : "
");
        putchar('\n');
    }
    return 0;
}
/*****
The output should be similar to :
:
0x1f          C
0x20          B          S          PR
0x21  !          G          PU          PR
0x22  "          G          PU          PR
0x23  #          G          PU          PR
0x24  $          G          PU          PR
0x25  %          G          PU          PR
0x26  &          G          PU          PR
0x27  '          G          PU          PR
0x28  (          G          PU          PR
0x29  )          G          PU          PR
0x2a  *          G          PU          PR
0x2b  +          G          PU          PR
0x2c  ,          G          PU          PR
0x2d  -          G          PU          PR
0x2e  .          G          PU          PR
0x2f  /          G          PU          PR
0x30  0  AN          D  G          PR  X
0x31  1  AN          D  G          PR  X
0x32  2  AN          D  G          PR  X
0x33  3  AN          D  G          PR  X
0x34  4  AN          D  G          PR  X
0x35  5  AN          D  G          PR  X
:
*****/
}

```

## Related Information

- “<wchar.h>” on page 18
- “<wctype.h>” on page 19

---

## wcwidth() — Determine the Display Width of a Wide Character

### Format

```
#include <wchar.h>
int wcwidth (const wint_t wc);
```

**Language Level:** XPG4

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the LC\_CTYPE category of the  
| current locale. The behavior might also be affected by the LC\_UNI\_CTYPE category of the current locale  
| if LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation  
| command. This function is not available when LOCALETYPE(\*CLD) is specified on the compilation  
| command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `wcwidth()` function 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 value of `errno` may be set to `EINVAL` (non-printing wide character).

### Return Value

The `wcwidth()` function either returns:

- 0, if `wc` is a null wide character; or
- the number of printing position occupied by `wc`; or
- -1, if `wc` is not a printing wide character.

### Example that uses `wcwidth()`

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
    wint_t wc = L'A';

    printf("%lc has a width of %d\n", wc, wcwidth(wc));
    return 0;

    /*****
    The output should be similar to :
    A has a width of 1
    *****/
}
```

## Related Information

- “wcswidth() — Determine the Display Width of a Wide Character String” on page 489
- “<wchar.h>” on page 18

---

## wfopen() —Open Files

### Format

```
#include <ifs.h>
FILE * wfopen(const wchar_t *filename,const wchar_t *mode);
```

**Language Level:** ILE C Extension

**Threadsafe:** Yes

| **Locale Sensitive:** This function is only available when LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. For more information, see “Understanding CCSIDs and Locales” on page 524.

| **Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified on the compilation command.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The wfopen() function works like the fopen() function, except:

- wfopen() accepts file name and mode as wide characters.
- The default CCSID for files opened with wfopen() (used when the ccsid=value, o\_ccsid=value, and codepage=value keywords are not specified) is UCS2 when LOCALETYPE(\*LOCALEUCS2) is specified on the compilation command. The default CCSID is UTF-32 when LOCALETYPE(\*LOCALEUTF) is specified on the compilation command.

---

## wmemchr() —Locate Wide Character in Wide-Character Buffer

### Format

```
#include <wchar.h>
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The wmemchr() function locates the first occurrence of *c* in the initial *n* wide characters of the object pointed to by *s*. If *n* has the value 0, the wmemchr() function finds no occurrence of *c*, and returns a NULL pointer.

### Return Value

The wmemchr() function returns a pointer to the located wide character, or a NULL pointer if the wide character does not occur in the object.

**Example that uses wmemchr()**

This example finds the first occurrence of 'A' in the wide-character string.

```
#include <stdio.h>
#include <wchar.h>

main()
{
    wchar_t *in = L"1234ABCD";
    wchar_t *ptr;
    wchar_t fnd = L'A';

    printf("\nEXPECTED: ABCD");
    ptr = wmemchr(in, L'A', 6);
    if (ptr == NULL)
        printf("\n** ERROR ** ptr is NULL, char L'A' not found\n");
    else
        printf("\nRECEIVED: %ls \n", ptr);
}
```

### Related Information

- “memchr() — Search Buffer” on page 212
- “strchr() — Search for Character” on page 359
- “wcschr() — Search for Wide Character” on page 452
- “wmemcmp() — Compare Wide-Character Buffers”
- “wmemcpy() — Copy Wide-Character Buffer” on page 500
- “wmemmove() — Copy Wide-Character Buffer” on page 501
- “wmemset() — Set Wide Character Buffer to a Value” on page 502
- “<wchar.h>” on page 18

---

## wmemcmp() — Compare Wide-Character Buffers

### Format

```
#include <wchar.h>
int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `wmemcmp()` function compares the first  $n$  wide characters of the object pointed to by  $s1$  to the first  $n$  wide characters of the object pointed to by  $s2$ . If  $n$  has the value 0, the `wmemcmp()` function returns 0.

### Return Value

The `wmemcmp()` function returns a value according to the relationship between the two strings,  $s1$  and  $s2$ :

Integer Value	Meaning
Less than 0	$s1$ less than $s2$
0	$s1$ equal to $s2$
Greater than 0	$s1$ greater than $s2$

## Example that uses wmemcmp()

This example compares the wide-character string in to out using the wmemcmp() function.

```
#include <wchar.h>
#include <stdio.h>
#include <locale.h>

main()
{
    int rc;
    wchar_t *in = L"12345678";
    wchar_t *out = L"12AAAAAB";
    setlocale(LC_ALL, "POSIX");

    printf("\nGREATER is the expected result");
    rc = wmemcmp(in, out, 3);
    if (rc == 0)
        printf("\nArrays are EQUAL %ls %ls \n", in, out);
    else
    {
        if (rc > 0)
            printf("\nArray %ls GREATER than %ls \n", in, out);
        else
            printf("\nArray %ls LESS than %ls \n", in, out);
    }

    /*****
    The output should be:

    GREATER is the expected result
    Array 12345678 GREATER than 12AAAAAB
    *****/
}
```

## Related Information

- “memcmp() — Compare Buffers” on page 213
- “strcmp() — Compare Strings” on page 360
- “wcscmp() — Compare Wide-Character Strings” on page 453
- “wcsncmp() — Compare Wide-Character Strings” on page 464
- “wmemchr() — Locate Wide Character in Wide-Character Buffer” on page 498
- “wmemcpy() — Copy Wide-Character Buffer”
- “wmemmove() — Copy Wide-Character Buffer” on page 501
- “wmemset() — Set Wide Character Buffer to a Value” on page 502
- “<wchar.h>” on page 18

---

## wmemcpy() — Copy Wide-Character Buffer

### Format

```
#include <wchar.h>
wchar_t *wmemcpy(wchar_t *s1, const wchar_t *s2, size_t n);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

The `wmemcpy()` function copies *n* wide characters from the object pointed to by *s2* to the object pointed to by *s1*. If *s1* and *s2* overlap, the result of the copy is unpredictable. If *n* has the value 0, the `wmemcpy()` function copies 0 wide characters.

### Return Value

The `wmemcpy()` function returns the value of *s1*.

### Example that uses `wmemcpy()`

This example copies the first four characters from `out` to `in`. In the expected output, the first four characters in both strings will be "ABCD".

```
#include <wchar.h>
#include <stdio.h>

main()
{
    wchar_t *in = L"12345678";
    wchar_t *out = L"ABCDEFGH";
    wchar_t *ptr;

    printf("\nExpected result: First 4 chars of in change");
    printf(" and are the same as first 4 chars of out");
    ptr = wmemcpy(in, out, 4);
    if (ptr == in)
        printf("\nArray in %ls array out %ls \n", in, out);
    else
    {
        printf("\n*** ERROR ***");
        printf(" returned pointer wrong");
    }
}
```

### Related Information

- “`memcpy()` — Copy Bytes” on page 214
- “`strcpy()` — Copy Strings” on page 364
- “`strncpy()` — Copy Strings” on page 380
- “`wscpy()` — Copy Wide-Character Strings” on page 456
- “`wcsncpy()` — Copy Wide-Character Strings” on page 466
- “`wmemchr()` — Locate Wide Character in Wide-Character Buffer” on page 498
- “`wmemcmp()` — Compare Wide-Character Buffers” on page 499
- “`wmemmove()` — Copy Wide-Character Buffer”
- “`wmemset()` — Set Wide Character Buffer to a Value” on page 502
- “`<wchar.h>`” on page 18

---

## `wmemmove()` — Copy Wide-Character Buffer

### Format

```
#include <wchar.h>
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

## Description

The `wmemmove()` function copies  $n$  wide characters from the object pointed to by  $s2$  to the object pointed to by  $s1$ . Copying takes place as if the  $n$  wide characters from the object pointed to by  $s2$  are first copied into a temporary array, of  $n$  wide characters, that does not overlap the objects pointed to by  $s1$  or  $s2$ . Then, the `wmemmove()` function copies the  $n$  wide characters from the temporary array into the object pointed to by  $s1$ . If  $n$  has the value 0, the `wmemmove()` function copies 0 wide characters.

## Return Value

The `wmemmove()` function returns the value of  $s1$ .

## Example that uses `wmemmove()`

This example copies the first five characters in a string to overlay the last five characters in the same string. Since the string is only nine characters long, the source and target overlap.

```
#include <wchar.h>
#include <stdio.h>

void main()
{
    wchar_t *theString = L"ABCDEFGH I";

    printf("\nThe original string: %ls \n", theString);
    wmemmove(theString+4, theString, 5);
    printf("\nThe string after wmemmove: %ls \n", theString);

    return;

    /*****
    The output should be:

    The original string: ABCDEFGH I
    The string after wmemmove: ABCDABCDE
    *****/
}
```

## Related Information

- “`memmove()` — Copy Bytes” on page 217
- “`wmemchr()` — Locate Wide Character in Wide-Character Buffer” on page 498
- “`wmemcpy()` — Copy Wide-Character Buffer” on page 500
- “`wmemcmp()` — Compare Wide-Character Buffers” on page 499
- “`wmemset()` — Set Wide Character Buffer to a Value”
- “`<wchar.h>`” on page 18

---

## `wmemset()` — Set Wide Character Buffer to a Value

### Format

```
#include <wchar.h>
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

## Description

The `wmemset()` function copies the value of `c` into each of the first `n` wide characters of the object pointed to by `s`. If `n` has the value 0, the `wmemset()` function copies 0 wide characters.

## Return Value

The `wmemset()` function returns the value of `s`.

## Example that uses `wmemset()`

This example sets the first 6 wide characters to the wide character 'A'.

```
#include <wchar.h>
#include <stdio.h>

void main()
{
    wchar_t *in = L"1234ABCD";
    wchar_t *ptr;

    printf("\nEXPECTED: AAAAAACD");
    ptr = wmemset(in, L'A', 6);
    if (ptr == in)
        printf("\nResults returned - %ls \n", ptr);
    else
    {
        printf("\n** ERROR ** wrong pointer returned\n");
    }
}
```

## Related Information

- “`memset()` — Set Bytes to Value” on page 218
- “`wmemchr()` — Locate Wide Character in Wide-Character Buffer” on page 498
- “`wmemcpy()` — Copy Wide-Character Buffer” on page 500
- “`wmemcmp()` — Compare Wide-Character Buffers” on page 499
- “`wmemmove()` — Copy Wide-Character Buffer” on page 501
- “`<wchar.h>`” on page 18

---

## wprintf() — Format Data as Wide Characters and Print

### Format

```
#include <stdio.h>
int wprintf(const wchar_t *format,...);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the `LC_CTYPE` and `LC_NUMERIC`  
| categories of the current locale. The behavior might also be affected by the `LC_UNI_CTYPE` and  
| `LC_UNI_NUMERIC` categories of the current locale if `LOCALETYPE(*LOCALEUCS2)` or  
| `LOCALETYPE(*LOCALEUTF)` is specified on the compilation command. This function is not available  
| when `LOCALETYPE(*CLD)` is specified on the compilation command. For more information, see  
| “Understanding CCSIDs and Locales” on page 524.

| **Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified  
| on the compilation command.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

### Description

A `wprintf(format, ...)` is equivalent to `fprintf(stdout, format, ...)`.

### Return Value

The `wprintf()` function returns the number of wide characters transmitted. If an output error occurred, the `wprintf()` function returns a negative value.

### Example that uses `wprintf()`

This example prints the wide character *a*. Date and time may be formatted according to your locale’s representation. The output goes to `stdout`.

```
#include <wchar.h>
#include <stdarg.h>
#include <locale.h>

int main(void)
{
    setlocale(LC_ALL, "POSIX");
    wprintf (L"%c\n", L'a');
    return(0);
}

/* A long 'a' is written to stdout */
```

### Related Information

- “`printf()` — Print Formatted Characters” on page 229
- “`btowc()` — Convert Single Byte to Wide Character” on page 53
- “`mbrtowc()` — Convert a Multibyte Character to a Wide Character (Restartable)” on page 201
- “`vfwprintf()` — Format Argument Data as Wide Characters and Write to a Stream” on page 428
- “`fwprintf()` — Format Data as Wide Characters and Write to a Stream” on page 143
- “`vswprintf()` — Format and Write Wide Characters to Buffer” on page 439
- “`<wchar.h>`” on page 18

---

## wscanf() — Read Data Using Wide-Character Format String

### Format

```
#include <stdio.h>
int wscanf(const wchar_t *format,...);
```

**Language Level:** ANSI

**Threadsafe:** Yes.

| **Locale Sensitive:** The behavior of this function might be affected by the `LC_CTYPE` and `LC_NUMERIC`  
| categories of the current locale. The behavior might also be affected by the `LC_UNI_CTYPE` and  
| `LC_UNI_NUMERIC` categories of the current locale if `LOCALETYPE(*LOCALEUCS2)` or

| LOCALETYPE(\*LOCALEUTF) is specified on the compilation command. This function is not available  
| when LOCALETYPE(\*CLD) is specified on the compilation command. For more information, see  
| “Understanding CCSIDs and Locales” on page 524.

| **Integrated File System Interface:** This function is not available when SYSIFCOPT(\*NOIFSIO) is specified  
| on the compilation command.

| **Wide Character Function:** See “Wide Characters” on page 527 for more information.

## Description

The `wscanf()` function is equivalent to the `fscanf()` function with the argument `stdin` interposed before the arguments of the `wscanf()` function.

## Return Value

If an input failure occurs before any conversion, the `wscanf()` function returns the value of the macro `EOF`.

Otherwise, the `wscanf()` function returns the number of input items assigned. It can be fewer than provided for, or even zero, in the event of an early matching failure.

## Example that uses `wscanf()`

This example scans various types of data.

```
#include <stdio.h>
#include <wchar.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 (wscanf(L"%d %f %c %s", &i, &fp,&c, s) != 4)
        printf("Some fields were not assigned\n");
    else {
        printf("integer = %d\n", i);
        printf("real number = %f\n", fp);
        printf("character = %c\n", c);
        printf("string = %s\n", s);
    }
    return 0;

    /*****
    The output should be similar to:

    Enter an integer, a real number, a character and a string :
    12 2.5 a yes
    integer = 12
    real number = 2.500000
    character = a
    string = yes
    *****/
}
```

## Related Information

- “`fscanf()` — Read Formatted Data” on page 132
- “`fwprintf()` — Format Data as Wide Characters and Write to a Stream” on page 143

- “fwscanf() — Read Data from Stream Using Wide Character” on page 147
- “scanf() — Read Data” on page 330
- “sscanf() — Read Data” on page 355
- “swprintf() — Format and Write Wide Characters to Buffer” on page 406
- “swscanf() — Read Wide Character Data” on page 407
- “vfscanf() — Read Formatted Data” on page 427
- “vfwscanf() — Read Formatted Wide Character Data” on page 430
- “vscanf() — Read Formatted Data” on page 433
- “vsscanf() — Read Formatted Data” on page 437
- “vswscanf() — Read Formatted Wide Character Data” on page 441
- “vwscanf() — Read Formatted Wide Character Data” on page 445
- “wprintf() — Format Data as Wide Characters and Print” on page 503
- “<wchar.h>” on page 18

---

## Chapter 3. Runtime Considerations

This chapter provides the following information:

- Exception and condition management
- Interlanguage data type compatibility
- CCSID (Coded Character Set Identifier) source file conversion

---

### errno Macros

The following table lists which error macros the ILE C library functions can set.

Table 12. *errno* Macros

Error Macro	Description	Set by Function
EBADDATA	The message data is not valid.	perror, strerror
EBADF	The catalog descriptor is not valid.	catclose, catgets, clearerr, fgetc, fgetpos, fgets, fileno, freopen, fseek, fsetpos, getc, rewind
EBADKEYLN	The key length specified is not valid.	_Rreadk, _Rlocate
EBADMODE	The file mode specified is not valid.	fopen, freopen, _Ropen
EBADNAME	Bad file name specified.	fopen, freopen, _Ropen
EBADPOS	The position specified is not valid.	fsetpos
EBADSEEK	Bad offset for a seek operation.	fgetpos, fseek
EBUSY	The record or file is in use.	perror, strerror
ECONVERT	Conversion error.	wcstomb, wcswidth
EDOM	Domain error in math function.	acos, asin, atan2, cos, exp, fmod, gamma, hypot, j0, j1, jn, y0, y1, yn, log, log10, pow, sin, strtol, strtoul, sqrt, tan
EGETANDPUT	An illegal read operation occurred after a write operation.	fgetc, fread, getc, getchar
EILSEQ	The character sequence does not form a valid multibyte character.	fgetwc, fgetws, getwc, mblen, mbrlen, mbrtowc, mbsrtowcs, mbstowcs, mbtowc, printf, scanf, ungetwc, wcrctomb, wcsrtombs, wcstombs, wctomb, wcswidth, wcwidth
EINVAL	The signal is not valid.	printf, scanf, signal, swprintf, swscanf, wcstol, wcstoll, wcstoul, wcstoull
EIO	Consecutive calls of I/O occurred.	I/O
EIOERROR	A non-recoverable I/O error occurred.	All I/O functions
EIORECERR	A recoverable I/O error occurred.	All I/O functions
ENODEV	Operation attempted on a wrong device.	fgetpos, fsetpos, fseek, ftell, rewind
ENOENT	File or library is not found.	perror, strerror
ENOPOS	No record at specified position.	fsetpos

Table 12. *errno* Macros (continued)

Error Macro	Description	Set by Function
ENOREC	Record not found.	fread, perror, strerror
ENOTDLT	File is not opened for delete operations.	_Rdelete
ENOTOPEN	File is not opened.	clearerr, fclose, fflush, fgetpos, fopen, freopen, fseek, ftell, setbuf, setvbuf, _Ropen, _Rclose
ENOTREAD	File is not opened for read operations.	fgetc, fread, ungetc, _Rreadd, _Rreadf, _Rreadindv, _Rreadk, _Rreadl, _Rreadn, _Rreadnc, _Rreadp, _Rreads, _Rlocate
ENOTUPD	File is not opened for update operations.	_Rrslck, _Rupdate
ENOTWRITE	File is not opened for write operations.	fputc, fwrite, _Rwrite, _Rwrited, _Rwriterd
ENUMMBRS	More than 1 member.	ftell
ENUMRECS	Too many records.	ftell
EPAD	Padding occurred on a write operation.	fwrite
EPERM	Insufficient authorization for access.	perror, strerror
EPUTANDGET	An illegal write operation occurred after a read operation.	fputc, fwrite, fputs, putc, putchar
ERANGE	Range error in math function.	cos, cosh, gamma, exp, j0, j1, jn, y0, y1, yn, log, log10, ldexp, pow, sin, sinh, strtod, strtol, strtoul, tan, wcstol, wcstoll, wcstoul, wcstoull, wcstod
ERECIO	File is opened for record I/O, so character-at-a-time processing functions cannot be used.	fgetc, fgetpos, fputc, fread, fseek, fsetpos, ftell
ESTDERR	stderr cannot be opened.	feof, ferror, fgetpos, fputc, fseek, fsetpos, ftell, fwrite
ESTDIN	stdin cannot be opened.	fgetc, fgetpos, fread, fseek, fsetpos, ftell
ESTDOUT	stdout cannot be opened.	fgetpos, fputc, fseek, fsetpos, ftell, fwrite
ETRUNC	Truncation occurred on I/O operation.	Any I/O function that reads or writes a record sets <i>errno</i> to ETRUNC.

## errno Values for Integrated File System Enabled C Stream I/O

The following table describes the possible settings when using integrated file system enabled stream I/O.

Table 13. *errno* Values for Integrated File System Enabled C Stream I/O

C Stream Function	Possible <i>errno</i> Values
clearerr	EBADF
fclose	EAGAIN, EBADF, EIO, ESCANFAILURE, EUNKNOWN

Table 13. *errno* Values for Integrated File System Enabled C Stream I/O (continued)

C Stream Function	Possible <i>errno</i> Values
<code>feof</code>	EBADF
<code>ferror</code>	EBADF
<code>fflush</code>	EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD
<code>fgetc</code>	EBADF, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOMEM, EUNKNOWN, EGETANDPUT, EDOM, ENOTREAD,
<code>fgetpos</code>	EACCESS, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOSYSRSC, EUNATCH, EUNKNOWN
<code>fgets</code>	EBADF, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOMEM, EUNKNOWN, EGETANDPUT, EDOM, ENOTREAD
<code>fgetwc</code>	EBADF, EILSEQ
<code>fgetws</code>	EBADF, EILSEQ
<code>fopen</code>	EAGAIN, EBADNAME, EBADF, ECONVERT, EDAMAGE, EEXITS, EFAULT, EINVAL, EIO, EISDIR, ELOOP, ENOENT, ENOMEM, ENOSPC, ENOSYS, ENOSYSRSC, ENOTDIR, ESCANFAILURE
<code>fprintf</code>	EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD
<code>fputc</code>	EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD
<code>fputs</code>	EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD
<code>fread</code>	EBADF, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOMEM, EUNKNOWN, EGETANDPUT, EDOM, ENOTREAD
<code>freopen</code>	EACCES, EAGAIN, EBADNAME, EBADF, EBUSY, ECONVERT, EDAMAGE, EEXITS, EFAULT, EINVAL, EIO, EISDIR, ELOOP, EMFILE, ENAMETOOLONG, ENFILE, ENOENT, ENOMEM, ENOSPC, ENOSYS, ENOSYSRSC, ENOTDIR
<code>fscanf</code>	EBADF, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOMEM, EUNKNOWN, EGETANDPUT, EDOM, ENOTREAD
<code>fseek</code>	EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EINVAL, EIO, ENOENT, ENOSPC, ENOSYSRSC, ESPIPE, EUNKNOWN, EFAULT, EPERM, EUNATCH, EUNKNOWN
<code>fsetpos</code>	EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EINVAL, EIO, ENOENT, ENOSPC, ENOSYSRSC, ESPIPE, EUNKNOWN, EFAULT, EPERM, EUNATCH, EUNKNOWN
<code>ftell</code>	EACCESS, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOSYSRSC, EUNATCH, EUNKNOWN
<code>fwrite</code>	EACCESS, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOSYSRSC, EUNATCH, EUNKNOWN
<code>getc</code>	EBADF, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOMEM, EUNKNOWN, EGETANDPUT, EDOM, ENOTREAD
<code>getchar</code>	EBADF, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOMEM, EUNKNOWN, EGETANDPUT, EDOM, ENOTREAD
<code>gets</code>	EBADF, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOMEM, EUNKNOWN, EGETANDPUT, EDOM, ENOTREAD
<code>getwc</code>	EBADF, EILSEQ
<code>perror</code>	EBADF
<code>printf</code>	EACCES, EAGAIN, EBADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EILSEQ, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD

Table 13. *errno* Values for Integrated File System Enabled C Stream I/O (continued)

C Stream Function	Possible <i>errno</i> Values
putc	EACCES, EAGAIN, EBAADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD
putchar	EACCES, EAGAIN, EBAADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD
puts	EACCES, EAGAIN, EBAADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD
remove	EACCES, EAGAIN, EBADNAME, EBAADF, EBUSY, ECONVERT, EDAMAGE, EEXITS, EFAULT, EINVAL, EIO, EISDIR, ELOOP, ENAMETOOLONG, ENOENT, ENOMEM, ENOSPC, ENOTDIR, EPERM, EROOJB, EUNKNOWN, EXDEV
rename	EACCES, EAGAIN, EBADNAME, EBUSY, ECONVERT, EDAMAGE, EEXIST, EFAULT, EINVAL, EIO, EISDIR, ELOOP, ENAMETOOLONG, ENOTEMPTY, ENOENT, ENOMEM, ENOSPC, ENOTDIR, EMLINK, EPERM, EUNKNOWN, EXDEV
rewind	EACCES, EAGAIN, EBAADF, EBUSY, EDAMAGE, EINVAL, EIO, ENOENT, ENOSPC, ENOSYSRSC, ESPIPE, EUNKNOWN, EFAULT, EPERM, EUNATCH, EUNKNOWN
scanf	EBAADF, EACCES, EAGAIN, EBUSY, EDAMAGE, EFAULT, EILSEQ, EINVAL, EIO, ENOMEM, EUNKNOWN, EGETANDPUT, EDOM, ENOTREAD
setbuf	EBAADF, EINVAL, EIO
setvbuf	EBAADF, EINVAL, EIO
tmpfile	EACCES, EAGAIN, EBADNAME, EBAADF, EBUSY, ECONVERT, EDAMAGE, EEXITS, EFAULT, EINVAL, EIO, EISDIR, ELOOP, EMFILE, ENAMETOOLONG, ENFILE, ENOENT, ENOMEM, ENOSPC, ENOSYS, ENOSYSRSC, ENOTDIR, EPERM, EROOJB, EUNKNOWN, EXDEV
tmpnam	EACCESS, EAGAIN, EBAADF, EBUSY, EDAMAGE, EFAULT, EINVAL, EIO, ENOENT, ENOSYSRSC, EUNATCH, EUNKNOWN
ungetc	EBAADF, EIO
ungetwc	EBAADF, EILSEQ
vfprintf	EACCES, EAGAIN, EBAADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD
vprintf	EACCES, EAGAIN, EBAADF, EBUSY, EDAMAGE, EFAULT, EFBIG, EINVAL, EIO, ENOMEM, ENOSPC, ETRUNC, EUNKNOWN, EPUTANDGET, ENOTWRITE, EPAD

## Record Input and Output Error Macro to Exception Mapping

The following table describes what occurs if the signal SIGIO is raised. Only \*ESCAPE, \*NOTIFY, and \*STATUS messages are monitored.

Table 14. Record Input and Output Error Macro to Exception Mapping

Description	Messages (_EXCP_MSGID)	errno setting
*STATUS and *NOTIFY	CPF4001 to CPF40FF, CPF4401 to CPF44FF, CPF4901 to CPF49FF, CPF5004	errno is not set, a default reply is returned to the operating system.
Recoverable I/O error	CPF4701 to CPF47FF, CPF4801 to CPF48FF, CPF5001 to CPF5003, CPF5005 to CPF50FF,	EIORECERR

Table 14. Record Input and Output Error Macro to Exception Mapping (continued)

Description	Messages (_EXCP_MSGID)	errno setting
Non-recoverable I/O error <sup>2</sup>	CPF4101 to CPF41FF, CPF4201 to CPF42FF, CPF4301 to CPF43FF, CPF4501 to CPF45FF, CPF4601 to CPF46FF, CPF5101 to CPF51FF, CPF5201 to CPF52FF, CPF5301 to CPF53FF, CPF5401 to CPF54FF, CPF5501 to CPF55FF, CPF5601 to CPF56FF	EIOERROR
Truncation occurred at I/O operation	C2M3003	ETRUNC
File is not opened	C2M3004	ENOTOPEN
File is not opened for read operations	C2M3005	ENOTREAD
File is not opened for write operations	C2M3009	ENOTWRITE
Bad file name specified	C2M3014	EBADNAME
The file mode specified is not valid	C2M3015	EBADMODE
File is not opened for update operations	C2M3041	ENOTUPD
File is not opened for delete operations	C2M3042	ENOTDLT
The key length specified is not valid	C2M3044	EBADKEYLN
A non-recoverable I/O error occurred	C2M3101	EIOERROR
A recoverable I/O error occurred	C2M3102	EIORECERR
<p><b>Note:</b></p> <ul style="list-style-type: none"> <li>• <sup>1</sup> The error is percolated to the user, therefore the user's direct monitor handlers, ILE C condition handlers and signal handler may get control. The initial setting for SIGIO is SIG_IGN.</li> <li>• <sup>2</sup> The type of device determines whether the error is recoverable or not recoverable. The following IBM publications contain information about recoverable and non-recoverable system exceptions for each specific file type: <ul style="list-style-type: none"> <li>– ICF Programming</li> <li>– ADTS/400: Advanced Printer Function</li> <li>– Application Display Programming</li> <li>– Database Programming</li> </ul> </li> </ul>		

## Signal Handling Action Definitions

The following table shows the initial state of the C signal values and their handling action definitions when SYSIFCOPT(\*NOASYNCSIGNAL) is specified on the compilation command. SIG\_DFL always percolates the condition to the handler. Resume indicates the exception is handled, and the application continues.

Table 15. Handling Action Definitions for Signal Values

Signal Value	Initial State	SIG_DFL	SIG_IGN	Return from Handler
SIGABRT <sup>1</sup>	SIG_DFL	Percolate	Ignore	Resume
SIGALL <sup>2</sup>	SIG_DFL	Percolate	Ignore	Resume
SIGFPE	SIG_DFL	Percolate	Ignore <sup>3</sup>	Resume <sup>4</sup>
SIGILL	SIG_DFL	Percolate	Ignore <sup>3</sup>	Resume <sup>4</sup>

Table 15. Handling Action Definitions for Signal Values (continued)

Signal Value	Initial State	SIG_DFL	SIG_IGN	Return from Handler
SIGINT	SIG_DFL	Percolate	Ignore	Resume
SIGIO	SIG_IGN	Percolate	Ignore	Resume
SIGOTHER	SIG_DFL	Percolate	Ignore <sup>3</sup>	Resume <sup>4</sup>
SIGSEGV	SIG_DFL	Percolate	Ignore <sup>3</sup>	Resume <sup>4</sup>
SIGTERM	SIG_DFL	Percolate	Ignore	Resume
SIGUSR1	SIG_DFL	Percolate	Ignore	Resume
SIGUSR2	SIG_DFL	Percolate	Ignore	Resume

**Note:**

- <sup>1</sup> Can only be signaled by the raise() function or the abort() function
- <sup>2</sup> SIGALL cannot be signaled by the raise() function.
- <sup>3</sup> If the value of the signal is SIGFPE, SIGILL or SIGSEGV the behavior is undefined.
- <sup>4</sup> If the signal is hardware-generated, then the behavior undefined.

The following table shows the initial state of the C signal values and their handling action definitions with SYSIFCOPT(\*ASYNCSIGNAL) is specified on the compilation command.

Table 16. Default Actions for Signal Values

Value	Default Action	Meaning
SIGABRT	2	Abnormal termination.
SIGFPE	2	Arithmetic exceptions that are not masked, such as overflow, division by zero, and incorrect operation.
SIGILL	2	Detection of an incorrect function image.
SIGINT	2	Interactive attention.
SIGSEGV	2	Incorrect access to storage.
SIGTERM	2	Termination request sent to the program.
SIGUSR1	2	Intended for use by user applications.
SIGUSR2	2	Intended for use by user applications.
SIGALRM	2	A timeout signal that is sent by alarm().
SIGHUP	2	A controlling terminal is hung up, or the controlling process ended.
SIGKILL	1	A termination signal that cannot be caught or ignored.
SIGPIPE	3	A write to a pipe that is not being read.
SIGQUIT	2	A quit signal for a terminal.
SIGCHLD	3	An ended or stopped child process. SIGCLD is an alias name for this signal.
SIGCONT	5	If stopped, continue.
SIGSTOP	4	A stop signal that cannot be caught or ignored.
SIGTSTP	4	A stop signal for a terminal.
SIGTTIN	4	A background process attempted to read from a controlling terminal.
SIGTTOU	4	A background process attempted to write to a controlling terminal.
SIGIO	3	Completion of input or output.

Table 16. Default Actions for Signal Values (continued)

SIGURG	3	High bandwidth data is available at a socket.
SIGPOLL	2	Pollable event.
SIGBUS	2	Specification exception.
SIGPRE	2	Programming exception.
SIGSYS	2	Bad system call.
SIGTRAP	2	Trace or breakpoint trap.
SIGPROF	2	Profiling timer expired.
SIGVTALRM	2	Virtual timer expired.
SIGXCPU	2	Processor time limit exceeded.
SIGXFSZ	2	File size limit exceeded.
SIGDANGER	2	System crash is imminent.
SIGPCANCEL	2	Thread termination signal that cannot be caught or ignored.

Default Actions:

- 1 End the process immediately.
- 2 End the request.
- 3 Ignore the signal.
- 4 Stop the process.
- 5 Continue the process if it is currently stopped. Otherwise, ignore the signal.

## Signal to i5/OS Exception Mapping

The following table shows the system exception messages that are mapped to a signal. All \*ESCAPE exception messages are mapped to signals. The \*STATUS and \*NOTIFY messages that map to SIGIO as defined in Table 14 on page 510 are mapped to signals.

Table 17. Signal to i5/OS Exception Mapping

Signal	Message
SIGABRT	C2M1601
SIGALL	C2M1610 (if explicitly raised)
SIGFPE	C2M1602, MCH1201 to MCH1204, MCH1206 to MCH1215, MCH1221 to MCH1224, MCH1838 to MCH1839
SIGILL	C2M1603, MCH0401, MCH1002, MCH1004, MCH1205, MCH1216 to MCH1219, MCH1801 to MCH1802, MCH1807 to MCH1808, MCH1819 to MCH1820, MCH1824 to MCH1825, MCH1832, MCH1837, MCH1852, MCH1854 to MCH1857, MCH1867, MCH2003 to MCH2004, MCH2202, MCH2602, MCH2604, MCH2808, MCH2810 to MCH2811, MCH3201 to MCH3203, MCH4201 to MCH4211, MCH4213, MCH4296 to MCH4298, MCH4401 to MCH4403, MCH4406 to MCH4408, MCH4421, MCH4427 to MCH4428, MCH4801, MCH4804 to MCH4805, MCH5001 to MCH5003, MCH5401 to MCH5402, MCH5601, MCH6001 to MCH6002, MCH6201, MCH6208, MCH6216, MCH6220, MCH6403, MCH6601 to MCH6602, MCH6609 to MCH6612
SIGINT	C2M1604
SIGIO	C2M1609, See Table 14 on page 510 for the exception mappings.
SIGOTHER	C2M1611 (if explicitly raised)

Table 17. Signal to i5/OS Exception Mapping (continued)

Signal	Message
SIGSEGV	C2M1605, MCH0201, MCH0601 to MCH0606, MCH0801 to MCH0803, MCH1001, MCH1003, MCH1005 to MCH1006, MCH1220, MCH1401 to MCH1402, MCH1602, MCH1604 to MCH1605, MCH1668, MCH1803 to MCH1806, MCH1809 to MCH1811, MCH1813 to MCH1815, MCH1821 to MCH1823, MCH1826 to MCH1829, MCH1833, MCH1836, MCH1848, MCH1850, MCH1851, MCH1864 to MCH1866, MCH1898, MCH2001 to MCH2002, MCH2005 to MCH2006, MCH2201, MCH2203 to MCH2205, MCH2401, MCH2601, MCH2603, MCH2605, MCH2801 to MCH2804, MCH2806 to MCH2809, MCH3001, MCH3401 to MCH3408, MCH3410, MCH3601 to MCH3602, MCH3603 to MCH3604, MCH3802, MCH4001 to MCH4002, MCH4010, MCH4212, MCH4404 to MCH4405, MCH4416 to MCH4420, MCH4422 to MCH4426, MCH4429 to MCH4437, MCH4601, MCH4802 to MCH4803, MCH4806 to MCH4812, MCH5201 to MCH5204, MCH5602 to MCH5603, MCH5801 to MCH5804, MCH6203 to MCH6204, MCH6206, MCH6217 to MCH6219, MCH6221 to MCH6222, MCH6401 to MCH6402, MCH6404, MCH6603 to MCH6608, MCH6801
SIGTERM	C2M1606
SIGUSR1	C2M1607
SIGUSR2	C2M1608

## Cancel Handler Reason Codes

The following table lists the bits that are set in the reason code. If the activation group is to be stopped, then the **activation group is stopped** bit is also set in the reason code. These bits must be correlated to `_CNL_MASK_T` in `_CNL_Hndlr_Parms_T` in `<except.h>`. Column 2 contains the macro constant defined for the cancel reason mask in `<except.h>`.

Table 18. Determining Canceled Invocation Reason Codes

Function	Bits set in reason code	Rationale
Library routines		
exit	<code>_EXIT_VERB</code>	The definition of <code>exit</code> is normal end of processing, and therefore invocations canceled by this function is done with a reason code of <i>normal</i> .
abort	<code>_ABNORMAL_TERM</code> <code>_EXIT_VERB</code>	The definition of <code>abort</code> is abnormal end of processing, and therefore invocations canceled by this function are done with a reason code of <i>abnormal</i> .
longjmp	<code>_JUMP</code>	The general use of the <code>longjmp()</code> function is to return from an exception handler, although it may be used in non-exception situations as well. It is used as part of the "normal" path for a program, and therefore any invocations canceled because of it are cancelled with a reason code of <i>normal</i> .
Unhandled function check	<code>_ABNORMAL_TERM</code> <code>UNHANDLED_EXCP</code>	Not handling an exception which is an abnormal situation.
System APIs		
CEEMRCR	<code>_ABNORMAL_TERM</code> <code>_EXCP_SENT</code>	This API is only used during exception processing. It is typically used to cancel invocations where a resume is not possible, or at least the behavior would be undefined if control was resumed in them. Also, these invocations have had a chance to handle the exception but did not do so. Invocations canceled by this API are done with reason code of <i>abnormal</i> .

Table 18. Determining Canceled Invocation Reason Codes (continued)

Function	Bits set in reason code	Rationale
QMHSNDPM /QMHSNEM (escape messages) Message Handler APIs	_ABNORMAL_TERM _EXCP_SENT	All invocations down to the target invocation are canceled without any chance of handling the exception. The API topic contains information about these APIs.
i5/OS commands		
Process end	_ABNORMAL_TERM _PROCESS_TERM _AG_TERMINATING	Any externally initiated shutdown of an activation group is considered abnormal.
RCLACTGRP	_ABNORMAL_TERM _RCLRSC	The default is abnormal termination. The termination could be normal if a normal/abnormal flag is added to the command.

Table 19. Common Reason Code for Cancelling Invocations

Bit	Description	Header File Constant <except.h>
Bits 0	Reserved	
Bits 1	Invocation canceled due to sending exception message	_EXCP_SENT
Bits 2-15	Reserved	
Bit 16	0 - normal end of process 1 - abnormal end of process	_ABNORMAL_TERM
Bit 17	Activation Group is ending.	_AG_TERMINATING
Bit 18	Initiated by <i>Reclaim Activation Group (RCLACTGRP)</i>	_RCLRSC
Bit 19	Initiated by the process end.	_PROCESS_TERM
Bit 20	Initiated by an exit() function.	_EXIT_VERB
Bit 21	Initiated by an unhandled function check.	_UNHANDLED_EXCP
Bit 22	Invocation canceled due to a longjmp() function.	_JUMP
Bit 23	Invocation canceled due to a jump because of exception processing.	_JUMP_EXCP
Bits 24-31	Reserved (0)	

## Exception Classes

In a CL program, you can monitor for a selected group of exceptions, or a single exception, based on the **exception identifier**. The only class2 values the exception handler will monitor for are `_C2_MH_ESCAPE`, `_C2_MH_STATUS`, `_C2_MH_NOTIFY`, and `_C2_MH_FUNCTION_CHECK`. For more information about using the `#pragma` exception handler directive, see the *WebSphere Development Studio: ILE C/C++ Compiler Reference*. This table defines all the exception classes you can specify.

Table 20. Exception Classes

Bit position	Header File Constant in <except.h>	Exception class
0	_C1_BINARY_OVERFLOW	Binary overflow or divide by zero
1	_C1_DECIMAL_OVERFLOW	Decimal overflow or divide by zero
2	_C1_DECIMAL_DATA_ERROR	Decimal data error
3	_C1_FLOAT_OVERFLOW	Floating-point overflow or divide by zero

Table 20. Exception Classes (continued)

Bit position	Header File Constant in <except.h>	Exception class
4	_C1_FLOAT_UNDERFLOW	Floating-point underflow or inexact result
5	_C1_INVALID_FLOAT_OPERAND	Floating-point invalid operand or conversion error
6	_C1_OTHER_DATA_ERROR	Other data error, for example edit mask
7	_C1_SPECIFICATION_ERROR	Specification (operand alignment) error
8	_C1_POINTER_NOT_VALID	Pointer not set/pointer type invalid
9	_C1_OBJECT_NOT_FOUND	Object not found
10	_C1_OBJECT_DESTROYED	Object destroyed
11	_C1_ADDRESS_COMP_ERROR	Address computation underflow or overflow
12	_C1_SPACE_ALLOC_ERROR	Space not allocated at specified offset
13	_C1_DOMAIN_OR_STATE_VIOLATION	Domain/State protection violation
14	_C1_AUTHORIZATION_VIOLATION	Authorization violation
15	_C1_JAVA_THROWN_CLASS	Exception thrown for a Java class.
16-28	_C1_VLIC_RESERVED	VLIC reserved
29	_C1_OTHER_MI_EXCEPTION	Remaining MI-generated exceptions (other than function check)
30	_C1_MI_GEN_FC_OR_MC	MI-generated function check or machine check
31	_C1_MI_SIGEXP_EXCEPTION	Message generated via Signal Exception instruction
32-39	n/a	reserved
40	_C2_MH_ESCAPE	*ESCAPE
41	_C2_MH_NOTIFY	*NOTIFY
42	_C2_MH_STATUS	*STATUS
43	_C2_MH_FUNCTION_CHECK	function check
44-63	n/a	reserved

## Data Type Compatibility

Each high-level language has different data types. When you want to pass data between programs that are written in different languages, you must be aware of these differences.

Some data types in the ILE C programming language have no direct equivalent in other languages. However, you can simulate data types in other languages that use ILE C data types.

The following table shows the ILE C data type compatibility with ILE RPG.

Table 21. ILE C Data Type Compatibility with ILE RPG

ILE C declaration in prototype	ILE RPG D spec, columns 33 to 39	Length	Comments
char[n] char *	nA	n	An array of characters where n=1 to 32766.
char	1A	1	An Indicator that is a variable starting with *IN.
char[n]	nS 0	n	A zoned decimal.
char[2n]	nG	2n	A graphic added.
char[2n+2]	Not supported.	2n+2	A graphic data type.
_Packed struct {short i; char[n]}	Not supported.	n+2	A variable length field where i is the intended length and n is the maximum length.
char[n]	D	8, 10	A date field.
char[n]	T	8	A time field.
char[n]	Z	26	A timestamp field.
short int	5I 0	2	An integer field.
short unsigned int	5U 0	2	An unsigned integer field.
int	10I 0	4	An integer field.
unsigned int	10U 0	4	An unsigned integer field
long int	10I 0	4	An integer field.
long unsigned int	10U 0	4	An unsigned integer field.
struct {unsigned int : n}x;	Not supported.	4	A 4-byte unsigned integer, a bitfield.
float	Not supported.	4	A 4-byte floating point.
double	Not supported.	8	An 8-byte double.
long double	Not supported.	8	An 8-byte long double.
enum	Not supported.	1, 2, 4	Enumeration.
*	*	16	A pointer.
decimal(n,p)	nP p	n/2+1	A packed decimal. n must be less than or equal to 30.
union.element	<type> with keyword OVERLAY(longest field)	element length	An element of a union.
data_type[n]	<type> with keyword DIM(n)	16	An array to which C passes a pointer.
struct	data structure	n	A structure. Use the _Packed qualifier on the struct.
pointer to function	*  with keyword PROCPTR	16	A 16-byte pointer.

The following table shows the ILE C data type compatibility with ILE COBOL.

Table 22. ILE C Data Type Compatibility with ILE COBOL

ILE C declaration in prototype	ILE COBOL LINKAGE SECTION	Length	Comments
char[n] char *	PIC X(n).	n	An array of characters where n=1 to 3,000,000
char	PIC 1 INDIC ..	1	An indicator.
char[n]	PIC S9(n) DISPLAY	n	A zoned decimal.
wchar_t[n]	PIC G(n)	2n	A graphic data type.
_Packed struct {short i; char[n]}	05 VL-FIELD. 10 i PIC S9(4) COMP-4. 10 data PIC X(n).	n+2	A variable length field where i is the intended length and n is the maximum length.
char[n]	PIC X(n).	6	A date field.
char[n]	PIC X(n).	5	A day field.
char	PIC X.	1	A day-of-week field.
char[n]	PIC X(n).	8	A time field.
char[n]	PIC X(n).	26	A time stamp field.
short int	PIC S9(4) COMP-4.	2	A 2-byte signed integer with a range of -9999 to +9999.
short int	PIC S9(4) BINARY.	2	A 2-byte signed integer with a range of -9999 to +9999.
int	PIC S9(9) COMP-4.	4	A 4-byte signed integer with a range of -999999999 to +999999999.
int	PIC S9(9) BINARY.	4	A 4-byte signed integer with a range of -999999999 to +999999999.
int	USAGE IS INDEX	4	A 4-byte integer.
long int	PIC S9(9) COMP-4.	4	A 4-byte signed integer with a range of -999999999 to +999999999.
long int	PIC S9(9) BINARY.	4	A 4-byte signed integer with a range of -999999999 to +999999999.
struct {unsigned int : n}x;	PIC 9(9) COMP-4. PIC X(4).	4	Bitfields can be manipulated using hex literals.
float	Not supported.	4	A 4-byte floating point.
double	Not supported.	8	An 8-byte double.
long double	Not supported.	8	An 8-byte long double.
enum	Not supported.	1, 2, 4	Enumeration.
*	USAGE IS POINTER	16	A pointer.
decimal(n,p)	PIC S9(n-p)V9(p) COMP-3	n/2+1	A packed decimal.
decimal(n,p)	PIC S9(n-p) 9(p) PACKED-DECIMAL	n/2+1	A packed decimal.
union.element	REDEFINES	element length	An element of a union.
data_type[n]	OCCURS	16	An array to which C passes a pointer.

Table 22. ILE C Data Type Compatibility with ILE COBOL (continued)

ILE C declaration in prototype	ILE COBOL LINKAGE SECTION	Length	Comments
struct	01 record 05 field1 05 field2	n	A structure. Use the <code>_Packed</code> qualifier on the struct. Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
pointer to function	PROCEDURE-POINTER	16	A 16 byte pointer to a procedure.
Not supported.	PIC S9(18) COMP-4.	8	An 8 byte integer.
Not supported.	PIC S9(18) BINARY.	8	An 8 byte integer.

The following table shows the ILE C data type compatibility with ILE CL.

Table 23. ILE C Data Type Compatibility with ILE CL

ILE C declaration in prototype	CL	Length	Comments
char[n] char *	*CHAR LEN(&N)	n	An array of characters where n=1 to 32766. A null-terminated string. For example, CHGVAR &V1 VALUE (&V *TCAT X'00') where &V1 is one byte bigger than &V.
char	*LGL	1	Holds '1' or '0'.
<code>_Packed</code> struct {short i; char[n]}	Not supported.	n+2	A variable length field where i is the intended length and n is the maximum length.
integer types	Not supported.	1, 2, 4	A 1-, 2-, or 4- byte signed or unsigned integer.
float constants	CL constants only.	4	A 4- or 8- byte floating point.
decimal(n,p)	*DEC	n/2+1	A packed decimal. The limit of n is 15 and p is 9.
union.element	Not supported.	element length	An element of a union.
struct	Not supported.	n	A structure. Use the <code>_Packed</code> qualifier on the struct.
pointer to function	Not supported.	16	A 16-byte pointer.

The following table shows the ILE C data type compatibility with OPM RPG/400®.

Table 24. ILE C Data Type Compatibility with OPM RPG/400

ILE C declaration in prototype	OPM RPG/400 I spec, DS subfield columns spec	Length	Comments
char[n] char *	1 10	n	An array of characters where n=1 to 32766.
char	*INxxxx	1	An Indicator that is a variable starting with *IN.
char[n]	1 nd (d>=0)	n	A zoned decimal. The limit of n is 30.
char[2n+2]	Not supported.	2n+2	A graphic data type.

Table 24. ILE C Data Type Compatibility with OPM RPG/400 (continued)

ILE C declaration in prototype	OPM RPG/400 I spec, DS subfield columns spec	Length	Comments
_Packed struct {short i; char[n]}	Not supported.	n+2	A variable length field where i is the intended length and n is the maximum length.
char[n]	Not supported.	6, 8, 10	A date field.
char[n]	Not supported.	8	A time field.
char[n]	Not supported.	26	A time stamp field.
short int	B 1 20	2	A 2-byte signed integer with a range of -9999 to +9999.
int	B 1 40	4	A 4-byte signed integer with a range of -999999999 to +999999999.
long int	B 1 40	4	A 4-byte signed integer with a range of -999999999 to +999999999.
struct {unsigned int : n}x;	Not supported.	4	A 4-byte unsigned integer, a bitfield.
float	Not supported.	4	A 4-byte floating point.
double	Not supported.	8	An 8-byte double.
long double	Not supported.	8	An 8-byte long double.
enum	Not supported.	1, 2, 4	Enumeration.
*	Not supported.	16	A pointer.
decimal(n,p)	P 1 n/2+1d	n/2+1	A packed decimal. n must be less than or equal to 30.
union.element	data structure subfield	element length	An element of a union.
data_type[n]	E-SPEC array	16	An array to which C passes a pointer.
struct	data structure	n	A structure. Use the _Packed qualifier on the struct.
pointer to function	Not supported.	16	A 16 byte pointer.

The following table shows the ILE C data type compatibility with OPM COBOL/400®.

Table 25. ILE C Data Type Compatibility with OPM COBOL/400

ILE C declaration in prototype	OPM COBOL LINKAGE SECTION	Length	Comments
char[n] char *	PIC X(n).	n	An array of characters where n=1 to 3,000,000
char	PIC 1 INDIC ..	1	An indicator.
char[n]	PIC S9(n) USAGE IS DISPLAY	n	A zoned decimal. The limit of n is 18.
_Packed struct {short i; char[n]}	05 VL-FIELD. 10 i PIC S9(4) COMP-4. 10 data PIC X(n).	n+2	A variable length field where i is the intended length and n is the maximum length.
char[n]	PIC X(n).	6, 8, 10	A date field.
char[n]	PIC X(n).	8	A time field.
char[n]	PIC X(n).	26	A time stamp field.

Table 25. ILE C Data Type Compatibility with OPM COBOL/400 (continued)

ILE C declaration in prototype	OPM COBOL LINKAGE SECTION	Length	Comments
short int	PIC S9(4) COMP-4.	2	A 2 byte signed integer with a range of -9999 to +9999.
int	PIC S9(9) COMP-4.	4	A 4-byte signed integer with a range of -999999999 to +999999999.
long int	PIC S9(9) COMP-4.	4	A 4-byte signed integer with a range of -999999999 to +999999999.
struct {unsigned int : n}x;	PIC 9(9) COMP-4. PIC X(4).	4	Bitfields can be manipulated using hex literals.
float	Not supported.	4	A 4-byte floating point.
double	Not supported.	8	An 8-byte double.
long double	Not supported.	8	An 8-byte long double.
enum	Not supported.	1, 2, 4	Enumeration.
*	USAGE IS POINTER	16	A pointer.
decimal(n,p)	PIC S9(n-p)V9(p) COMP-3	n/2+1	A packed decimal. The limits of n and p are 18.
union.element	REDEFINES	element length	An element of a union.
data_type[n]	OCCURS	16	An array to which C passes a pointer.
struct	01 record	n	A structure. Use the <code>_Packed</code> qualifier on the struct. Structures passed should be passed as a pointer to the structure if you want to change the contents of the structure.
pointer to function	Not supported.	16	A 16-byte pointer.
Not supported.	PIC S9(18) COMP-4.	8	An 8 byte integer.

The following table shows the ILE C data type compatibility with CL.

Table 26. ILE C Data Type Compatibility with CL

ILE C declaration in prototype	CL	Length	Comments
char[n] char *	*CHAR LEN(&N)	n	An array of characters where n=1 to 32766. A null terminated string. For example, CHGVAR &V1 VALUE (&V *TCAT X'00') where &V1 is one byte bigger than &V. The limit of n is 9999.
char	*LGL	1	Holds '1' or '0'.
_Packed struct {short i; char[n]}	Not supported.	n+2	A variable length field where i is the intended length and n is the maximum length.
integer types	Not supported.	1, 2, 4	A 1-, 2- or 4- byte signed or unsigned integer.
float constants	CL constants only.	4	A 4- or 8- byte floating point.
decimal(n,p)	*DEC	n/2+1	A packed decimal. The limit of n is 15 and p is 9.

Table 26. ILE C Data Type Compatibility with CL (continued)

ILE C declaration in prototype	CL	Length	Comments
union.element	Not supported.	element length	An element of a union.
struct	Not supported.	n	A structure. Use the _Packed qualifier on the struct.
pointer to function	Not supported.	16	A 16-byte pointer.

The following table shows how arguments are passed from a command line CL call to an ILE C program.

Table 27. Arguments Passed From a Command Line CL Call to an ILE C Program

Command Line Argument	Argv Array	ILE C Arguments
	argv[0]	"LIB/PGMNAME"
	argv[1..255]	normal parameters
'123.4'	argv[1]	"123.4"
123.4	argv[2]	0000000123.40000D
'Hi'	argv[3]	"Hi"
Lo	argv[4]	"LO"

A CL character array (string) will not be NULL-ended when passed to an ILE C program. A C program that will receive such arguments from a CL program should not expect the strings to be NULL-ended. You can use the QCMDEXC to ensure that all the arguments will be NULL-ended.

The following table shows how CL constants are passed from a compiled CL program to an ILE C program.

Table 28. CL Constants Passed from a Compiled CL Program to an ILE C Program

Compile CL Program Argument	Argv Array	ILE C Arguments
	argv[0]	"LIB/PGMNAME"
	argv[1..255]	normal parameters
'123.4'	argv[1]	"123.4"
123.4	argv[2]	0000000123.40000D
'Hi'	argv[3]	"Hi"
Lo	argv[4]	"LO"

A command processing program (CPP) passes CL constants as defined in Table 28. You define an ILE C program as a command processing program when you create your own CL command with the Create Command (CRTCMD) command to call the ILE C program.

The following table shows how CL variables are passed from a compiled CL program to an ILE C program. All arguments are passed by reference from CL to C.

Table 29. CL Variables Passed from a Compiled CL Program to an ILE C Program

CL Variables	ILE C Arguments
DCL VAR(&v) TYPE(*CHAR) LEN(10) VALUE('123.4')	123.4
DCL VAR(&d) TYPE(*DEC) LEN(10) VALUE(123.4)	0000000123.40000D

Table 29. CL Variables Passed from a Compiled CL Program to an ILE C Program (continued)

CL Variables	ILE C Arguments
DCL VAR(&h) TYPE(*CHAR) LEN(10) VALUE('Hi')	Hi
DCL VAR(&i) TYPE(*CHAR) LEN(10) VALUE('Lo')	LO
DCL VAR(&j) TYPE(*LGL) LEN(1) VALUE('1')	1

CL variables and numeric constants are not passed to an ILE C program with null-ended strings. Character constants and logical literals are passed as null-ended strings, but are not padded with blanks. Numeric constraints such as packed decimals are passed as 15,5 (8 bytes).

## Runtime Character Set

Each EBCDIC CCSID consists of two character types: invariant characters and variant characters.

The following table identifies the hexadecimal representation of the invariant characters in the C character set.

Table 30. Invariant Characters

.	<	(	+	&	*	)	;
0x4b	0x4c	0x4d	0x4e	0x50	0x5c	0x5d	0x5e
-		,	%	_	>	?	:
0x60	0x6a	0x6b	0x6c	0x6d	0x6e	0x6f	0x7a
@	'	=	"	a-i	j-r	s-z	A-I
0x7c	0x7d	0x7e	0x7f	0x81 - 0x89	0x91 - 0x99	0xa2 - 0xa9	0xc1 - 0xc9
J-R	S-Z	0-9	'\a'	'\b'	'\t'	'\v'	'\f'
0xd1 - 0xd9	0xe2 - 0xe9	0xf0 - 0xf9	0x2f	0x16	0x05	0x0b	0x0c
'\r'	'\n'	' '					
0x0d	0x15	0x40					

**Note:** Not all EBCDIC character sets have all invariant characters at the invariant code points. Here are the exceptions:

- Code page 290, used in Japanese CCSIDs 290, 930, and 5026, has the lowercase Latin characters a-z in a nonstandard position.
- Code page 420, used in some Arabic CCSIDs, does not have the back quotation mark (`) whose hexadecimal value is 0x7a.
- Code page 423, used in some older Greek CCSIDs, does not have the ampersand (&) whose hexadecimal value is 0x50.
- Code pages 905 and 1026, both used in some Turkish CCSIDs, have a hexadecimal value of 0xfc for the double quotation mark instead of the invariant hexadecimal value of 0x7f.

The following table identifies the hexadecimal representation of the variant characters in the C character set for the most commonly used CCSIDs.

Table 31. Variant Characters in Different CCSIDs

CC-SID		!	~	\	`	#	~	[	]	^	{	}	/	€	\$
037	0x4f	0x5a	0x5f	0xe0	0x79	0x7b	0xa1	0xba	0xbb	0xb0	0xc0	0xd0	0x61	0x4a	0x5b
256	0xbb	0x4f	0xba	0xe0	0x79	0x7b	0xa1	0x4a	0x5a	0x5f	0xc0	0xd0	0x61	0xb0	0x5b

Table 31. Variant Characters in Different CCSIDs (continued)

CC-SID		!	-	\	`	#	~	[	]	^	{	}	/	€	\$
273	0xbb	0x4f	0xba	0xec	0x79	0x7b	0x59	0x63	0xfc	0x5f	0x43	0xdc	0x61	0xb0	0x5b
277	0xbb	0x4f	0xba	0xe0	0x79	0x4a	0xdc	0x9e	0x9f	0x5f	0x9c	0x47	0x61	0xb0	0x67
278	0xbb	0x4f	0xba	0x71	0x51	0x63	0xdc	0xb5	0x9f	0x5f	0x43	0x47	0x61	0xb2	0x67
280	0xbb	0x4f	0xba	0x48	0xdd	0xb1	0x58	0x90	0x51	0x5f	0x44	0x45	0x61	0xb0	0x5b
284	0x4f	0xbb	0x5f	0xe0	0x79	0x69	0xbd	0x4a	0x5a	0xba	0xc0	0xd0	0x61	0xb0	0x5b
285	0x4f	0x5a	0x5f	0xe0	0x79	0x7b	0xbc	0xb1	0xbb	0xba	0xc0	0xd0	0x61	0xb0	0x4a
297	0xbb	0x4f	0xba	0x48	0xa0	0xb1	0xbd	0x90	0x65	0x5f	0x51	0x54	0x61	0xb0	0x5b
500	0xbb	0x4f	0xba	0xe0	0x79	0x7b	0xa1	0x4a	0x5a	0x5f	0xc0	0xd0	0x61	0xb0	0x5b

See the *i5/OS* globalization topic for more information about coding variant characters in the other IBM CCSIDs.

## Understanding CCSIDs and Locales

### CCSIDs of Characters and Character Strings

Every character or character string has a CCSID associated with it. The CCSID of the character or character string depends on the origin of the data. You need to pay attention to the CCSID of a character or character string. It is also important that values are converted to the appropriate CCSID when required.

If `LOCALETYPE(*LOCALEUTF)` is not specified on the compilation command, the following assumptions are made:

- The CCSID of the job is the same as the CCSID of the `LC_CTYPE` category of the current locale.
- The CCSID of character literal values matches the CCSID of the `LC_CTYPE` category of the current locale.
- The CCSID of the `LC_CTYPE` category of the current locale is an EBCDIC CCSID.
- The CCSID that is used has all of the invariant characters in the proper positions, and some functions assume that certain variant characters have the same hexadecimal value as they would in CCSID 37.

When `LOCALETYPE(*LOCALEUTF)` is specified, most functions (unless otherwise specified) expect character data input in the CCSID of the `LC_CTYPE` category of the current locale, regardless of the source of the character data. See “Unicode Support” on page 530 for more information.

For more information about variant and invariant characters, see “Runtime Character Set” on page 523. For more information about CCSIDs, code pages, and other globalization concepts, see the *i5/OS globalization* topic.

### Character Literal CCSID

Character literal CCSID is the CCSID of the character and character string literals in compiled source code. If a programmer does not take special action, the CCSID of these literals is set to the CCSID of the source file. The CCSID of all the literals in a compilation unit can be changed by using the `TGTCCSID` option on the compilation command. The `#pragma convert` directive can be used to change the CCSID of character and character string literals within C or C++ source code. See *WebSphere Development Studio: ILE C/C++ Compiler Reference* for more information.

If `LOCALETYPE(*CLD)` or `LOCALETYPE(*LOCALE)` is specified on the compilation command, all wide character literals will be wide EBCDIC literals in the CCSID of the source file. If

| LOCALETYPE(\*LOCALEUCS2) is specified on the compilation command, all wide character literals will  
| be UCS-2 literals. If LOCALETYPE(\*LOCALEUTF) is specified on the compilation command, all wide  
| characters will be UTF-32 literals.

| The programmer must be aware of the CCSID of character literal values. The character literal CCSID  
| cannot be retrieved at run time.

### | **Job CCSID**

| The CCSID of the job is always an EBCDIC CCSID. ASCII and Unicode job CCSIDs are not supported.  
| Data read from files is sometimes in the job CCSID. Some functions (for example, `getenv()`) produce job  
| CCSID output; some functions (for example, `putenv()`) expect job CCSID input. The CCSID used most  
| often by the C runtime is the CCSID of the LC\_CTYPE category of the current locale. If the job CCSID  
| does not match the locale CCSID, conversion might be necessary.

| Using the JOBI0400 receiver variable format, the job CCSID value can be retrieved at run time using the  
| QUSRJOBI API. The Default Coded Character Set ID field contains the job CCSID value.

### | **File CCSID**

| When a file is opened, a CCSID is associated with it. Read operations of character and string values  
| return data in the CCSID of the file. Write operations to the file expect the data in the CCSID of the file.  
| The CCSID associated with a file when it is opened is dependent on the function that is used to open the  
| file:

- | • `catopen` function

| The CCSID associated with a catalog file that is opened using `catopen` depends on the content of the  
| `oflag` parameter. Two of the flags that can be specified for the `oflag` parameter are  
| `NL_CAT_JOB_MODE` and `NL_CAT_CTYPE_MODE`. These flags are mutually exclusive.

- | – If `NL_CAT_JOB_MODE` is specified, the job CCSID is associated with the file.
- | – If `NL_CAT_CTYPE_MODE` is specified, the CCSID of the LC\_CTYPE category of the current locale  
| is associated with the file.
- | – If neither flag is specified, no conversion takes place and the CCSID of the returned messages is the  
| same CCSID as that of the message file.

- | • `fdopen()` function

| – If `LOCALETYPE(*LOCALEUTF)` is not specified, then the default CCSID for a file is the job CCSID.  
| The keyword `ccsid=value`, `o_ccsid=value`, or `codepage=value` can be used in the mode string on the  
| file open command to change the CCSID associated with the file. `o_ccsid=value` is the recommended  
| keyword. The standard files are always associated with the default file CCSID, so they are  
| associated with the job CCSID.

| – If `LOCALETYPE(*LOCALEUTF)` is specified, then the default CCSID for a file is the CCSID of the  
| LC\_CTYPE category of the current locale when the `fopen()` function is called. The keywords  
| described in the previous paragraph can still be used to override the CCSID associated with the file.  
| The standard files are always associated with the default file CCSID, so they are associated with the  
| CCSID of the LC\_CTYPE category of the current locale when they are opened.

- | • `fopen()` and `freopen()` functions

- | – If `LOCALETYPE(*LOCALEUTF)` is not specified, the default CCSID for a file is the job CCSID.
  - | - If `SYSIFCOPT(*NOIFSIO)` is specified on the compilation command, the keyword `ccsid=value` can  
| be used in the mode string on the file open command to change the CCSID of data read from or  
| written to the file.
  - | - If `SYSIFCOPT(*NOIFSIO)` is not specified on the compilation command, the keyword `ccsid=value`,  
| `o_ccsid=value`, or `codepage=value` can be used in the mode string on the file open command to  
| change the CCSID associated with the file. `o_ccsid=value` is the recommended keyword.

| The standard files are always associated with the default file CCSID, so they are associated with the  
| job CCSID.

- | – If `LOCALETYPE(*LOCALEUTF)` is specified, then the default CCSID for a file is the CCSID of the `LC_CTYPE` category of the current locale when the `fopen()` or `freopen()` function is called. The keyword `ccsid=value`, `o_ccsid=value`, or `codepage=value` can still be used to override the CCSID associated with the file. The standard files are always associated with the default file CCSID, so they are associated with the CCSID of the `LC_CTYPE` category of the current locale when they are opened.
- | • `_Ropen()` function
  - | The default CCSID associated with a file opened with the `_Ropen()` function is the job CCSID. The `ccsid=value` keyword can be used in the mode parameter on the `_Ropen()` function to change the CCSID associated with the file.
- | • `wfopen()` function
  - | – If `LOCALETYPE(*LOCALEUCS2)` is specified, the default CCSID for a file is UCS-2. The keyword `ccsid=value`, `o_ccsid=value`, or `codepage=value` can be used in the mode string on the file open command to change the CCSID associated with the file. `o_ccsid=value` is the recommended keyword.
  - | – If `LOCALETYPE(*LOCALEUTF)` is specified, then the default CCSID for a file is UTF-32. The keywords described in the previous paragraph can still be used to override the CCSID associated with the file.

## | **Locale CCSID**

| A CCSID is associated with each category of the locale (see “`setlocale()` — Set Locale” on page 339 for a list of locale categories). The most commonly used CCSID from the locale is the CCSID associated with the `LC_CTYPE` category of the locale. Confusion might arise if different locale categories have different CCSID values, so it is recommended that all locale categories have the same CCSID value. You can retrieve the CCSID of the `LC_CTYPE` category of the current locale by using the `nl_langinfo()` function and specifying `CODESET` as the `nl_item`. Here are some additional locale CCSID details, broken down by `LOCALETYPE` option specified on the compilation command:

- | • `LOCALETYPE(*CLD)`
  - | `LOCALETYPE(*CLD)` is only supported by the ILE C compiler. Many POSIX functions are not supported when `LOCALETYPE(*CLD)` is specified. One benefit of the `LOCALETYPE(*CLD)` option is that all `*CLD` locales are CCSID 37. A limited number of locale objects are shipped with the system that can be used with `LOCALETYPE(*CLD)`. These objects all have the object type `*CLD`. To get a list of `*CLD` locale objects, use the following command:

```
| WRKOBJ OBJ(QSYS/*ALL) OBJTYPE(*CLD)
```

| For more information about `*CLD` locales, see *WebSphere Development Studio: ILE C/C++ Compiler Reference*.

- | • `LOCALETYPE(*LOCALE)`
  - | This is the default `LOCALETYPE` setting for the ILE C compiler and ILE C++ compiler. The default locale value usually has a CCSID that is equal to the job CCSID. A wide variety of locale objects exists for this setting. These locale objects have the `*LOCALE` object type. The `LOCALETYPE(*LOCALE)` option supports a larger number of CCSIDs and a larger number of functions than the `LOCALETYPE(*CLD)` option.
- | • `LOCALETYPE(*LOCALEUCS2)`
  - | This setting introduces a new set of locale categories for UCS-2 characters. These locale category names begin with the `LC_UNI_` substring. The original locale categories are still present, and all the preceding notes for `LOCALETYPE(*LOCALE)` apply to `LOCALETYPE(*LOCALEUCS2)`. This setting causes wide characters to be interpreted as UCS-2 characters instead of wide EBCDIC characters. For more information, see “Unicode Support” on page 530.
- | • `LOCALETYPE(*LOCALEUTF)`
  - | The CCSID of the non-wide locale categories is UTF-8 (CCSID 1208) by default, but it can be changed to have any single-byte or multibyte CCSID. The CCSID of the wide character (`LC_UNI_*`) locale

categories is UTF-32. This setting includes limited CCSID neutrality. `LOCALETYPE(*LOCALEUTF)` uses locale objects of type `*LOCALE`. For more information, see “Unicode Support” on page 530.

## Wide Characters

The ILE C/C++ compilers support the following:

- If `LOCALETYPE(*CLD)` or `LOCALETYPE(*LOCALE)` is specified on the compilation command, wide characters are treated as 2-byte wide EBCDIC characters.
- If `LOCALETYPE(*LOCALEUCS2)` is specified on the compilation command, wide characters are treated as 2-byte UCS-2 characters.
- If `LOCALETYPE(*LOCALEUTF)` is specified on the compilation command, wide characters are treated as 4-byte UTF-32 characters.

When EBCDIC wide characters are used, the CCSID of the EBCDIC characters depends on the CCSID of the `LC_CTYPE` category of the current locale. See “Unicode Support” on page 530 for more information about Unicode characters.

## Wide Character Conversions to and from Single-Byte or Multibyte Characters

The character conversion routines examine the CCSID setting for the `LC_CTYPE` category of the current locale to determine whether single-byte or multibyte characters are expected for the conversion from or to wide characters.

The handling of wide character conversions (to and from single-byte or multibyte character strings) is dependent on the `LOCALETYPE` parameter value specified on the compilation command. The handling depends on the shift state of the single-byte or multibyte character string. The `mbtowc`, `mbstowcs`, `wctomb`, and `wcstombs` functions maintain an internal shift state variable. The `mbrtowc`, `mbsrtowcs`, `wcrtomb`, and `wcsrombs` functions allow the shift state variable to be passed as a parameter. The second set of functions is recommended because they are more versatile and are also threadsafe.

**LOCALETYPE(\*CLD) and LOCALETYPE(\*LOCALE) behavior:** When converting from a single-byte CCSID to wide EBCDIC, the wide EBCDIC character is constructed by adding a zero byte to the single-byte character. For example, the single-byte CCSID 37 character A (hexadecimal value 0xC1) would have the hexadecimal value 0x00C1 when it is converted to a wide EBCDIC character.

When converting from a multibyte CCSID to wide EBCDIC, the conversion method depends on the shift state of the input string. In the initial shift state, characters are read exactly as if they were single-byte characters until a shift-out character (hexadecimal value 0x0E) is read. This character indicates a shift to double-byte shift state. In the double-byte shift state, 2 bytes are read at a time: the first byte makes up the first byte of the EBCDIC wide character and the second byte will be the second byte of the EBCDIC wide character. If the shift-in character (hexadecimal value 0x0F) is encountered, the function returns to the initial shift state parsing. For example, the multibyte string represented by the hexadecimal value C10E43DA0FC2 is translated to the EBCDIC wide character string with the hexadecimal value 00C143DA00C2.

When converting from wide EBCDIC to a single-byte CCSID, if the character has a hexadecimal value greater than 0x00FF, EOF is returned; otherwise, the top byte is truncated and the lower byte is returned. For example, the wide EBCDIC character with the hexadecimal value 0x00C1 is converted to the single-byte character whose hexadecimal value is 0xC1.

When converting from wide EBCDIC to a multibyte CCSID, the conversion method is determined by the shift state of the output string:

- If the output string is in the initial shift state, any EBCDIC wide character with a hexadecimal value that is less than or equal to 0x00FF is truncated to 1 byte and placed in the output string.

- If the output string is in the initial shift state, any EBCDIC wide character with a value that is greater than 0x00FF causes a shift-out character (hexadecimal value 0x0E) to be generated in the output string. The shift state of the output string is updated to double-byte, and both bytes of the EBCDIC wide character are copied to the output string.
- If the output string is in the double-byte shift state and an EBCDIC wide character whose hexadecimal value is less than or equal to 0x00FF is encountered, a shift-in character (hexadecimal value 0x0F) is placed in the output string. The shift-in character is followed by the value of the EBCDIC wide character that is truncated to 1 byte. The shift state of the output string is changed to single-byte.
- If the output string is in the double-byte shift state and an EBCDIC wide character whose value is greater than 0x00FF is encountered, the 2 bytes of the EBCDIC wide character are copied to the output string.

For example, the EBCDIC wide character string with the hexadecimal value 00C143DA00C2 is translated to a multibyte string with the hexadecimal value C10E43DA0FC2.

**LOCALETYPE(\*LOCALEUCS2) and LOCALETYPE(\*LOCALEUTF) behavior:** If LOCALETYPE(\*LOCALEUCS2) is specified on the compilation command, wide character values are 2-byte UCS-2 values. All conversions between UCS-2 strings and single-byte or multibyte strings are conducted as if the `iconv()` function were used. CCSID 13488 is used for the UCS-2 string, and the CCSID of the LC\_CTYPE category of the current locale is used for the single-byte or multibyte string.

If LOCALETYPE(\*LOCALEUTF) is specified on the compilation command, wide character values are 4-byte UTF-32 values. All conversions between UTF-32 strings and single-byte or multibyte strings are conducted as if the `iconv()` function were used. UTF-32 is not supported by the `iconv()` function. Therefore, in conversions between a UTF-32 string and a single-byte or multibyte string, UTF-16 (CCSID 1200) is used as an intermediary data type. Transformations between UTF-32 and UTF-16 are accomplished using the `QlgTransformUCSData()` API. The `iconv()` API is used for the conversion between UTF-16 and the CCSID of the LC\_CTYPE category of the current locale.

## Wide Characters and File I/O

**Wide character write routines:** Several routines, including `fwprintf`, `vwprintf`, `vfwprintf`, `wprintf`, `fputwc`, `fputws`, `putwc`, `putwchar`, and `ungetwc` can be used to write wide characters to a file. These routines are not available when either `LOCALETYPE(*CLD)` or `SYSIFCOPT(*NOIFSIO)` is specified on the compilation command.

If `LOCALETYPE(*LOCALE)` is specified on the compilation command, the wide characters that are written are assumed to be wide character equivalents of the code points in the file CCSID. The CCSID of the file is assumed to be a single or multibyte EBCDIC CCSID.

If `LOCALETYPE(*LOCALEUCS2)` or `LOCALETYPE(*LOCALEUTF)` is specified on the compilation command, the wide characters that are being written are assumed to be Unicode characters. For `LOCALETYPE(*LOCALEUCS2)`, they are assumed to be 2-byte UCS-2 characters. For `LOCALETYPE(*LOCALEUTF)`, they are assumed to be 4-byte UTF-32 characters. If the file that is being written to is not one of the standard files, the Unicode characters are then written directly to the file as if the file had been opened for writing in binary mode. The CCSID of the file is assumed to be a Unicode CCSID that matches the locale setting. If the file that is being written to is a standard file, the Unicode input is converted to the CCSID of the job before being written to the file.

**Non-wide character write routines:** The non-wide character write routines (`fprintf`, `vfprintf`, `vprintf`, and `printfcan`) can take a wide character as input.

In all cases, the wide characters are converted to multibyte character strings in the CCSID of the LC\_CTYPE category of the current locale as if the `wctomb` function or the `wctombs` function were used. The file CCSID is assumed to match the CCSID of the LC\_CTYPE category of the current locale.

- | If LOCALETYPE(\*LOCALEUTF) is specified on the compilation command and the file that is being written to is a standard file, the output will automatically be converted from the CCSID of the LC\_CTYPE category of the current locale to the CCSID of the file (which usually matches the job CCSID).
- | **Wide character read routines:** The routines that can read wide characters from a file include fgetwc, fgetws, fwscanf, getwc, getwchar, vfwscanf, vwscanf, and wscanf. These routines are not available when either LOCALETYPE(\*CLD) or SYSIFCOPT(\*NOIFSIO) is specified on the compilation command.
- | If LOCALETYPE(\*LOCALE) is specified on the compilation command, the wide characters read from the file are assumed to be EBCDIC wide character equivalents of the code points in the file CCSID.
- | If LOCALETYPE(\*LOCALEUCS2) or LOCALETYPE(\*LOCALEUTF) is specified on the compilation command, the input wide characters and the characters in the file are assumed to be Unicode characters. For LOCALETYPE(\*LOCALEUCS2), they are assumed to be 2-byte UCS-2 characters. For LOCALETYPE(\*LOCALEUTF), they are assumed to be 4-byte UTF-32 characters. If the file that is being read is not one of the standard files, the Unicode characters are read directly from the file as if the file had been opened in binary mode. The CCSID of the file is assumed to be a Unicode CCSID that matches the locale setting. If the file that is being read is a standard file, then the job CCSID input that is read from the file is converted to the appropriate Unicode CCSID.
- | **Non-wide character read routines:** The non-wide character read routines (fscanf, scanf, vfscanf, and vscanf) can produce a wide character as output.
- | In all cases, the wide characters are converted from multibyte character strings in the CCSID of the LC\_CTYPE category of the current locale to the appropriate wide character type for the locale setting as if the mbtowc function or the mbstowcs function were used.

---

## Asynchronous Signal Model

- | The Asynchronous Signal Model (ASM) is used when the SYSIFCOPT(\*ASYNCSIGNAL) option is specified on the Create C Module (CRTCMOD) or Create Bound C Program (CRTBNDC) compilation command. The ASM is also used when the RTBND(\*LLP64) option is specified on the Create C++ Module (CRTCPPMOD) or Create Bound C++ Program (CRTBNDCPP) compilation command. It is intended for compatibility with applications ported from the UNIX operating system. For modules that use the ASM, the signal() and raise() functions are implemented using the i5/OS Signal APIs described in the Application programming interfaces topic under the Programming heading in the i5/OS Information Center.
- | i5/OS exceptions sent to an ASM module or program are converted to asynchronous signals. The exceptions are processed by an asynchronous signal handler.
- | Modules compiled to use the ASM can be intermixed with modules using the Original Signal Model (OSM) in the same processes, programs, and service programs. There are several differences between the two signal models:
  - The OSM is based on exceptions, while the ASM is based on asynchronous signals.
  - Under the OSM, the signal vector and signal mask are scoped to the activation group. Under the ASM, there is one signal vector per process and one signal mask per thread. Both types of signal vectors and signal masks are maintained at run time.
  - The same asynchronous signal vector and signal masks are operated on by all ASM modules in a thread, regardless of the activation group the signal vector and signal masks belong to. You must save and restore the state of the signal vector and signal masks to ensure that they are not changed by any ASM modules. The OSM does not use the asynchronous signal vector and signal masks.
  - Signals that are raised by OSM modules are sent as exceptions. Under the OSM, the exceptions are received and handled by the \_C\_exception\_router function, which directly calls the OSM signal handler of the user.

Asynchronous signals are not mapped to exceptions, and are not handled by signal handlers that are registered under the OSM. Under the ASM, the exceptions are received and handled by the `_C_async_exception_router` function, which maps the exception to an asynchronous signal. An ASM signal handler receives control from the i5/OS asynchronous signal component.

When an OSM module raises a signal, the generated exception percolates up the call stack until it finds an exception monitor. If the previous call is an OSM function, the `_C_exception_router` catches the exception and performs the OSM signal action. The ASM signal handler does not receive the signal.

If the previous call is an ASM function, the `_C_async_exception_router` handles the exception and maps it to an asynchronous signal. The handling of the asynchronous signal then depends on the asynchronous signal vector and mask state of the thread, as defined in the i5/OS Signal management topic.

If the previous call is an ASM function within a different program or service program, one of two actions occurs. If the OSM program that raises the signal is running in the same activation group with the ASM program, the exception is mapped to an asynchronous signal using the mapping described previously. The signal ID is preserved when the exception is mapped to a signal. So, signal handlers that are registered with the asynchronous signal model are able to receive signals generated under the original signal model. This approach can be used to integrate two programs with different signal models.

If the OSM program that raises the signal is running in a different activation group than the ASM program, any signal that is unmonitored in that activation group causes the termination of that program and activation group. The unmonitored signal is then percolated to the calling program as a CEE9901 exception. The CEE9901 exception is mapped to a SIGSEGV asynchronous signal.

- Under the ASM, the C functions `raise()` and `signal()` are integrated with the system signal functions, such as `kill()` and `sigaction()`. These two sets of APIs can be used interchangeably. This cannot be done under the OSM.
- A user-specified exception monitor established with `#pragma exception_handler` has precedence over the compiler-generated monitor, which calls `_C_async_exception_router`. In some situations, this precedence enables you to bypass the compiler-generated monitor, which invokes `_C_async_exception_router`.
- The `_GetExcData()` function is not available under the ASM to retrieve the exception ID associated with the signal. However, if an extended signal handler is established using the `sigaction()` function, it can access the exception information from the signal-specific data structure. For more information, see “`_GetExcData()` — Get Exception Data” on page 154.

---

## Unicode Support

The Unicode Standard is a standardized character code designed to encode international texts for display and storage. It uses a unique 16- or 32-bit value to represent each individual character, regardless of platform, language, or program. Using Unicode, you can develop a software product that will work with various platforms, languages, and countries or regions. Unicode also allows data to be transported through many different systems.

There are two different forms of Unicode support available from the compiler and run time. This section describes the two forms of Unicode support as well as some of the features of and considerations for using that support. To obtain additional information about Unicode, visit the Unicode Home Page at [www.unicode.org](http://www.unicode.org).

The first type of Unicode support is UCS-2 support. When the `LOCALETYPE(*LOCALEUCS2)` option is specified on the compilation command, the compiler and run time use wide characters (that is, characters of the `wchar_t` type) and wide character strings (that is, strings of the `wchar_t *` type) that represent 2-byte Unicode characters. Narrow (non-wide) characters and narrow character strings represent EBCDIC characters, just as they do when the UCS-2 support is not enabled. The Unicode characters represent codepoints in CCSID 13488.

| The second type of Unicode support is UTF-8 or UTF-32 support (also known as UTF support). When the  
| LOCALETYPE(\*LOCALEUTF) option is specified on the compilation command, the compiler and run  
| time use wide characters and wide character strings that represent 4-byte Unicode characters. Each 4-byte  
| character represents a single UTF-32 character. Narrow characters and narrow character strings represent  
| UTF-8 characters. Each UTF-8 character is from 1 to 4 bytes in size. Most normal characters are a single  
| byte in size, and, in fact, all 7-bit ASCII characters map directly to UTF-8 and are 1 byte in size. The  
| UTF-8 characters represent codepoints in CCSID 1208.

When the UTF support is enabled, not only do the wide characters become UTF-32 Unicode, but the narrow characters become UTF-8 Unicode as well. As an example, consider the following HelloWorld program.

```
#include <stdio.h>

int main() {
    printf("Hello World\n");
    return 0;
}
```

When this program is compiled with UTF support, the character string is stored within the program as UTF-8 characters and not EBCDIC characters. The printf() function knows this and is able to parse the UTF-8 characters and generate the output as expected. However, if this program called some other user-supplied routine that did not know how to handle UTF-8 characters, the other routine might yield incorrect results or behavior.

## Reasons to Use Unicode Support

| You might want to use Unicode support for your application in two situations. The first situation is if  
| your application is an international application and requires support for several different languages. The  
| Unicode character set provides an easy way to allow a single application to handle any language or  
| character set. The application can perform all input, processing, and output using Unicode characters.  
| Another situation for using Unicode support is for porting a 7-bit ASCII application. Because the UTF-8  
| character set is a superset of 7-bit ASCII, an ASCII application can be ported more easily to a UTF-8  
| environment than to an EBCDIC environment.

## Pseudo-CCSID Neutrality

When a program is compiled with UTF support, the run time allows more than just UTF-8 characters, and it essentially becomes CCSID neutral. The run time handles whatever CCSID is contained within the current locale. By default, when a program is compiled with UTF support, the locale that is loaded is a UTF-8 (CCSID 1208) locale. This allows the run time to handle CCSID 1208. If the setlocale() routine is called to set the locale to an EBCDIC locale (for example, a CCSID 37 locale), the run time handles CCSID 37. This, along with the #pragma convert support within the compiler, can be used to provide international application support. Here is an example:

```
| #include <stdio.h>
| #include <locale.h>
|
| int main() {
|     /* This string is in CCSID 1208 */
|     printf("Hello World\n");
|
|     /* Change locale to a CCSID 37 locale */
|     setlocale(LC_ALL, "/QSYS.LIB/EN_US.LOCALE");
|     #pragma convert(37)
|
|     /* This string is in CCSID 37 */
|     printf("Hello World\n");
|
|     return 0;
| }
```

## Unicode from Other ILE Languages

The Unicode routines are easily accessible in the C and C++ languages if you include the appropriate header files and use the appropriate LOCALETYPE option on the C or C++ compilation command. The Unicode routines are accessible from other ILE languages, such as RPG, COBOL, and CL, although no header files are provided for these languages.

The following table shows the routines added for UCS-2 support. The support routines have a prefix of `_UCS2_` or `_C_UCS2_` added to the standard routine name. The Unicode routine has the same parameters as the standard (non-Unicode) routine.

<code>_C_UCS2_btowc</code>	<code>_C_UCS2_iswspace</code>	<code>_C_UCS2_vfprintf</code>	<code>_C_UCS2_wcstod128</code>
<code>_C_UCS2_fgetwc</code>	<code>_C_UCS2_iswupper</code>	<code>_C_UCS2_vfscanf</code>	<code>_C_UCS2_wctob</code>
<code>_C_UCS2_fgetws</code>	<code>_C_UCS2_iswxdigit</code>	<code>_C_UCS2_vfwprintf</code>	<code>_C_UCS2_wprintf</code>
<code>_C_UCS2_fprintf</code>	<code>_C_UCS2_mblen</code>	<code>_C_UCS2_vfwscanf</code>	<code>_C_UCS2_wscanf</code>
<code>_C_UCS2_fputwc</code>	<code>_C_UCS2_mbrlen</code>	<code>_C_UCS2_vprintf</code>	<code>_UCS2_mbstowcs</code>
<code>_C_UCS2_fputws</code>	<code>_C_UCS2_mbrtowc</code>	<code>_C_UCS2_vscanf</code>	<code>_UCS2_mbtowc</code>
<code>_C_UCS2_fscanf</code>	<code>_C_UCS2_mbsinit</code>	<code>_C_UCS2_vsnprintf</code>	<code>_UCS2_setlocale</code>
<code>_C_UCS2_fwprintf</code>	<code>_C_UCS2_mbsrtowcs</code>	<code>_C_UCS2_vsprintf</code>	<code>_UCS2_wcrtomb</code>
<code>_C_UCS2_fwscanf</code>	<code>_C_UCS2_printf</code>	<code>_C_UCS2_vsscanf</code>	<code>_UCS2_wcstod</code>
<code>_C_UCS2_getwc</code>	<code>_C_UCS2_putwc</code>	<code>_C_UCS2_vswprintf</code>	<code>_UCS2_wcstol</code>
<code>_C_UCS2_getwchar</code>	<code>_C_UCS2_putwchar</code>	<code>_C_UCS2_vswscanf</code>	<code>_UCS2_wcstoll</code>
<code>_C_UCS2_iswalnum</code>	<code>_C_UCS2_scanf</code>	<code>_C_UCS2_vwprintf</code>	<code>_UCS2_wcstombs</code>
<code>_C_UCS2_iswalpha</code>	<code>_C_UCS2_snprintf</code>	<code>_C_UCS2_vwscanf</code>	<code>_UCS2_wcstoul</code>
<code>_C_UCS2_iswcntrl</code>	<code>_C_UCS2_sprintf</code>	<code>_C_UCS2_wcsftime</code>	<code>_UCS2_wcstoull</code>
<code>_C_UCS2_iswctype</code>	<code>_C_UCS2_sscanf</code>	<code>_C_UCS2_wcsicmp</code>	<code>_UCS2_wcswidth</code>
<code>_C_UCS2_iswdigit</code>	<code>_C_UCS2_swprintf</code>	<code>_C_UCS2_wcslocaleconv</code>	<code>_UCS2_wctomb</code>
<code>_C_UCS2_iswgraph</code>	<code>_C_UCS2_swscanf</code>	<code>_C_UCS2_wcsnicmp</code>	<code>_UCS2_wcwidth</code>
<code>_C_UCS2_iswlower</code>	<code>_C_UCS2_towlower</code>	<code>_C_UCS2_wcsrtombs</code>	
<code>_C_UCS2_iswprint</code>	<code>_C_UCS2_towupper</code>	<code>_C_UCS2_wcstod32</code>	
<code>_C_UCS2_iswpunct</code>	<code>_C_UCS2_ungetwc</code>	<code>_C_UCS2_wcstod64</code>	

When you use the LOCALETYPE(\*LOCALEUCS2) option with either the C or C++ compiler, the default UCS-2 locale is loaded when the program starts. When you use any of the Unicode routines in the preceding table from a different language, a call to `_UCS2_setlocale(LC_ALL, "")` should be added when the application starts to ensure that the default UCS2 locale is loaded.

The following table shows the routines added for CCSID neutral and UTF-8 support. The routines have a prefix of `_C_NEU_DM_` (for data management I/O functions), `_C_NEU_IFS_` or `_C_UTF_IFS_` (for IFS I/O functions), or `_C_NEU_` or `_C_UTF_` added to the standard routine name. The Unicode routine has the same parameters as the standard (non-Unicode) routine.

Routines that operate on wide characters have UTF in the prefix. Routines that do not operate on wide characters have NEU in the prefix.

	_C_NEU_asctime	_C_NEU_IFS_fgetpos	_C_NEU_localtime_r
	_C_NEU_asctime_r	_C_NEU_IFS_fgetpos64	_C_NEU_localtime64
	_C_NEU_atof	_C_NEU_IFS_fgets	_C_NEU_localtime64_r
	_C_NEU_atoi	_C_NEU_IFS_fopen	_C_NEU_ltoa
	_C_NEU_catopen	_C_NEU_IFS_fopen64	_C_NEU_memicmp
	_C_NEU_ctime	_C_NEU_IFS_fprintf	_C_NEU_mktime
	_C_NEU_ctime_r	_C_NEU_IFS_fputc	_C_NEU_mktime64
	_C_NEU_ctime64	_C_NEU_IFS_fputs	_C_NEU_nl_langinfo
	_C_NEU_ctime64_r	_C_NEU_IFS_fread	_C_NEU_snprintf
	_C_NEU_DM_clearerr	_C_NEU_IFS_freopen	_C_NEU_sprintf
	_C_NEU_DM_feof	_C_NEU_IFS_freopen64	_C_NEU_sscanf
	_C_NEU_DM_ferror	_C_NEU_IFS_fscanf	_C_NEU_strerror
	_C_NEU_DM_fflush	_C_NEU_IFS_fseek	_C_NEU_strerror
	_C_NEU_DM_fgetc	_C_NEU_IFS_fseeko	_C_NEU_strerror
	_C_NEU_DM_fgetpos	_C_NEU_IFS_fseeko64	_C_NEU_strerror
	_C_NEU_DM_fgets	_C_NEU_IFS_fsetpos	_C_NEU_strerror
	_C_NEU_DM_fopen	_C_NEU_IFS_fsetpos64	_C_NEU_strerror
	_C_NEU_DM_fprintf	_C_NEU_IFS_ftell	_C_NEU_strerror
	_C_NEU_DM_fputc	_C_NEU_IFS_ftello	_C_NEU_strerror
	_C_NEU_DM_fputs	_C_NEU_IFS_ftello64	_C_NEU_strerror
	_C_NEU_DM_fread	_C_NEU_IFS_fwrite	_C_NEU_strerror
	_C_NEU_DM_freopen	_C_NEU_IFS_getc	_C_NEU_strerror
	_C_NEU_DM_fscanf	_C_NEU_IFS_getchar	_C_NEU_strerror
	_C_NEU_DM_fseek	_C_NEU_IFS_gets	_C_NEU_strerror
	_C_NEU_DM_fsetpos	_C_NEU_IFS_perror	_C_NEU_strerror
	_C_NEU_DM_ftell	_C_NEU_IFS_printf	_C_NEU_strerror
	_C_NEU_DM_fwrite	_C_NEU_IFS_putc	_C_NEU_strerror
	_C_NEU_DM_getc	_C_NEU_IFS_putchar	_C_NEU_strerror
	_C_NEU_DM_getchar	_C_NEU_IFS_puts	_C_NEU_strerror
	_C_NEU_DM_gets	_C_NEU_IFS_remove	_C_NEU_strerror
	_C_NEU_DM_perror	_C_NEU_IFS_rename_keep	_C_NEU_strerror
	_C_NEU_DM_printf	_C_NEU_IFS_rename_unlink	_C_NEU_strerror
	_C_NEU_DM_putc	_C_NEU_IFS_rewind	_C_NEU_strerror
	_C_NEU_DM_putchar	_C_NEU_IFS_scanf	_C_NEU_strerror
	_C_NEU_DM_puts	_C_NEU_IFS_setbuf	_C_NEU_strerror
	_C_NEU_DM_remove	_C_NEU_IFS_setvbuf	_C_NEU_strerror
	_C_NEU_DM_rename	_C_NEU_IFS_tmpfile	_C_NEU_strerror
	_C_NEU_DM_rewind	_C_NEU_IFS_tmpfile64	_C_NEU_strerror
	_C_NEU_DM_ropen	_C_NEU_IFS_tmpnam	_C_NEU_strerror
	_C_NEU_DM_scanf	_C_NEU_IFS_ungetc	_C_NEU_strerror
	_C_NEU_DM_setbuf	_C_NEU_IFS_vfprintf	_C_NEU_strerror
	_C_NEU_DM_setvbuf	_C_NEU_IFS_vfscanf	_C_NEU_strerror
	_C_NEU_DM_tmpfile	_C_NEU_IFS_vprintf	_C_NEU_strerror
	_C_NEU_DM_tmpnam	_C_NEU_IFS_vscanf	_C_NEU_strerror
	_C_NEU_DM_ungetc	_C_NEU_isalnum	_C_UTF_btowc
	_C_NEU_DM_vfprintf	_C_NEU_isalpha	_C_UTF_IFS_fgetwc
	_C_NEU_DM_vfscanf	_C_NEU_iscntrl	_C_UTF_IFS_fgetws
	_C_NEU_DM_vprintf	_C_NEU_isdigit	_C_UTF_IFS_fputc
	_C_NEU_DM_vscanf	_C_NEU_isgraph	_C_UTF_IFS_fputws
	_C_NEU_gcvt	_C_NEU_islower	_C_UTF_IFS_fwprintf
	_C_NEU_gmtime	_C_NEU_isprint	_C_UTF_IFS_fwscanf
	_C_NEU_gmtime_r	_C_NEU_ispunct	_C_UTF_IFS_getwc
	_C_NEU_IFS_clearerr	_C_NEU_isspace	_C_UTF_IFS_getwchar
	_C_NEU_IFS_fdopen	_C_NEU_isupper	_C_UTF_IFS_putwc
	_C_NEU_IFS_feof	_C_NEU_isxdigit	_C_UTF_IFS_putwchar
	_C_NEU_IFS_ferror	_C_NEU_itoa	_C_UTF_IFS_ungetwc
	_C_NEU_IFS_fflush	_C_NEU_localeconv	_C_UTF_IFS_vfwprintf
	_C_NEU_IFS_fgetc	_C_NEU_localtime	_C_UTF_IFS_vfwscanf

<code>_C_UTF_IFS_vwprintf</code>	<code>_C_UTF_mbsinit</code>	<code>_C_UTF_wcsncmp</code>
<code>_C_UTF_IFS_wscanf</code>	<code>_C_UTF_mbsrtowcs</code>	<code>_C_UTF_wcsncpy</code>
<code>_C_UTF_IFS_wfopen</code>	<code>_C_UTF_mbstowcs</code>	<code>_C_UTF_wcsnicmp</code>
<code>_C_UTF_IFS_wfopen64</code>	<code>_C_UTF_mbtowc</code>	<code>_C_UTF_WCS_nl_langinfo</code>
<code>_C_UTF_IFS_wprintf</code>	<code>_C_UTF_regcomp</code>	<code>_C_UTF_wcsprbk</code>
<code>_C_UTF_IFS_wscanf</code>	<code>_C_UTF_regerror</code>	<code>_C_UTF_wcsptime</code>
<code>_C_UTF_isalnum</code>	<code>_C_UTF_regexec</code>	<code>_C_UTF_wcsrchr</code>
<code>_C_UTF_isalpha</code>	<code>_C_UTF_setlocale</code>	<code>_C_UTF_wcsrtombs</code>
<code>_C_UTF_iscntrl</code>	<code>_C_UTF_strcoll</code>	<code>_C_UTF_wcsspn</code>
<code>_C_UTF_isdigit</code>	<code>_C_UTF_strxfrm</code>	<code>_C_UTF_wcsstr</code>
<code>_C_UTF_isgraph</code>	<code>_C_UTF_swprintf</code>	<code>_C_UTF_wcstod</code>
<code>_C_UTF_islower</code>	<code>_C_UTF_wscanf</code>	<code>_C_UTF_wcstod32</code>
<code>_C_UTF_isprint</code>	<code>_C_UTF_toascii</code>	<code>_C_UTF_wcstod64</code>
<code>_C_UTF_ispunct</code>	<code>_C_UTF_tolower</code>	<code>_C_UTF_wcstod128</code>
<code>_C_UTF_isspace</code>	<code>_C_UTF_toupper</code>	<code>_C_UTF_wcstok</code>
<code>_C_UTF_isupper</code>	<code>_C_UTF_towctrans</code>	<code>_C_UTF_wcstol</code>
<code>_C_UTF_iswalnum</code>	<code>_C_UTF_towlower</code>	<code>_C_UTF_wcstoll</code>
<code>_C_UTF_iswalph</code>	<code>_C_UTF_towupper</code>	<code>_C_UTF_wcstombs</code>
<code>_C_UTF_iswcntrl</code>	<code>_C_UTF_vswprintf</code>	<code>_C_UTF_wcstoul</code>
<code>_C_UTF_iswctype</code>	<code>_C_UTF_wscanf</code>	<code>_C_UTF_wcstoul1</code>
<code>_C_UTF_iswdigit</code>	<code>_C_UTF_wcrtomb</code>	<code>_C_UTF_wcswcs</code>
<code>_C_UTF_iswgraph</code>	<code>_C_UTF_wcscat</code>	<code>_C_UTF_wcswidth</code>
<code>_C_UTF_iswlower</code>	<code>_C_UTF_wcschr</code>	<code>_C_UTF_wcsxfrm</code>
<code>_C_UTF_iswprint</code>	<code>_C_UTF_wcscmp</code>	<code>_C_UTF_wctob</code>
<code>_C_UTF_iswpunct</code>	<code>_C_UTF_wcscoll</code>	<code>_C_UTF_wctomb</code>
<code>_C_UTF_isspace</code>	<code>_C_UTF_wcsncpy</code>	<code>_C_UTF_wcwidth</code>
<code>_C_UTF_iswupper</code>	<code>_C_UTF_wcsdspn</code>	<code>_C_UTF_wmemchr</code>
<code>_C_UTF_iswxdigit</code>	<code>_C_UTF_wcsfmon</code>	<code>_C_UTF_wmemcmp</code>
<code>_C_UTF_isxdigit</code>	<code>_C_UTF_wcsftime</code>	<code>_C_UTF_wmemcpy</code>
<code>_C_UTF_mblen</code>	<code>_C_UTF_wcsicmp</code>	<code>_C_UTF_wmemmove</code>
<code>_C_UTF_mbrlen</code>	<code>_C_UTF_wcslen</code>	<code>_C_UTF_wmemset</code>
<code>_C_UTF_mbrtowc</code>	<code>_C_UTF_wcslocaleconv</code>	
<code>_C_UTF_mbrtowc</code>	<code>_C_UTF_wcsncat</code>	

| When you use the LOCALETYPE(\*LOCALEUTF) option with either the C or C++ compiler, the default UTF locale is loaded at program startup time. If you use any of the Unicode routines in the preceding table from a different language, a call to `_C_UTF_setlocale(LC_ALL, "")` should be added when the application starts to ensure that the default UTF locale is loaded.

## Standard Files

When using the UTF support, the default standard input and output files `stdin`, `stdout`, and `stderr` have some special processing done for them by the run time. Since a program using UTF support contains data in UTF-8 and the standard files interact with the screen and spool files, there is a potential mismatch in data. The screen and spool file routines are provided by the operating system and thus expect EBCDIC. For `stdout` and `stderr`, the run time will automatically convert UTF-8 data to EBCDIC. For `stdin`, the run time will automatically convert the incoming EBCDIC to UTF-8 data.

## Considerations

| Because the default environment for i5/OS is primarily an EBCDIC environment, you must be aware of the situations described in this topic when you use UTF support in an application.

If a program or service program has some modules compiled with the UTF support and some modules compiled without the UTF support, care must be taken to ensure that unexpected mismatches do not occur. The wide characters and wide character strings are two bytes in size for a non-UTF module and four bytes in size for a UTF module, so sharing wide characters between the modules may not work

correctly. The narrow (non-wide) characters and character strings are in job CCSID for a non-UTF module and in CCSID 1208 for a UTF module, so sharing narrow characters between the modules may not work correctly either.

Whenever a `setlocale()` is performed to set the locale to a different CCSID, the standard output files should be flushed to avoid buffering problems with character data containing multiple CCSIDs. Since `stdout` is line buffered by default, if each output line ends in a newline character, the problem will not occur. However, if this is not done, the output may not be shown as intended. The following example illustrates the problem.

```
#include <stdio>
#include <locale.h>

int main() {
    /* This string is in CCSID 1208 */
    printf("Hello World");

    /* Change locale to a CCSID 37 locale */
    setlocale(LC_ALL, "/QSYS.LIB/EN_US.LOCALE");
    #pragma convert(37)

    /* This string is in CCSID 37 */
    printf("Hello World\n");

    return 0;
}
```

In this case, the first `printf()` causes the CCSID 1208 string “Hello World” to be copied to the `stdout` buffer. Before the `setlocale()` is done, `stdout` should be flushed to copy that string to the screen. The second `printf()` causes the CCSID 37 string “Hello World\n” to be copied to the `stdout` buffer. Because of the trailing newline character, the buffer is flushed at that point and the whole buffer is copied to the screen. Because the CCSID of the current locale is 37 and the screen can handle CCSID 37 without problems, the whole buffer is copied without conversion. The CCSID 1208 characters are displayed as unreadable characters. If a flush had been done, the CCSID 1208 characters would have been converted to CCSID 37 and would have been displayed correctly.

Nearly all of the runtime routines have been modified to support UTF, but there are a handful of them that have not. Routines and structures that deal with exception handling, such as the `_GetExcData()` function, the `_EXCP_MSGID` variable, and the exception handler structure `_INTRPT_Hndlr_Parms_T` are provided by the operating system, not the run time. They are strictly EBCDIC. The `getenv()` and `putenv()` functions handle only EBCDIC. The `QXXCHGDA()` and `QXXRTVDA()` functions handle only EBCDIC. The `argv` and `envp` parameters are also EBCDIC only.

Some of the record I/O routines (that is, functions beginning with `_R`) do not completely support UTF. The routines that do not support UTF are `_Rformat()`, `_Rcommit()`, `_Racquire()`, `_Rrelease()`, `_Rpgmdev()`, `_Rindara()`, and `_Rdevatr()`. They are available when compiling with the UTF option, but they accept and generate only EBCDIC. In addition, any character data within the structures returned by the `_R` functions will be in EBCDIC rather than UTF.

| Other operating system routines have not been modified to support UTF. For example, the integrated file  
| system routines, such as `open()`, still accept the job CCSID. Other operating system APIs still accept the  
| job CCSID. For UTF applications, the characters and character strings provided to these routines need to  
| be converted to the job CCSID using `QTQCVRT`, `iconv()`, `#pragma convert`, or some other method.

## Default File CCSID

When the `fopen()` function is used to open files, the default CCSID of the file is different depending on whether or not UTF support is used. If UTF support is not used (that is, if `LOCALETYPE(*CLD)`, `LOCALETYPE(*LOCALE)`, or `LOCALETYPE(*LOCALEUCS2)` are specified on the compilation

command), the file CCSID defaults to the current job CCSID. Usually this works well because the job CCSID is set correctly and the current locale is set to match the job CCSID.

| With UTF support, the job CCSID cannot be set to UTF-8 because of system limitations. When  
| LOCALETYPE(\*LOCALEUTF) is specified, the file CCSID defaults to the CCSID of the current locale. If  
| the default locale is being used, the CCSID defaults to UTF-8 (CCSID 1208). If this default is not desired,  
| the *ccsid* or *o\_ccsid* keyword can be specified in the second parameter of the `fopen()` call. However,  
| database files are an exception, because DB2® for i5/OS does not completely support UTF-8. When  
| SYSIFCOPT(\*NOIFSIO) is specified, and the CCSID of the current locale is 1208, the CCSID of the file  
| defaults to CCSID 65535 (no conversion) rather than CCSID 1208. This allows CCSID 1208 to be used  
| with database files. For more information about file CCSIDs, see “`fopen()` — Open Files” on page 109.

## Newline Character

When the UTF support is not used, the hexadecimal value generated by the compiler for the character `\n` and used by the run time has two different values. The hexadecimal value `0x15` is used if `SYSIFCOPT(*NOIFSIO)` is specified on the compilation command. The hexadecimal value `0x25` is used if `SYSIFCOPT(*IFSIO)` or `SYSIFCOPT(*IFS64IO)` is specified on the compilation command. When the UTF support is used, the newline character in UTF-8 will be hexadecimal `0x0a` regardless of what `SYSIFCOPT` value is used.

## Conversion Errors

Some runtime routines perform a CCSID conversion from UTF-8 to an EBCDIC CCSID when required to interface with an operating system function that does not support UTF-8. When a conversion error occurs in these cases, a C2M1217 message is generated to the job log with the conversion information.

---

## Appendix A. Library Functions and Extensions

This chapter summarizes all the standard C library functions and the ILE C library extensions.

---

### Standard C Library Functions Table, By Name

This table briefly describes the C library functions, listed in alphabetical order. This table provides the include file name and the function prototype for each function.

Table 32. Standard C Library Functions

Function	System Include File	Function Prototype	Description
abort	stdlib.h	void abort(void);	Stops a program abnormally.
abs	stdlib.h	int abs(int <i>n</i> );	Calculates the absolute value of an integer argument <i>n</i> .
acos	math.h	double acos(double <i>x</i> );	Calculates the arc cosine of <i>x</i> .
asctime	time.h	char *asctime(const struct tm * <i>time</i> );	Converts the <i>time</i> that is stored as a structure to a character string.
asctime_r	time.h	char *asctime_r (const struct tm * <i>tm</i> , char * <i>buf</i> );	Converts <i>tm</i> that is stored as a structure to a character string. (Restartable version of asctime.)
asin	math.h	double asin(double <i>x</i> );	Calculates the arc sine of <i>x</i> .
assert	assert.h	void assert(int <i>expression</i> );	Prints a diagnostic message and ends the program if the expression is false.
atan	math.h	double atan(double <i>x</i> );	Calculates the arc tangent of <i>x</i> .
atan2	math.h	double atan2(double <i>y</i> , double <i>x</i> );	Calculates the arc tangent of <i>y/x</i> .
atexit	stdlib.h	int atexit(void (* <i>func</i> )(void));	Registers a function to be called at normal termination.
atof	stdlib.h	double atof(const char * <i>string</i> );	Converts <i>string</i> to a double-precision floating-point value.
atoi	stdlib.h	int atoi(const char * <i>string</i> );	Converts <i>string</i> to an integer.
atol	stdlib.h	long int atol(const char * <i>string</i> );	Converts <i>string</i> to a long integer.
bsearch	stdlib.h	void *bsearch(const void * <i>key</i> , const void * <i>base</i> , size_t <i>num</i> , size_t <i>size</i> , int (* <i>compare</i> ) (const void * <i>element1</i> , const void * <i>element2</i> ));	Performs a binary search on an array of <i>num</i> elements, each of <i>size</i> bytes. The array must be sorted in ascending order by the function pointed to by <i>compare</i> .
btowc	stdio.h wchar.h	wint_t btowc(int <i>c</i> );	Determines whether <i>c</i> constitutes a valid multibyte character in the initial shift state.
calloc	stdlib.h	void *calloc(size_t <i>num</i> , size_t <i>size</i> );	Reserves storage space for an array of <i>num</i> elements, each of size <i>size</i> , and initializes the values of all elements to 0.
catclose <sup>6</sup>	nl_types.h	int catclose (nl_catd <i>catd</i> );	Closes a previously opened message catalog.

Table 32. Standard C Library Functions (continued)

Function	System Include File	Function Prototype	Description
catgets <sup>6</sup>	nl_types.h	char *catgets(nl_catd catd, int set_id, int msg_id, const char *s);	Retrieves a message from an open message catalog.
catopen <sup>6</sup>	nl_types.h	nl_catd catopen (const char *name, int oflag);	Opens a message catalog, which must be done before a message can be retrieved.
ceil	math.h	double ceil(double x);	Calculates the double value representing the smallest integer that is greater than or equal to $x$ .
clearerr	stdio.h	void clearerr(FILE *stream);	Resets the error indicators and the end-of-file indicator for <i>stream</i> .
clock	time.h	clock_t clock(void);	Returns the processor time that has elapsed since the job was started.
cos	math.h	double cos(double x);	Calculates the cosine of $x$ .
cosh	math.h	double cosh(double x);	Calculates the hyperbolic cosine of $x$ .
ctime	time.h	char *ctime(const time_t *time);	Converts <i>time</i> to a character string.
ctime64	time.h	char *ctime64(const time64_t *time);	Converts <i>time</i> to a character string.
ctime_r	time.h	char *ctime_r(const time_t *time, char *buf);	Converts <i>time</i> to a character string. (Restartable version of ctime.)
ctime64_r	time.h	char *ctime64_r(const time64_t *time, char *buf);	Converts <i>time</i> to a character string. (Restartable version of ctime64.)
difftime	time.h	double difftime(time_t time2, time_t time1);	Computes the difference between <i>time2</i> and <i>time1</i> .
difftime64	time.h	double difftime64(time64_t time2, time64_t time1);	Computes the difference between <i>time2</i> and <i>time1</i> .
div	stdlib.h	div_t div(int numerator, int denominator);	Calculates the quotient and remainder of the division of <i>numerator</i> by <i>denominator</i> .
erf	math.h	double erf(double x);	Calculates the error function of $x$ .
erfc	math.h	double erfc(double x);	Calculates the error function for large values of $x$ .
exit	stdlib.h	void exit(int status);	Ends a program normally.
exp	math.h	double exp(double x);	Calculates the exponential function of a floating-point argument $x$ .
fabs	math.h	double fabs(double x);	Calculates the absolute value of a floating-point argument $x$ .
fclose	stdio.h	int fclose(FILE *stream);	Closes the specified <i>stream</i> .
fdopen <sup>5</sup>	stdio.h	FILE *fdopen(int handle, const char *type);	Associates an input or output stream with the file identified by <i>handle</i> .
feof	stdio.h	int feof(FILE *stream);	Tests whether the end-of-file flag is set for a given <i>stream</i> .
ferror	stdio.h	int ferror(FILE *stream);	Tests for an error indicator in reading from or writing to <i>stream</i> .
fflush <sup>1</sup>	stdio.h	int fflush(FILE *stream);	Writes the contents of the buffer associated with the output <i>stream</i> .

Table 32. Standard C Library Functions (continued)

Function	System Include File	Function Prototype	Description
fgetc <sup>1</sup>	stdio.h	int fgetc(FILE *stream);	Reads a single unsigned character from the input <i>stream</i> .
fgetpos <sup>1</sup>	stdio.h	int fgetpos(FILE *stream, fpos_t *pos);	Stores the current position of the file pointer associated with <i>stream</i> into the object pointed to by <i>pos</i> .
fgets <sup>1</sup>	stdio.h	char *fgets(char *string, int n, FILE *stream);	Reads a string from the input <i>stream</i> .
fgetwc <sup>6</sup>	stdio.h wchar.h	wint_t fgetwc(FILE *stream);	Reads the next multibyte character from the input stream pointed to by <i>stream</i> .
fgetws <sup>6</sup>	stdio.h wchar.h	wchar_t *fgetws(wchar_t *wcs, int n, FILE *stream);	Reads wide characters from the stream into the array pointed to by <i>wcs</i> .
fileno <sup>5</sup>	stdio.h	int fileno(FILE *stream);	Determines the file handle currently associated with <i>stream</i> .
floor	math.h	double floor(double x);	Calculates the floating-point value representing the largest integer less than or equal to <i>x</i> .
fmod	math.h	double fmod(double x, double y);	Calculates the floating-point remainder of <i>x/y</i> .
fopen	stdio.h	FILE *fopen(const char *filename, const char *mode);	Opens the specified file.
fprintf	stdio.h	int fprintf(FILE *stream, const char *format-string, arg-list);	Formats and prints characters and values to the output <i>stream</i> .
fputc <sup>1</sup>	stdio.h	int fputc(int c, FILE *stream);	Prints a character to the output <i>stream</i> .
fputs <sup>1</sup>	stdio.h	int fputs(const char *string, FILE *stream);	Copies a string to the output <i>stream</i> .
fputwc <sup>6</sup>	stdio.h wchar.h	wint_t fputwc(wchar_t wc, FILE *stream);	Converts the wide character <i>wc</i> to a multibyte character and writes it to the output stream pointed to by <i>stream</i> at the current position.
fputws <sup>6</sup>	stdio.h wchar.h	int fputws(const wchar_t *wcs, FILE *stream);	Converts the wide-character string <i>wcs</i> to a multibyte-character string and writes it to <i>stream</i> as a multibyte character string.
fread	stdio.h	size_t fread(void *buffer, size_t size, size_t count, FILE *stream);	Reads up to <i>count</i> items of <i>size</i> length from the input <i>stream</i> , and stores them in <i>buffer</i> .
free	stdlib.h	void free(void *ptr);	Frees a block of storage.
freopen	stdio.h	FILE *freopen(const char *filename, const char *mode, FILE *stream);	Closes <i>stream</i> , and reassigns it to the file specified.
frexp	math.h	double frexp(double x, int *exp_ptr);	Separates a floating-point number into its mantissa and exponent.
fscanf	stdio.h	int fscanf(FILE *stream, const char *format-string, arg-list);	Reads data from <i>stream</i> into locations given by <i>arg-list</i> .
fseek <sup>1</sup>	stdio.h	int fseek(FILE *stream, long int offset, int origin);	Changes the current file position associated with <i>stream</i> to a new location.

Table 32. Standard C Library Functions (continued)

Function	System Include File	Function Prototype	Description
fsetpos <sup>1</sup>	stdio.h	int fsetpos(FILE *stream, const fpos_t *pos);	Moves the current file position to a new location determined by <i>pos</i> .
ftell <sup>1</sup>	stdio.h	long int ftell(FILE *stream);	Gets the current position of the file pointer.
fwide <sup>6</sup>	stdio.h wchar.h	int fwide(FILE *stream, int mode);	Determines the orientation of the stream pointed to by <i>stream</i> .
fwprintf <sup>6</sup>	stdio.h wchar.h	int fwprintf(FILE *stream, const wchar_t *format, arg-list);	Writes output to the stream pointed to by <i>stream</i> .
fwrite	stdio.h	size_t fwrite(const void *buffer, size_t size, size_t count, FILE *stream);	Writes up to <i>count</i> items of <i>size</i> length from <i>buffer</i> to <i>stream</i> .
fwscanf <sup>6</sup>	stdio.h wchar.h	int fwscanf(FILE *stream, const wchar_t *format, arg-list)	Reads input from the stream pointed to by <i>stream</i> .
gamma	math.h	double gamma(double x);	Computes the Gamma Function
getc <sup>1</sup>	stdio.h	int getc(FILE *stream);	Reads a single character from the input <i>stream</i> .
getchar <sup>1</sup>	stdio.h	int getchar(void);	Reads a single character from <i>stdin</i> .
getenv	stdlib.h	char *getenv(const char *varname);	Searches environment variables for <i>varname</i> .
gets	stdio.h	char *gets(char *buffer);	Reads a string from <i>stdin</i> , and stores it in <i>buffer</i> .
getwc <sup>6</sup>	stdio.h wchar.h	wint_t getwc(FILE *stream);	Reads the next multibyte character from <i>stream</i> , converts it to a wide character and advances the associated file position indicator for <i>stream</i> .
getwchar <sup>6</sup>	wchar.h	wint_t getwchar(void);	Reads the next multibyte character from <i>stdin</i> , converts it to a wide character, and advances the associated file position indicator for <i>stdin</i> .
gmtime	time.h	struct tm *gmtime(const time_t *time);	Converts a <i>time</i> value to a structure of type <i>tm</i> .
gmtime64	time.h	struct tm *gmtime64(const time64_t *time);	Converts a <i>time</i> value to a structure of type <i>tm</i> .
gmtime_r	time.h	struct tm *gmtime_r (const time_t *time, struct tm *result);	Converts a <i>time</i> value to a structure of type <i>tm</i> . (Restartable version of <i>gmtime</i> .)
gmtime64_r	time.h	struct tm *gmtime64_r (const time64_t *time, struct tm *result);	Converts a <i>time</i> value to a structure of type <i>tm</i> . (Restartable version of <i>gmtime64</i> .)
hypot	math.h	double hypot(double side1, double side2);	Calculates the hypotenuse of a right-angled triangle with sides of length <i>side1</i> and <i>side2</i> .
isalnum	ctype.h	int isalnum(int c);	Tests if <i>c</i> is alphanumeric.
isalpha	ctype.h	int isalpha(int c);	Tests if <i>c</i> is alphabetic.
isascii	ctype.h	int isascii(int c);	Tests if <i>c</i> is within the 7-bit US-ASCII range.
isctrl	ctype.h	int isctrl(int c);	Tests if <i>c</i> is a control character.

Table 32. Standard C Library Functions (continued)

Function	System Include File	Function Prototype	Description
isdigit	ctype.h	int isdigit(int <i>c</i> );	Tests if <i>c</i> is a decimal digit.
isgraph	ctype.h	int isgraph(int <i>c</i> );	Tests if <i>c</i> is a printable character excluding the space.
islower	ctype.h	int islower(int <i>c</i> );	Tests if <i>c</i> is a lowercase letter.
isprint	ctype.h	int isprint(int <i>c</i> );	Tests if <i>c</i> is a printable character including the space.
ispunct	ctype.h	int ispunct(int <i>c</i> );	Tests if <i>c</i> is a punctuation character.
isspace	ctype.h	int isspace(int <i>c</i> );	Tests if <i>c</i> is a whitespace character.
isupper	ctype.h	int isupper(int <i>c</i> );	Tests if <i>c</i> is an uppercase letter.
iswalnum <sup>4</sup>	wctype.h	int iswalnum (wint_t <i>wc</i> );	Checks for any alphanumeric wide character.
iswalpha <sup>4</sup>	wctype.h	int iswalpha (wint_t <i>wc</i> );	Checks for any alphabetic wide character.
iswcntrl <sup>4</sup>	wctype.h	int iswcntrl (wint_t <i>wc</i> );	Tests for any control wide character.
iswctype <sup>4</sup>	wctype.h	int iswctype(wint_t <i>wc</i> , wctype_t <i>wc_prop</i> );	Determines whether or not the wide character <i>wc</i> has the property <i>wc_prop</i> .
iswdigit <sup>4</sup>	wctype.h	int iswdigit (wint_t <i>wc</i> );	Checks for any decimal-digit wide character.
iswgraph <sup>4</sup>	wctype.h	int iswgraph (wint_t <i>wc</i> );	Checks for any printing wide character except for the wide-character space.
iswlower <sup>4</sup>	wctype.h	int iswlower (wint_t <i>wc</i> );	Checks for any lowercase wide character.
iswprint <sup>4</sup>	wctype.h	int iswprint (wint_t <i>wc</i> );	Checks for any printing wide character.
iswpunct <sup>4</sup>	wctype.h	int iswpunct (wint_t <i>wc</i> );	Test for a wide non-alphanumeric, non-space character.
iswspace <sup>4</sup>	wctype.h	int iswspace (wint_t <i>wc</i> );	Checks for any wide character that corresponds to an implementation-defined set of wide characters for which iswalnum is false.
iswupper <sup>4</sup>	wctype.h	int iswupper (wint_t <i>wc</i> );	Checks for any uppercase wide character.
iswxdigit <sup>4</sup>	wctype.h	int iswxdigit (wint_t <i>wc</i> );	Checks for any hexadecimal digit character.
isxdigit <sup>4</sup>	wctype.h	int isxdigit(int <i>c</i> );	Tests if <i>c</i> is a hexadecimal digit.
j0	math.h	double j0(double <i>x</i> );	Calculates the Bessel function value of the first kind of order 0.
j1	math.h	double j1(double <i>x</i> );	Calculates the Bessel function value of the first kind of order 1.
jn	math.h	double jn(int <i>n</i> , double <i>x</i> );	Calculates the Bessel function value of the first kind of order <i>n</i> .
labs	stdlib.h	long int labs(long int <i>n</i> );	Calculates the absolute value of <i>n</i> .
ldexp	math.h	double ldexp(double <i>x</i> , int <i>exp</i> );	Returns the value of <i>x</i> multiplied by (2 to the power of <i>exp</i> ).

Table 32. Standard C Library Functions (continued)

Function	System Include File	Function Prototype	Description
ldiv	stdlib.h	ldiv_t ldiv(long int <i>numerator</i> , long int <i>denominator</i> );	Calculates the quotient and remainder of <i>numerator/denominator</i> .
localeconv	locale.h	struct lconv *localeconv(void);	Formats numeric quantities in struct lconv according to the current locale.
localtime	time.h	struct tm *localtime(const time_t * <i>timeval</i> );	Converts <i>timeval</i> to a structure of type tm.
localtime64	time.h	struct tm *localtime64(const time64_t * <i>timeval</i> );	Converts <i>timeval</i> to a structure of type tm.
localtime_r	time.h	struct tm *localtime_r (const time_t * <i>timeval</i> , struct tm * <i>result</i> );	Converts a <i>time</i> value to a structure of type tm. (Restartable version of localtime.)
localtime64_r	time.h	struct tm *localtime64_r (const time64_t * <i>timeval</i> , struct tm * <i>result</i> );	Converts a <i>time</i> value to a structure of type tm. (Restartable version of localtime64.)
log	math.h	double log(double <i>x</i> );	Calculates the natural logarithm of <i>x</i> .
log10	math.h	double log10(double <i>x</i> );	Calculates the base 10 logarithm of <i>x</i> .
longjmp	setjmp.h	void longjmp(jmp_buf <i>env</i> , int <i>value</i> );	Restores a stack environment previously set in <i>env</i> by the setjmp function.
malloc	stdlib.h	void *malloc(size_t <i>size</i> );	Reserves a block of storage.
mblen	stdlib.h	int mblen(const char * <i>string</i> , size_t <i>n</i> );	Determines the length of a multibyte character <i>string</i> .
mbrlen <sup>4</sup>	wchar.h	int mbrlen (const char * <i>s</i> , size_t <i>n</i> , mbstate_t * <i>ps</i> );	Determines the length of a multibyte character. (Restartable version of mblen.)
mbrtowc <sup>4</sup>	wchar.h	int mbrtowc (wchar_t * <i>pwc</i> , const char * <i>s</i> , size_t <i>n</i> , mbstate_t * <i>ps</i> );	Convert a multibyte character to a wide character (Restartable version of mbtowc.)
mbsinit <sup>4</sup>	wchar.h	int mbsinit (const mbstate_t * <i>ps</i> );	Test state object * <i>ps</i> for initial state.
mbsrtowcs <sup>4</sup>	wchar.h	size_t mbsrtowc (wchar_t * <i>dst</i> , const char ** <i>src</i> , size_t <i>len</i> , mbstate_t * <i>ps</i> );	Convert multibyte string to a wide character string. (Restartable version of mbstowcs.)
mbstowcs	stdlib.h	size_t mbstowcs(wchar_t * <i>pwc</i> , const char * <i>string</i> , size_t <i>n</i> );	Converts the multibyte characters in <i>string</i> to their corresponding wchar_t codes, and stores not more than <i>n</i> codes in <i>pwc</i> .
mbtowc	stdlib.h	int mbtowc(wchar_t * <i>pwc</i> , const char * <i>string</i> , size_t <i>n</i> );	Stores the wchar_t code corresponding to the first <i>n</i> bytes of multibyte character <i>string</i> into the wchar_t character <i>pwc</i> .
memchr	string.h	void *memchr(const void * <i>buf</i> , int <i>c</i> , size_t <i>count</i> );	Searches the first <i>count</i> bytes of <i>buf</i> for the first occurrence of <i>c</i> converted to an unsigned character.
memcmp	string.h	int memcmp(const void * <i>buf1</i> , const void * <i>buf2</i> , size_t <i>count</i> );	Compares up to <i>count</i> bytes of <i>buf1</i> and <i>buf2</i> .
memcpy	string.h	void *memcpy(void * <i>dest</i> , const void * <i>src</i> , size_t <i>count</i> );	Copies <i>count</i> bytes of <i>src</i> to <i>dest</i> .

Table 32. Standard C Library Functions (continued)

Function	System Include File	Function Prototype	Description
memmove	string.h	void *memmove(void *dest, const void *src, size_t count);	Copies <i>count</i> bytes of <i>src</i> to <i>dest</i> . Allows copying between objects that overlap.
memset	string.h	void *memset(void *dest, int c, size_t count);	Sets <i>count</i> bytes of <i>dest</i> to a value <i>c</i> .
mktime	time.h	time_t mktime(struct tm *time);	Converts local <i>time</i> into calendar time.
mktime64	time.h	time64_t mktime64(struct tm *time);	Converts local <i>time</i> into calendar time.
modf	math.h	double modf(double x, double *intptr);	Breaks down the floating-point value <i>x</i> into fractional and integral parts.
nextafter	math.h	double nextafter(double x, double y);	Calculates the next representable value after <i>x</i> in the direction of <i>y</i> .
nextafterl	math.h	long double nextafterl(long double x, long double y);	Calculates the next representable value after <i>x</i> in the direction of <i>y</i> .
nexttoward	math.h	double nexttoward(double x, long double y);	Calculates the next representable value after <i>x</i> in the direction of <i>y</i> .
nexttowardl	math.h	long double nexttowardl(long double x, long double y);	Calculates the next representable value after <i>x</i> in the direction of <i>y</i> .
nl_langinfo <sup>4</sup>	langinfo.h	char *nl_langinfo(nl_item item);	Retrieve from the current locale the string that describes the requested information specified by <i>item</i> .
perror	stdio.h	void perror(const char *string);	Prints an error message to stderr.
pow	math.h	double pow(double x, double y);	Calculates the value <i>x</i> to the power <i>y</i> .
printf	stdio.h	int printf(const char *format-string, arg-list);	Formats and prints characters and values to stdout.
putc <sup>1</sup>	stdio.h	int putc(int c, FILE *stream);	Prints <i>c</i> to the output <i>stream</i> .
putchar <sup>1</sup>	stdio.h	int putchar(int c);	Prints <i>c</i> to stdout.
putenv	stdlib.h	int *putenv(const char *varname);	Sets the value of an environment variable by altering an existing variable or creating a new one.
puts	stdio.h	int puts(const char *string);	Prints a string to stdout.
putwc <sup>6</sup>	stdio.h wchar.h	wint_t putwchar(wchar_t wc, FILE *stream);	Converts the wide character <i>wc</i> to a multibyte character, and writes it to the stream at the current position.
putwchar <sup>6</sup>	wchar.h	wint_t putwchar(wchar_t wc);	Converts the wide character <i>wc</i> to a multibyte character and writes it to stdout.
qsort	stdlib.h	void qsort(void *base, size_t num, size_t width, int(*compare)(const void *element1, const void *element2));	Performs a quick sort of an array of <i>num</i> elements, each of <i>width</i> bytes in size.
raise	signal.h	int raise(int sig);	Sends the signal <i>sig</i> to the running program.
rand	stdlib.h	int rand(void);	Returns a pseudo-random integer.

Table 32. Standard C Library Functions (continued)

Function	System Include File	Function Prototype	Description
rand_r	stdlib.h	int rand_r(void);	Returns a pseudo-random integer. (Restartable version)
realloc	stdlib.h	void *realloc(void *ptr, size_t size);	Changes the <i>size</i> of a previously reserved storage block.
regcomp	regex.h	int regcomp(regex_t *preg, const char *pattern, int cflags);	Compiles the source regular expression pointed to by <i>pattern</i> into an executable version and stores it in the location pointed to by <i>preg</i> .
regerror	regex.h	size_t regerror(int errcode, const regex_t *preg, char *errbuf, size_t errbuf_size);	Finds the description for the error code <i>errcode</i> for the regular expression <i>preg</i> .
regexec	regex.h	int regexec(const regex_t *preg, const char *string, size_t nmatch, regmatch_t *pmatch, int eflags);	Compares the null-ended string <i>string</i> against the compiled regular expression <i>preg</i> to find a match between the two.
regfree	regex.h	void regfree(regex_t *preg);	Frees any memory that was allocated by regcomp to implement the regular expression <i>preg</i> .
remove	stdio.h	int remove(const char *filename);	Deletes the file specified by <i>filename</i> .
rename	stdio.h	int rename(const char *oldname, const char *newname);	Renames the specified file.
rewind <sup>1</sup>	stdio.h	void rewind(FILE *stream);	Repositions the file pointer associated with <i>stream</i> to the beginning of the file.
scanf	stdio.h	int scanf(const char *format-string, arg-list);	Reads data from stdin into locations given by <i>arg-list</i> .
setbuf	stdio.h	void setbuf(FILE *stream, char *buffer);	Controls buffering for <i>stream</i> .
setjmp	setjmp.h	int setjmp(jmp_buf env);	Saves a stack environment that can be subsequently restored by longjmp.
setlocale	locale.h	char *setlocale(int category, const char *locale);	Changes or queries variables defined in the <i>locale</i> .
setvbuf	stdio.h	int setvbuf(FILE *stream, char *buf, int type, size_t size);	Controls buffering and buffer <i>size</i> for <i>stream</i> .
signal	signal.h	void(*signal(int sig, void(*func)(int)))(int);	Registers <i>func</i> as a signal handler for the signal <i>sig</i> .
sin	math.h	double sin(double x);	Calculates the sine of <i>x</i> .
sinh	math.h	double sinh(double x);	Calculates the hyperbolic sine of <i>x</i> .
snprintf	stdio.h	int snprintf(char *outbuf, size_t n, const char*, ...)	Same as sprintf except that the function will stop after <i>n</i> characters have been written to <i>outbuf</i> .
sprintf	stdio.h	int sprintf(char *buffer, const char *format-string, arg-list);	Formats and stores characters and values in <i>buffer</i> .
sqrt	math.h	double sqrt(double x);	Calculates the square root of <i>x</i> .
srand	stdlib.h	void srand(unsigned int seed);	Sets the <i>seed</i> for the pseudo-random number generator.

Table 32. Standard C Library Functions (continued)

Function	System Include File	Function Prototype	Description
sscanf	stdio.h	int sscanf(const char *buffer, const char *format, arg-list);	Reads data from <i>buffer</i> into the locations given by <i>arg-list</i> .
strcasemp	strings.h	int strcasemp(const char *string1, const char *string2);	Compares strings without case sensitivity.
strcat	string.h	char *strcat(char *string1, const char *string2);	Concatenates <i>string2</i> to <i>string1</i> .
strchr	string.h	char *strchr(const char *string, int c);	Locates the first occurrence of <i>c</i> in <i>string</i> .
strcmp	string.h	int strcmp(const char *string1, const char *string2);	Compares the value of <i>string1</i> to <i>string2</i> .
strcoll	string.h	int strcoll(const char *string1, const char *string2);	Compares two strings using the collating sequence in the current locale.
strcpy	string.h	char *strcpy(char *string1, const char *string2);	Copies <i>string2</i> into <i>string1</i> .
strcspn	string.h	size_t strcspn(const char *string1, const char *string2);	Returns the length of the initial substring of <i>string1</i> consisting of characters not contained in <i>string2</i> .
strerror	string.h	char *strerror(int errnum);	Maps the error number in <i>errnum</i> to an error message string.
strfmon <sup>4</sup>	wchar.h	int strfmon(char *s, size_t maxsize, const char *format, ...);	Converts monetary value to string.
strftime	time.h	size_t strftime(char *dest, size_t maxsize, const char *format, const struct tm *timeptr);	Stores characters in an array pointed to by <i>dest</i> , according to the string determined by <i>format</i> .
strlen	string.h	size_t strlen(const char *string);	Calculates the length of <i>string</i> .
strncasemp	strings.h	int strncasemp(const char *string1, const char *string2, size_t count);	Compares strings without case sensitivity.
strncat	string.h	char *strncat(char *string1, const char *string2, size_t count);	Concatenates up to <i>count</i> characters of <i>string2</i> to <i>string1</i> .
strncmp	string.h	int strncmp(const char *string1, const char *string2, size_t count);	Compares up to <i>count</i> characters of <i>string1</i> and <i>string2</i> .
strncpy	string.h	char *strncpy(char *string1, const char *string2, size_t count);	Copies up to <i>count</i> characters of <i>string2</i> to <i>string1</i> .
strpbrk	string.h	char *strpbrk(const char *string1, const char *string2);	Locates the first occurrence in <i>string1</i> of any character in <i>string2</i> .
strptime <sup>4</sup>	time.h	char *strptime(const char *buf, const char *format, struct tm *tm);	Date and time conversion
strrchr	string.h	char *strrchr(const char *string, int c);	Locates the last occurrence of <i>c</i> in <i>string</i> .
strspn	string.h	size_t strspn(const char *string1, const char *string2);	Returns the length of the initial substring of <i>string1</i> consisting of characters contained in <i>string2</i> .

Table 32. Standard C Library Functions (continued)

Function	System Include File	Function Prototype	Description
strstr	string.h	char *strstr(const char *string1, const char *string2);	Returns a pointer to the first occurrence of <i>string2</i> in <i>string1</i> .
strtod	stdlib.h	double strtod(const char *nptr, char **endptr);	Converts <i>nptr</i> to a double precision value.
strtod32	stdlib.h	_Decimal32 strtod32(const char *nptr, char **endptr);	Converts <i>nptr</i> to a single-precision decimal floating-point value.
strtod64	stdlib.h	_Decimal64 strtod64(const char *nptr, char **endptr);	Converts <i>nptr</i> to a double-precision decimal floating-point value.
strtod128	stdlib.h	_Decimal128 strtod128(const char *nptr, char **endptr);	Converts <i>nptr</i> to a quad-precision decimal floating-point value.
strtof	stdlib.h	float strtof(const char *nptr, char **endptr);	Converts <i>nptr</i> to a float value.
strtok	string.h	char *strtok(char *string1, const char *string2);	Locates the next token in <i>string1</i> delimited by the next character in <i>string2</i> .
strtok_r	string.h	char *strtok_r(char *string, const char *seps, char **lasts);	Locates the next token in <i>string</i> delimited by the next character in <i>seps</i> . (Restartable version of <i>strtok</i> .)
strtol	stdlib.h	long int strtol(const char *nptr, char **endptr, int base);	Converts <i>nptr</i> to a signed long integer.
strtold	stdlib.h	long double strtold(const char *nptr, char **endptr);	Converts <i>nptr</i> to a long double value.
strtoul	stdlib.h	unsigned long int strtoul(const char *string1, char **string2, int base);	Converts <i>string1</i> to an unsigned long integer.
strxfrm	string.h	size_t strxfrm(char *string1, const char *string2, size_t count);	Converts <i>string2</i> and places the result in <i>string1</i> . The conversion is determined by the program's current locale.
swprintf	wchar.h	int swprintf(wchar_t *wcsbuffer, size_t n, const wchar_t *format, arg-list);	Formats and stores a series of wide characters and values into the wide-character buffer <i>wcsbuffer</i> .
swscanf	wchar.h	int swscanf (const wchar_t *buffer, const wchar_t *format, arg-list)	Reads data from <i>buffer</i> into the locations given by <i>arg-list</i> .
system	stdlib.h	int system(const char *string);	Passes <i>string</i> to the system command analyzer.
tan	math.h	double tan(double x);	Calculates the tangent of <i>x</i> .
tanh	math.h	double tanh(double x);	Calculates the hyperbolic tangent of <i>x</i> .
time	time.h	time_t time(time_t *timeptr);	Returns the current calendar time.
time64	time.h	time64_t time64(time64_t *timeptr);	Returns the current calendar time.
tmpfile	stdio.h	FILE *tmpfile(void);	Creates a temporary binary file and opens it.
tmpnam	stdio.h	char *tmpnam(char *string);	Generates a temporary file name.
toascii	ctype.h	int toascii(int c);	Converts <i>c</i> to a character in the 7-bit US-ASCII character set.

Table 32. Standard C Library Functions (continued)

Function	System Include File	Function Prototype	Description
tolower	ctype.h	int tolower(int c);	Converts <i>c</i> to lowercase.
toupper	ctype.h	int toupper(int c);	Converts <i>c</i> to uppercase.
towctrans	wctype.h	wint_t towctrans(wint_t <i>wc</i> , wctrans_t <i>desc</i> );	Translates the wide character <i>wc</i> based on the mapping described by <i>desc</i> .
tolower <sup>4</sup>	wctype.h	wint_t tolower (wint_t <i>wc</i> );	Converts uppercase letter to lowercase letter.
toupper <sup>4</sup>	wctype.h	wint_t toupper (wint_t <i>wc</i> );	Converts lowercase letter to uppercase letter.
ungetc <sup>1</sup>	stdio.h	int ungetc(int <i>c</i> , FILE * <i>stream</i> );	Pushes <i>c</i> back onto the input <i>stream</i> .
ungetwc <sup>6</sup>	stdio.h wchar.h	wint_t ungetwc(wint_t <i>wc</i> , FILE * <i>stream</i> );	Pushes the wide character <i>wc</i> back onto the input stream.
va_arg	stdarg.h	<i>var_type</i> va_arg(va_list <i>arg_ptr</i> , <i>var_type</i> );	Returns the value of one argument and modifies <i>arg_ptr</i> to point to the next argument.
va_end	stdarg.h	void va_end(va_list <i>arg_ptr</i> );	Facilitates normal return from variable argument list processing.
va_start	stdarg.h	void va_start(va_list <i>arg_ptr</i> , <i>variable_name</i> );	Initializes <i>arg_ptr</i> for subsequent use by <i>va_arg</i> and <i>va_end</i> .
vfprintf	stdio.h stdarg.h	int vfprintf(FILE * <i>stream</i> , const char * <i>format</i> , va_list <i>arg_ptr</i> );	Formats and prints characters to the output <i>stream</i> using a variable number of arguments.
vfscanf	stdio.h stdarg.h	int vfscanf(FILE * <i>stream</i> , const char * <i>format</i> , va_list <i>arg_ptr</i> );	Reads data from a specified stream into locations given by a variable number of arguments.
vfwprintf <sup>6</sup>	stdarg.h stdio.h wchar.h	int vfwprintf(FILE * <i>stream</i> , const wchar_t * <i>format</i> , va_list <i>arg</i> );	Equivalent to fwprintf, except that the variable argument list is replaced by <i>arg</i> .
vfwscanf	stdio.h stdarg.h	int vfwscanf(FILE * <i>stream</i> , const wchar_t * <i>format</i> , va_list <i>arg_ptr</i> );	Reads wide data from a specified stream into locations given by a variable number of arguments.
vprintf	stdio.h stdarg.h	int vprintf(const char * <i>format</i> , va_list <i>arg_ptr</i> );	Formats and prints characters to stdout using a variable number of arguments.
vscanf	stdio.h stdarg.h	int vscanf(const char * <i>format</i> , va_list <i>arg_ptr</i> );	Reads data from stdin into locations given by a variable number of arguments.
vsprintf	stdio.h stdarg.h	int vsprintf(char * <i>target-string</i> , const char * <i>format</i> , va_list <i>arg_ptr</i> );	Formats and stores characters in a buffer using a variable number of arguments.
vsnprintf	stdio.h	int vsnprintf(char * <i>outbuf</i> , size_t <i>n</i> , const char*, va_list);	Same as vsprintf except that the function will stop after <i>n</i> characters have been written to <i>outbuf</i> .
vsscanf	stdio.h stdarg.h	int vsscanf(const char* <i>buffer</i> , const char * <i>format</i> , va_list <i>arg_ptr</i> );	Reads data from a buffer into locations given by a variable number of arguments.
vswprintf	stdarg.h wchar.h	int vswprintf(wchar_t * <i>wcsbuffer</i> , size_t <i>n</i> , const wchar_t * <i>format</i> , va_list <i>arg</i> );	Formats and stores a series of wide characters and values in the buffer <i>wcsbuffer</i> .

Table 32. Standard C Library Functions (continued)

Function	System Include File	Function Prototype	Description
vswscanf	stdio.h wchar.h	int vswscanf(const wchar_t *buffer, const wchar_t *format, va_list arg_ptr);	Reads wide data from a buffer into locations given by a variable number of arguments.
vwprintf <sup>6</sup>	stdarg.h wchar.h	int vwprintf(const wchar_t *format, va_list arg);	Equivalent to wprintf, except that the variable argument list is replaced by <i>arg</i> .
vwscanf	stdio.h wchar.h	int vwscanf(const wchar_t *format, va_list arg_ptr);	Reads wide data from stdin into locations given by a variable number of arguments.
wcrtomb <sup>4</sup>	wchar.h	int wcrtomb(char *s, wchar_t wchar, mbstate_t *pss);	Converts a wide character to a multibyte character. (Restartable version of wctomb.)
wcscat	wchar.h	wchar_t *wcscat(wchar_t *string1, const wchar_t *string2);	Appends a copy of the string pointed to by <i>string2</i> to the end of the string pointed to by <i>string1</i> .
wcschr	wchar.h	wchar_t *wcschr(const wchar_t *string, wchar_t character);	Searches the wide-character string pointed to by <i>string</i> for the occurrence of <i>character</i> .
wcscmp	wchar.h	int wcscmp(const wchar_t *string1, const wchar_t *string2);	Compares two wide-character strings, <i>*string1</i> and <i>*string2</i> .
wscoll <sup>4</sup>	wchar.h	int wscoll(const wchar_t *wcs1, const wchar_t *wcs2);	Compares two wide-character strings using the collating sequence in the current locale.
wscpy	wchar.h	wchar_t *wscpy(wchar_t *string1, const wchar_t *string2);	Copies the contents of <i>*string2</i> (including the ending wchar_t null character) into <i>*string1</i> .
wcscspn	wchar.h	size_t wcscspn(const wchar_t *string1, const wchar_t *string2);	Determines the number of wchar_t characters in the initial segment of the string pointed to by <i>*string1</i> that do not appear in the string pointed to by <i>*string2</i> .
wcsftime	wchar.h	size_t wcsftime(wchar_t *wdest, size_t maxsize, const wchar_t *format, const struct tm *timeptr);	Converts the time and date specification in the <i>timeptr</i> structure into a wide-character string.
wcslen	wchar.h	size_t wcslen(const wchar_t *string);	Computes the number of wide-characters in the string pointed to by <i>string</i> .
wcslocaleconv	locale.h	struct wcsconv *wcslocaleconv(void);	Formats numeric quantities in struct <i>wcsconv</i> according to the current locale.
wcsncat	wchar.h	wchar_t *wcsncat(wchar_t *string1, const wchar_t *string2, size_t count);	Appends up to <i>count</i> wide characters from <i>string2</i> to the end of <i>string1</i> , and appends a wchar_t null character to the result.
wcsncmp	wchar.h	int wcsncmp(const wchar_t *string1, const wchar_t *string2, size_t count);	Compares up to <i>count</i> wide characters in <i>string1</i> to <i>string2</i> .

Table 32. Standard C Library Functions (continued)

Function	System Include File	Function Prototype	Description
wcsncpy	wchar.h	wchar_t *wcsncpy(wchar_t *string1, const wchar_t *string2, size_t count);	Copies up to <i>count</i> wide characters from <i>string2</i> to <i>string1</i> .
wcspbrk	wchar.h	wchar_t *wcspbrk(const wchar_t *string1, const wchar_t *string2);	Locates the first occurrence in the string pointed to by <i>string1</i> of any wide characters from the string pointed to by <i>string2</i> .
wcsptime	wchar.h	wchar_t *wcsptime ( const wchar_t *buf, const wchar_t *format, struct tm *tm );	Date and time conversion. Equivalent to <code>strptime()</code> , except that it uses wide characters.
wcsrchr	wchar.h	wchar_t *wcsrchr(const wchar_t *string, wchar_t character);	Locates the last occurrence of <i>character</i> in the string pointed to by <i>string</i> .
wcsrtombs <sup>4</sup>	wchar.h	size_t wcsrtombs (char *dst, const wchar_t **src, size_t len, mbstate_t *ps);	Converts wide character string to multibyte string. (Restartable version of <code>wcstombs</code> .)
wcsspn	wchar.h	size_t wcsspn(const wchar_t *string1, const wchar_t *string2);	Computes the number of wide characters in the initial segment of the string pointed to by <i>string1</i> , which consists entirely of wide characters from the string pointed to by <i>string2</i> .
wcsstr	wchar.h	wchar_t *wcsstr(const wchar_t *wcs1, const wchar_t *wcs2);	Locates the first occurrence of <i>wcs2</i> in <i>wcs1</i> .
wcstod	wchar.h	double wcstod(const wchar_t *nptr, wchar_t **endptr);	Converts the initial portion of the wide-character string pointed to by <i>nptr</i> to a double value.
wcstod32	wchar.h	_Decimal32 wcstod32(const wchar_t *nptr, wchar_t **endptr);	Converts the initial portion of the wide-character string pointed to by <i>nptr</i> to a single-precision decimal floating-point value.
wcstod64	wchar.h	_Decimal64 wcstod64(const wchar_t *nptr, wchar_t **endptr);	Converts the initial portion of the wide-character string pointed to by <i>nptr</i> to a double-precision decimal floating-point value.
wcstod128	wchar.h	_Decimal128 wcstod128(const wchar_t *nptr, wchar_t **endptr);	Converts the initial portion of the wide-character string pointed to by <i>nptr</i> to a quad-precision decimal floating-point value.
wcstok	wchar.h	wchar_t *wcstok(wchar_t *wcs1, const wchar_t *wcs2, wchar_t **ptr)	Breaks <i>wcs1</i> into a sequence of tokens, each of which is delimited by a wide character from the wide string pointed to by <i>wcs2</i> .
wcstol	wchar.h	long int wcstol(const wchar_t *nptr, wchar_t **endptr, int base);	Converts the initial portion of the wide-character string pointed to by <i>nptr</i> to a long integer value.
wcstombs	stdlib.h	size_t wcstombs(char *dest, const wchar_t *string, size_t count);	Converts the <i>wchar_t</i> <i>string</i> into a multibyte string <i>dest</i> .

Table 32. Standard C Library Functions (continued)

Function	System Include File	Function Prototype	Description
wcstoul	wchar.h	unsigned long int wcstoul(const wchar_t *nptr, wchar_t **endptr, int base);	Converts the initial portion of the wide-character string pointed to by <i>nptr</i> to an unsigned long integer value.
wcsxfrm <sup>4</sup>	wchar.h	size_t wcsxfrm (wchar_t *wcs1, const wchar_t *wcs2, size_t n);	Transforms a wide-character string to values which represent character collating weights and places the resulting wide-character string into an array.
wctob	stdarg.h wchar.h	int wctob(wint_t wc);	Determines whether <i>wc</i> corresponds to a member of the extended character set whose multibyte character representation is a single byte when in the initial shift state.
wctomb	stdlib.h	int wctomb(char *string, wchar_t character);	Converts the wchar_t value of <i>character</i> into a multibyte <i>string</i> .
wctrans	wctype.h	wctrans_t wctrans(const char *property);	Constructs a value with type wctrans_t that describes a mapping between wide characters identified by the string argument property.
wctype <sup>4</sup>	wchar.h	wctype_t wctype (const char *property);	Obtains handle for character property classification.
wcwidth	wchar.h	int wcwidth(const wchar_t *pwcs, size_t n);	Determine the display width of a wide character string.
wmemchr	wchar.h	wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);	Locates the first occurrence of <i>c</i> in the initial <i>n</i> wide characters of the object pointed to by <i>s</i> .
wmemcmp	wchar.h	int wmemcmp(const wchar_t *s1, const wchar_t *s2, size_t n);	Compares the first <i>n</i> wide characters of the object pointed to by <i>s1</i> to the first <i>n</i> characters of the object pointed to by <i>s2</i> .
wmemcpy	wchar.h	wchar_t *wmemcpy(wchar_t *s1, const wchar_t *s2, size_t n);	Copies <i>n</i> wide characters from the object pointed to by <i>s2</i> to the object pointed to by <i>s1</i> .
wmemmove	wchar.h	wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);	Copies <i>n</i> wide characters from the object pointed to by <i>s2</i> to the object pointed to by <i>s1</i> .
wmemset	wchar.h	wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);	Copies the value of <i>c</i> into each of the first <i>n</i> wide characters of the object pointed to by <i>s</i> .
wprintf <sup>6</sup>	wchar.h	int wprintf(const wchar_t *format, arg-list);	Equivalent to fprintf with the argument stdout interposed before the arguments to printf.
wscanf <sup>6</sup>	wchar.h	int wscanf(const wchar_t *format, arg-list);	Equivalent to fscanf with the argument stdin interposed before the arguments of wscanf.
y0	math.h	double y0(double x);	Calculates the Bessel function value of the second kind of order 0.
y1	math.h	double y1(double x);	Calculates the Bessel function value of the second kind of order 1.

Table 32. Standard C Library Functions (continued)

Function	System Include File	Function Prototype	Description
yn	math.h	double yn(int <i>n</i> , double <i>x</i> );	Calculates the Bessel function value of the second kind of order <i>n</i> .
<p><b>Note:</b> <sup>1</sup> This function is not supported for files opened with type=record.  <b>Note:</b> <sup>2</sup> This function is not supported for files opened with type=record and mode=ab+, rb+, or wb+.  <b>Note:</b> <sup>3</sup> The ILE C compiler only supports fully buffered and line-buffered streams. Since a block and a line are equal to the record length of the opened file, fully buffered and line-buffered streams are supported in the same way. The setbuf() and setvbuf() functions have no effect.  <b>Note:</b> <sup>4</sup> This function is not available when LOCALETYPE(*CLD) is specified on the compilation command.  <b>Note:</b> <sup>5</sup> This function is available only when SYSIFCOPT(*IFSIO) is specified on the CRTCMOD or CRTBNDC command.  <b>Note:</b> <sup>6</sup> This function is not available when either LOCALETYPE(*CLD) or SYSIFCOPT(*NOIFSIO) is specified on the compilation command.</p>			

## ILE C Library Extensions to C Library Functions Table

This table briefly describes all the ILE C library extensions, listed in alphabetical order. This table provides the include file name, and the function prototype for each function.

Table 33. ILE C Library Extensions

Function	System Include file	Function prototype	Description
_C_Get_Ssn_Handle	stdio.h	_SSN_Handle_T _C_Get_Ssn_Handle (void);	Returns a handle to the C session for use with DSM APIs.
_C_Quickpool_Debug	stdio.h	_C_Quickpool_Debug_T _C_Quickpool_Debug(_C_Quickpool_Debug_T *newval);	Modifies Quick Pool memory characteristics.
_C_Quickpool_Init	stdio.h	int _C_Quickpool_Init(unsigned int numpools, unsigned int *cell_sizes, unsigned int *num_cells);	Initializes the use of the Quick Pool memory management algorithm.
_C_Quickpool_Report	stdio.h	void _C_Quickpool_Report(void);	Generates a spooled file that contains a snapshot of the memory used by the Quick Pool memory management algorithm in the current activation group.
_C_TS_malloc64	stdlib.h	void *_C_TS_malloc64(unsigned long long int);	Same as _C_TS_malloc, but takes an unsigned long long int so the user can ask for more than 2 GB of storage on a single request.
_C_TS_malloc_info	mallocinfo.h	int _C_TS_malloc_info(struct _C_mallinfo_t *output_record, size_t sizeofoutput);	Returns current memory usage information.
_C_TS_malloc_debug	mallocinfo.h	int _C_TS_malloc_debug(unsigned int dump_level, unsigned int verify_level, struct _C_mallinfo_t *output_record, size_t sizeofoutput);	Returns the same information as _C_TS_malloc_info, plus produces a spool file of detailed information about the memory structure used by C_TS_malloc functions.
_GetExcData	signal.h	void _GetExcData (_INTRPT_Hndlr_Parms_T *parms);	Retrieves information about an exception from within a signal handler.
QXXCHGDA	xxdtaa.h	void QXXCHGDA(_DTAA_NAME_T dtaname, short int offset, short int len, char *dtaptr);	Changes the i5/OS data area specified on dtaname using the data pointed to by dtaptr.
QXXDTOP	xxcvt.h	void QXXDTOP(unsigned char *pptr, int digits, int fraction, double value);	Converts a double value to a packed decimal value with digits total digits and fraction fractional digits.

Table 33. ILE C Library Extensions (continued)

Function	System Include file	Function prototype	Description
QXXDTOZ	xxcvt.h	void QXXDTOZ(unsigned char *zptr, int digits, int fraction, double value);	Converts a double value to a zoned decimal value with <i>digits</i> total digits and <i>fraction</i> fractional digits.
QXXITOP	xxcvt.h	void QXXITOP(unsigned char *pptr, int digits, int fraction, int value);	Converts an integer value to a packed decimal value.
QXXITOZ	xxcvt.h	void QXXITOZ(unsigned char *zptr, int digits, int fraction, int value);	Converts an integer value to a zoned decimal value.
QXXPTOD	xxcvt.h	double QXXPTOD(unsigned char *pptr, int digits, int fraction );	Converts a packed decimal number to a double value with <i>digits</i> total digits and <i>fraction</i> fractional digits.
QXXPTOI	xxcvt.h	int QXXPTOI(unsigned char *pptr, int digits, int fraction );	Converts a packed decimal number to an integer value with <i>digits</i> total digits and <i>fraction</i> fractional digits.
QXXRTVDA	xxdtaa.h	void QXXRTVDA(_DTAA_NAME_T dtaname, short int offset, short int len, char *dtaptr);	Retrieves a copy of the i5/OS data area specified on <i>dtaname</i> .
QXXZTOD	xxcvt.h	double QXXZTOD(unsigned char *zptr, int digits, int fraction );	Converts a zoned decimal number to a double value with <i>digits</i> total digits and <i>fraction</i> fractional digits.
QXXZTOI	xxcvt.h	int QXXZTOI(unsigned char *zptr, int digits, int fraction );	Converts a zoned decimal value to an integer value with <i>digits</i> total digits and <i>fraction</i> fractional digits.
_Racquire	recio.h	int _Racquire(_RFILE *fp, char *dev);	Prepares a device for record I/O operations.
_Rclose	recio.h	int _Rclose(_RFILE *fp);	Closes a file that is opened for record I/O operations.
_Rcommit	recio.h	int _Rcommit(char *cmtid);	Completes the current transaction, and establishes a new commitment boundary.
_Rdelete	recio.h	_RIOFB_T *_Rdelete(_RFILE *fp);	Deletes the currently locked record.
_Rdevatr	xxfdbk.h recio.h	_XXDEV_ATTR_T *_Rdevatr(_RFILE *fp, char *pgmdev);	Returns a pointer to a copy of the device attributes feedback area for the file referenced by <i>fp</i> and the device <i>pgmdev</i> .
_Rfeod	recio.h	int _Rfeod(_RFILE *fp);	Forces an end-of-file condition for the file referenced by <i>fp</i> .
_Rfeov	recio.h	int _Rfeov(_RFILE *fp);	Forces an end-of-volume condition for the tape file referenced by <i>fp</i> .
_Rformat	recio.h	void Rformat(_RFILE *fp, char *fmt);	Sets the record format to <i>fmt</i> for the file referenced by <i>fp</i> .
_Rindara	recio.h	void _Rindara (_RFILE *fp, char *indic_buf);	Sets up the separate indicator area to be used for subsequent record I/O operations.
_Riofbk	recio.h xxfdbk.h	_XXIOFB_T *_Riofbk(_RFILE *fp);	Returns a pointer to a copy of the I/O feedback area for the file referenced by <i>fp</i> .
_Rlocate	recio.h	_RIOFB_T *_Rlocate(_RFILE *fp, void *key, int klen_rrn, int opts);	Positions to the record in the file associated with <i>fp</i> and specified by the <i>key</i> , <i>klen_rrn</i> and <i>opt</i> parameters.
_Ropen	recio.h	_RFILE *_Ropen(const char *filename, const char *mode ...);	Opens a file for record I/O operations.
_Ropnfbk	recio.h xxfdbk.h	_XXOPFB_T *_Ropnfbk(_RFILE *fp);	Returns a pointer to a copy of the open feedback area for the file referenced by <i>fp</i> .
_Rpgmdev	recio.h	int _Rpgmdev(_RFILE *fp, char *dev);	Sets the default program device.

Table 33. ILE C Library Extensions (continued)

Function	System Include file	Function prototype	Description
_Rread	recio.h	_RIOFB_T *_Rread(_RFILE *fp, void *buf, size_t size, int opts, long rrm);	Reads a record by relative record number.
_Rreadf	recio.h	_RIOFB_T *_Rreadf(_RFILE *fp, void *buf, size_t size, int opts);	Reads the first record.
_Rreadindv	recio.h	_RIOFB_T *_Rreadindv(_RFILE *fp, void *buf, size_t size, int opts);	Reads a record from an invited device.
_Rreadk	recio.h	_RIOFB_T *_Rreadk(_RFILE *fp, void *buf, size_t size, int opts, void *key, int klen);	Reads a record by key.
_Rreadl	recio.h	_RIOFB_T *_Rreadl(_RFILE *fp, void *buf, size_t size, int opts);	Reads the last record.
_Rreadn	recio.h	_RIOFB_T *_Rreadn(_RFILE *fp, void *buf, size_t size, int opts);	Reads the next record.
_Rreadnc	recio.h	_RIOFB_T *_Rreadnc(_RFILE *fp, void *buf, size_t size);	Reads the next changed record in the subfile.
_Rreadp	recio.h	_RIOFB_T *_Rreadp(_RFILE *fp, void *buf, size_t size, int opts);	Reads the previous record.
_Rreads	recio.h	_RIOFB_T *_Rreads(_RFILE *fp, void *buf, size_t size, int opts);	Reads the same record.
_Rrelease	recio.h	int _Rrelease(_RFILE *fp, char *dev);	Makes the specified device ineligible for record I/O operations.
_Rrlsck	recio.h	int _Rrlsck(_RFILE *fp);	Releases the currently locked record.
_Rrollbck	recio.h	int _Rrollbck(void);	Reestablishes the last commitment boundary as the current commitment boundary.
_Rupdate	recio.h	_RIOFB_T *_Rupdate(_RFILE *fp, void *buf, size_t size);	Writes to the record that is currently locked for update.
_Rupfb	recio.h	_RIOFB_T *_Rupfb(_RFILE *fp);	Updates the feedback structure with information about the last record I/O operation.
_Rwrite	recio.h	_RIOFB_T *_Rwrite(_RFILE *fp, void *buf, size_t size);	Writes a record to the end of the file.
_Rwrited	recio.h	_RIOFB_T *_Rwrited(_RFILE *fp, void *buf, size_t size, unsigned long rrm);	Writes a record by relative record number. It only writes over deleted records.
_Rwriterd	recio.h	_RIOFB_T *_Rwriterd(_RFILE *fp, void *buf, size_t size);	Reads and writes a record.
_Rwrread	recio.h	_RIOFB_T *_Rwrread(_RFILE *fp, void *inbuf, size_t in_buf_size, void *out_buf, size_t out_buf_size);	Functions as _Rwriterd, except separate buffers may be specified for input and output data.
__wscicmp	wchar.h	int __wscicmp(const wchar_t *string1, const wchar_t *string2);	Compares wide character strings without case sensitivity.
__wscnicmp	wchar.h	int __wscnicmp(const wchar_t *string1, const wchar_t *string2, size_t count);	Compares wide character strings without case sensitivity.



---

## Appendix B. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing  
IBM Corporation  
North Castle Drive  
Armonk, NY 10504-1785  
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation  
Licensing  
2-31 Roppongi 3-chome, Minato-ku  
Tokyo 106-0032, Japan

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation

Software Interoperability Coordinator, Department YBWA  
3605 Highway 52 N  
Rochester, MN 55901  
U.S.A.

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, IBM License Agreement for Machine Code, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

#### COPYRIGHT LICENSE:

This information contains sample application programs in source language, which 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.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. \_enter the year or years\_. All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

---

## Programming interface information

This ILE C/C++ Runtime Library Functions publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM i5/OS.

---

## Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

COBOL/400

DB2

i5/OS

IBM

IBM (logo)

Integrated Language Environment

RPG/400

System i

WebSphere

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.



---

## Bibliography

For additional information about topics related to ILE C/C++ programming on the System i platform, refer to the following IBM System i publications and i5/OS Information Center topics: (<http://www.ibm.com/systems/i/infocenter/>)

- The Application programming interfaces topic in the Programming category of the i5/OS Information Center provides information for experienced application and system programmers who want to use the i5/OS application programming interfaces (APIs).
- *Application Display Programming*, SC41-5715-02, provides information about using DDS to create and maintain displays, creating and working with display files, creating online help information, using UIM to define displays, and using panel groups, records, and documents.
- The Backup and recovery topic in the Systems management category of the i5/OS Information Center includes information about how to plan a backup and recovery strategy, how to back up your system, how to manage tape libraries, and how to set up disk protection for your data. It also includes information about the Backup, Recovery and Media Services plug-in to System i Navigator, information about recovering your system, and answers to some frequently asked questions about backup and recovery.
- *Recovering your system*, SC41-5304-09 provides general information about recovery and availability options for the System i platform. It describes the options available on the system, compares and contrasts them, and tells where to find more information about them.
- The Control language topic in the Programming category of the i5/OS Information Center provides a description of the i5/OS control language commands. It also provides a wide-ranging discussion of i5/OS programming topics including a general discussion on objects and libraries, CL programming, controlling flow and communicating between programs, working with objects in CL programs, and creating CL programs. Other topics include predefined and impromptu messages and message handling, defining and creating user-defined commands and menus, application testing, including debug mode, breakpoints, traces, and display functions.
- *Standard C/C++ Library Reference*, SC09-4949-01 contains reference information for the C/C++ languages.
- *Communications Management*, SC41-5406-02, provides information about work management in a communications environment, communications status, tracing and diagnosing communications problems, error handling and recovery, performance, and specific line speed and subsystem storage information.
- The Files and file systems category in the i5/OS Information Center provides information about using files in application programs.
- The i5/OS globalization topic in the Programming category of the i5/OS Information Center provides information for planning, installing, configuring, and using globalization and multilingual support of the System i product. It also provides an explanation of the database management of multilingual data and application considerations for a multilingual system.
- The *ICF Programming*, SC41-5442-00, manual provides information needed to write application programs that use i5/OS communications and the i5/OS intersystem communications function (i5/OS-ICF). It also contains information about data description specifications (DDS) keywords, system-supplied formats, return codes, file transfer support, and program examples.
- *ILE Concepts*, SC41-5606-08, explains concepts and terminology pertaining to the Integrated Language Environment architecture of the i5/OS licensed program. Topics covered include creating modules, binding, running programs, debugging programs, and handling exceptions.
- *ILE C/C++ for AS/400 MI Library Reference*, SC09-2418-00, provides information about Machine Interface instructions available in the ILE C compiler that provide system-level programming capabilities.
- The Printing category of information in the i5/OS Information Center provides information about how to plan for and configure printing functions, as well as basic printing information.

- | • The Basic printing topic provides specific information about printing elements and concepts of the System i product, printer file and print spooling support, and printer connectivity.
- | • The Security category in the i5/OS Information Center provides information about how to set up and plan for your system security, how to secure network and communications applications, and how to add highly secure cryptographic processing capability to your System i product. It also includes information about object signing and signature validation, identity mapping, and solutions to Internet security risks.
- | • *ILE C/C++ for AS/400 MI Library Reference*, SC09-2418-00 contains reference information for the C/C++ languages.
- | • *Security reference*, SC41-5302-10, tells how system security support can be used to protect the system and data from being used by people who do not have the proper authorization, protect data from intentional or unintentional damage or destruction, keep security information up-to-date, and set up security on the system.
- | • The Systems management category in the i5/OS Information Center provides information about the system unit control panel, starting and stopping the system, using tapes and diskettes, working with program temporary fixes, as well as handling problems.
- | • *WebSphere Development Studio: ILE C/C++ Language Reference*, SC09-7852-01, contains reference information for the C/C++ languages.
- | • *WebSphere Development Studio: ILE C/C++ Compiler Reference*, SC09-4816-04, contains reference information about using preprocessor statements, macros defined by and pragmas recognized by the ILE C/C++ compiler, command line options for both System i and QShell working environments, and I/O considerations for the System i environment.
- | • *WebSphere Development Studio: ILE C/C++ Programmer's Guide*, SC09-2712-06, provides information about how to develop applications using the ILE C language. It includes information about creating, running and debugging programs. It also includes programming considerations for interlanguage program and procedure calls, locales, handling exceptions, database, externally described and device files. Some performance tips are also

described. An appendix includes information about migrating source code from EPM C or System C to ILE C.

For more information about programming utilities, see the following books at the IBM Publications Center:

- *ADTS/400: Programming Development Manager*, SC09-1771-00
- *ADTS for AS/400: Screen Design Aid*, SC09-2604-00
- *ADTS for AS/400: Source Entry Utility*, SC09-2605-00

---

# Index

## Special characters

- `__EXBDY` built-in 6
- `__VBDY` built-in 6
- `_C_Get_Ssn_Handle()` function 55
- `_C_Quickpool_Debug()` function 66
  - Quick Pool Memory Management Debug 66
- `_C_Quickpool_Init()` function 68
  - Initialize Quick Pool Memory Management 68
- `_C_Quickpool_Report()` function 70
  - Generate Quick Pool Memory Management Report 70
- `_C_TS_malloc` 196, 551
- `_C_TS_malloc_debug` 551
- `_C_TS_malloc_debug` function 77
- `_C_TS_malloc_info` 551
- `_C_TS_malloc_info()` function 79
- `_C_TS_malloc64` 196, 551
- `__EXBDY` macro 6
- `_fputc()` function 120
- `_gcvt()` function 151
- `_GetExcData()` function 154
- `_INTRPT_Hndlr_Parms_T` 4
- `_itoa()` function 176
- `_ltoa()` function 192
- `_Racquire()` function 257
- `_Rclose()` function 258
- `_Rcommit()` function 259
- `_Rdelete()` function 261
- `_Rdevatr()` function 263
- `_Rfeod()` function 278
- `_Rfeov()` function 279
- `_Rformat()` function 280
- `_Rindara()` function 282
- `_Riofbk()` function 284
- `_Rlocate()` function 286
- `_Ropen()` function 289
- `_Ropnfbk()` function 293
- `_Rpgmdev()` function 294
- `_Rreadd()` function 295
- `_Rreadf()` function 297
- `_Rreadindv()` function 299
- `_Rreadk()` function 302
- `_Rreadl()` function 305
- `_Rreadn()` function 306
- `_Rreadnc()` function 308
- `_Rreadp()` function 310
- `_Rreads()` function 312
- `_Rrelease()` function 314
- `_Rrlsck()` function 316
- `_Rrollbck()` function 317
- `_Rupdate()` function 319
- `_Rupfb()` function 320
- `_Rwrite()` function 322
- `_Rwrited()` function 324
- `_Rwriterd()` function 327
- `_Rwrread()` function 328
- `_ultoa()` function 419
- `__VBDY` macro 6
- `_wcsicmp()` function 460

- `_wcsnicmp()` function 467

## A

- abnormal program end 36
- `abort()` function 36
- `abs()` function 37
- absolute value
  - `abs()` function 37
  - `fabs` 90
  - `labs` 177
- access mode 92, 109
- `acos()` function 38
- acquire a program device 257
- adding data to streams 92
- append mode
  - using `fopen()` function 109
- appending data to files 92
- arccosine 38
- arcsine 42
- arctangent 44
- argument list functions 30
- `asctime_r()` function 41
- `asctime()` function 39
- `asin()` function 42
- `assert.h` include file 3
- `assert()` function 43
- `atan()` function 44
- `atan2()` function 44
- `atexit()` function 45
- `atof()` function 46
- `atoi()` function 48
- `atol()` function 49
- `atoll()` function
  - strings to long long values 49

## B

- bessel functions 23, 50
- binary files 111
- binary search 51
- `blksize` 111
- block size 111
- `bsearch()` function 51
- `btowc()` function 53
- buffers
  - assigning 336
  - comparing 213
  - copying 214, 217
  - flushing 96
  - searching 212
  - setting characters 218
- `bufsiz` constant 16
- builtins
  - `__EXBDY` 6
  - `__VBDY` 6

## C

- calculating
  - absolute value 37
  - absolute value of long integer 177
  - arccosine 38
  - arctangent 44
  - base 10 logarithm 191
  - calculate the next representable floating-point value 223
  - cosine 64
  - error functions 87
  - exponential function 89
  - floating-point absolute value 90
  - floating-point remainder 108
  - hyperbolic cosine 65
  - hyperbolic sine 349
  - hypotenuse 168
  - logarithm 191
  - natural logarithm 191
  - quotient and remainder 86
  - sine 348
  - time difference 82, 84
- `calloc()` function 55
- cancel handler reason codes 514
- case mapping functions 34
- `catclose()` function 57
- `catgets()` function 58
- `catopen()` function 59
- `ceil()` function 61
- ceiling function 61
- changing
  - data area 247
  - environment variables 240
  - file position 134
  - reserved storage block size 264
- character
  - converting
    - to floating-point 49
    - to integer 48
    - to long integer 49
  - reading 98, 152
  - setting 218
  - ungetting 420
  - writing 118, 239
- character case mapping
  - `tolower` 416
  - `toupper` 416
  - `towlower` 418
  - `towupper` 418
- character testing
  - ASCII value 171
  - character property 172, 175
  - `isalnum` 169
  - `isalpha` 169
  - `iscntrl` 169
  - `isdigit` 169
  - `isgraph` 169
  - `islower` 169
  - `isprint` 169
  - `ispunct` 169
  - `isspace` 169

- character testing (*continued*)
  - isupper 169
  - isxdigit 169
  - wide alphabetic character 173
  - wide alphanumeric character 173
  - wide control character 173
  - wide decimal-digit character 173
  - wide hexadecimal digit 173
  - wide lowercase character 173
  - wide non-alphanumeric character 173
  - wide non-space character 173
  - wide printing character 173
  - wide uppercase character 173
  - wide whitespace character 173
- character testing functions 33
- clear error indicators 62
- clearerr 62
- clock() function 63
- CLOCKS\_PER\_SEC 63
- closing
  - file 258
  - message catalog 57
  - stream 91
- comparing
  - buffers 213
  - strings 360, 363, 365, 379
- comparing strings 357, 376
- compile regular expression 267
- concatenating strings 358, 377
- conversion functions
  - QXXDTOP 248
  - QXXDTOZ 249
  - QXXITOP() 250
  - QXXITOZ 250
  - QXXPTOD 251
  - QXXPTOI 252
  - QXXZTOD 254
  - QXXZTOI 255
- converting
  - character case 416
  - character string to decimal floating-point 395
  - character string to double 392
  - character string to long integer 400
  - date 385, 469
  - double to zoned decimal 249
  - floating-point numbers to integers and fractions 222
  - floating-point to packed decimal 248
  - from structure to string 39
  - from structure to string (restartable version) 41
  - integer to a character in the ASCII character set 415
  - integer to packed decimal 250
  - integer to zoned decimal 250
  - local time 218, 220
  - monetary value to string 368
  - multibyte character to a wide character 201
  - multibyte character to wchar\_t 211
  - multibyte string to a wide character string 206
  - packed decimal to double 251
  - packed decimal to integer 252
  - single byte to wide character 53

- converting (*continued*)
  - string segment to unsigned integer 403
  - string to formatted date and time 458
  - strings to floating-point values 46
  - strings to integer values 48
  - strings to long values 49
  - time 161, 163, 165, 167, 185, 187, 188, 189, 385, 469
  - time to character string 71, 73, 74, 76
  - wide character case 418
  - wide character string to multibyte string 473
  - wide character to a multibyte character 446
  - wide character to byte 491
  - wide character to long integer 481
  - wide character to multibyte character 492
  - wide-character string to decimal floating-point 478
  - wide-character string to double 476
  - wide-character string to unsigned long 486
  - zoned decimal to double 254
  - zoned decimal to integer 255

- copying
  - bytes 214, 217
  - strings 364, 380
- cos() function 64
- cosh() function 65
- creating
  - a temporary file 414
- ctime\_r() function 74
- ctime() function 71
- ctime64\_r() function 76
- ctime64() function 73
- ctype functions 169
- ctype.h include file 3
- currency functions 24

## D

- data conversion
  - atof() function 46
  - atoi() function 48
  - atol() function 49
- data items 126
- data type compatibility
  - CL 519, 521, 522
  - COBOL 520
  - ILE COBOL 518
  - RPG 516, 519
- data type limits 7
- date and time conversion 385, 469
- decimal.h include file 3
- deleting
  - file 274
  - record 261, 274
- determine the display width of a wide character 497
- determining
  - display width of a wide character string 489
  - display width of a wide-character string 490

- determining (*continued*)
  - length of a multibyte character 199
- differential equations 23
- difftime() function 82
- difftime64() function 84
- divf() function 86

## E

- end-of-file indicator 62, 95
- ending a program 36, 88
- environment
  - functions 32
  - interaction 32
  - retrieving information 181
  - table 154
  - variables 154, 240
- environment variables
  - adding 240
  - changing 240
  - searching 154
- eofile
  - clearing 276
  - macro 16
  - resetting error indicator 62
- erf() function 87
- erfc() function 87
- errno 4
- errno macros 507
- errno values for Integrated File System 508
- errno variable 227
- errno.h include file 4
- error handling
  - assert 43
  - clearerr 62
  - ferror 95
  - functions 21
  - perror 227
  - stream I/O 95
  - strerror 367
- error indicator 95
- error macros, mapping stream I/O exceptions 510
- error messages
  - printing 227
- except.h include file 4
- exception class
  - listing 515
  - mapping 513
- EXIT\_FAILURE 17, 88
- EXIT\_SUCCESS 17, 88
- exit() function 88
- exp() function 89
- exponential functions
  - exp 89
  - frexp 132
  - ldexp 178
  - log 191
  - log10 191
  - pow 228
  - sqrt 353

## F

- fabs() function 90
- fclose() function 91
- fdopen() function 92
- feof() function 95
- ferror() function 95
- fflush() function 96
- fgetc() function 98
- fgetpos() function 99
- fgets() function 101
- fgetwc() function 102
- fgetws() function 104
- file
  - appending to 92
  - handle 106
  - include 3
  - maximum opened 16
  - name length 16
  - positioning 276
  - renaming 275
  - updating 92
- file errors 62
- file handling
  - remove 274
  - rename 275
  - tmpnam 414
- file name length 16
- file names, temporary 16
- file positioning 99, 134, 136, 138
- FILE type 16
- fileno() function 106
- float.h include file 7
- floor() function 107
- flushing buffers 96
- fmod() function 108
- fopen, maximum simultaneous files 16
- fopen() function 109
- format data as wide characters 143
- formatted I/O 116
- fpos\_t 16
- fprintf() function 116
- fputc() function 118
- fputs() function 121
- fputwc() function 122
- fputws() function 124
- fread() function 126
- free() function 128
- freopen() function 130
- frexp() function 132
- fscanf() function 132
- fseek() function 134
- fseeko() function 134
- fsetpos() function 136
- ftell() function 138
- fwide() function 140
- fwprintf() function 143
- fwrite() function 146
- fwscanf() function 147

## G

- gamma() function 150
- getc() function 152
- getchar() function 152
- getenv() function 154
- gets() function 156

- getting
  - handle for character mapping 493
  - handle for character property classification 495
  - wide character from stdin 159
- getwc() function 157
- getwchar() function 159
- gmtime\_r() function 165
- gmtime() function 161
- gmtime64\_r() function 167
- gmtime64() function 163

## H

- handling interrupt signals 346
- HUGE\_VAL 8
- hypot() function 168
- hypotenuse 168

## I

- I/O errors 62
- idate
  - correcting for local time 185, 187, 188, 189
  - functions 24
- include files
  - assert.h 3
  - ctype.h 3
  - decimal.h 3
  - errno.h 4
  - except.h 4
  - float.h 7
  - inttypes.h 7
  - limits.h 7
  - locale.h 7
  - math.h 8
  - pointer.h 9
  - recio.h 9
  - regex.h 12
  - setjmp.h 13
  - signal.h 13
  - stdarg.h 13
  - stddef.h 13
  - stdint.h 14
  - stdio.h 15
  - stdlib.h 17
  - string.h 17
  - time.h 18
  - xxcvt.h 19
  - xxdta.h 19
  - xxenv.h 19
  - xxfdbk.h 19
- indicators, error 62
- initial strings 380
- integer
  - pseudo-random 256
- Integrated File System errno values 508
- internationalization 7
- interrupt signal 346
- inttypes.h include file 7
- invariant character
  - hexadecimal representation 523
- isalnum() function 169
- isalpha() function 169
- isascii() function 171

- isblank() function 172
- iscntrl() function 169
- isdigit() function 169
- isgraph() function 169
- islower() function 169
- isprint() function 169
- ispunct() function 169
- isspace() function 169
- isupper() function 169
- iswalnu() function 173
- iswcntrl() function 173
- iswctype() function 175
- iswdigit() function 173
- iswgraph() function 173
- iswlower() function 173
- iswprint() function 173
- iswpunct() function 173
- iswspace() function 173
- iswupper() function 173
- iswxdigit() function 173
- isxdigit() function 169

## L

- labs() function 177
- langinfo.h include file 7
- language collation string comparison 455
- ldexp() function 178
- ldiv() function 179
- length function 375
- length of variables 516
- library functions
  - absolute value
    - abs 37
    - fabs 90
    - labs 177
  - character case mapping
    - tolower 416
    - toupper 416
    - tolower 418
    - toupper 418
  - character testing
    - isalnum 169
    - isalpha 169
    - isascii 171
    - iscntrl 169
    - isdigit 169
    - isgraph 169
    - islower 169
    - isprint 169
    - ispunct 169
    - isspace 169
    - isupper 169
    - iswalnum 173
    - iswalph 173
    - iswcntrl 173
    - iswctype 175
    - iswdigit 173
    - iswgraph 173
    - iswlower 173
    - iswprint 173
    - iswpunct 173
    - iswspace 173
    - iswupper 173
    - iswxdigit 173
    - isxdigit 169

## library functions (continued)

conversion  
 QXXDTOP 248  
 QXXDTOZ 249  
 QXXITOP 250  
 QXXITOZ 250  
 QXXPTOD 251  
 QXXPTOI 252  
 QXXZTOD 254  
 QXXZTOI 255  
 strfmon 368  
 strptime 385  
 wcsftime 458  
 wcsptime 469

data areas  
 QXXCHGDA 247  
 QXXRTVDA 252

error handling  
 \_GetExcData 154  
 clearerr 62  
 raise 255  
 strerror 367

exponential  
 exp 89  
 frexp 132  
 ldexp 178  
 log 191  
 log10 191  
 pow 228

file handling  
 fileno 106  
 remove 274  
 rename 275  
 tmpfile 414  
 tmpnam 414

locale  
 localeconv 181  
 nl\_langinfo 224  
 setlocale 339  
 strxfrm 404

math  
 acos 38  
 asin 42  
 atan 44  
 atan2 44  
 bessell 50  
 ceil 61  
 cos 64  
 cosh 65  
 div 86  
 erf 87  
 erfc 87  
 floor 107  
 fmod 108  
 frexp 132  
 gamma 150  
 hypot 168  
 ldiv 179  
 log 191  
 log10 191  
 modf 222  
 sin 348  
 sinh 349  
 sqrt 353  
 tan 409  
 tanh 410

## library functions (continued)

memory management  
 \_C\_TS\_malloc\_debug 77  
 \_C\_TS\_malloc\_info 79  
 calloc 55  
 free 128  
 malloc 195  
 realloc 264

memory operations  
 memchr 212  
 memcmp 213  
 memcpy 214  
 memmove 217  
 memset 218  
 wmemchr 498  
 wmemcmp 499  
 wmemcpy 500  
 wmemmove 501  
 wmemset 502

message catalog  
 catclose 57  
 catgets 58  
 catopen 59

miscellaneous  
 assert 43  
 getenv 154  
 longjmp 193  
 perror 227  
 putenv 240  
 rand 256  
 rand\_r 256  
 setjmp 338  
 srand 354

multibyte  
 \_wcsicmp 460  
 \_wcsnicmp 467  
 btowc 53  
 mblen 197  
 mbrlen 199  
 mbrtowc 201  
 mbsinit 205  
 mbsrtowcs 206  
 mbstowcs 207  
 mbtowc 211  
 towctrans 417  
 wctomb 446  
 wscat 451  
 wcschr 452  
 wscmp 453  
 wscoll 455  
 wcsncpy 456  
 wcsncpy 457  
 wcslen 461  
 wcslocaleconv 462  
 wcsncat 463  
 wcsncmp 464  
 wcsncpy 466  
 wcsprbrk 468  
 wcsrchr 471  
 wcsrtombs 473  
 wcssp 474  
 wcstombs 483  
 wcs wcs 488  
 wcs width 489  
 wcsxfrm 490  
 wctob 491  
 wctomb 492

## library functions (continued)

multibyte (continued)  
 wctrans 493  
 wctype 495  
 wcwidth 497

program  
 abort 36  
 atexit 45  
 exit 88  
 signal 346

regular expression  
 regcomp 267  
 regerror 269  
 regexec 271  
 regfree 273

searching  
 bsearch 51  
 qsort 245

stream input/output  
 fclose 91  
 feof 95  
 ferror 95  
 fflush 96  
 fgetc 98  
 fgetpos 99  
 fgets 101  
 fgetwc 102  
 fgetws 104  
 fprintf 116  
 fputc 118  
 fputs 121  
 fputwc 122  
 fputws 124  
 fread 126  
 freopen 130  
 fscanf 132  
 fseek 134  
 fsetpos 136  
 ftell 138  
 fwide 140  
 fwprintf 143  
 fwrite 146  
 fwscanf 147  
 getc 152  
 getchar 152  
 gets 156  
 getwc 157  
 getwchar 159  
 printf 229  
 putc 239  
 putchar 239  
 puts 241  
 putwc 242  
 putwchar 244  
 scanf 330  
 setbuf 336  
 setvbuf 344  
 sprintf 352  
 sscanf 355  
 swprintf 406  
 swscanf 407  
 ungetc 420  
 ungetwc 422  
 vfprintf 425  
 vfscanf 427  
 vfwprintf 428  
 vfwscanf 430

library functions (*continued*)

stream input/output (*continued*)

vprintf 432  
 vscanf 433  
 vsnprintf 435  
 vsprintf 436  
 vsscanf 437  
 vswprintf 439  
 vswscanf 441  
 vwprintf 443  
 vwscanf 445  
 wprintf 503  
 wscanf 504

string manipulation

strcat 358  
 strchr 359  
 strcmp 360  
 strcoll 363  
 strcpy 364  
 strcspn 365  
 strlen 375  
 strncmp 379  
 strncpy 380  
 strpbrk 384  
 strrchr 389  
 strspn 390  
 strstr 391  
 strtod 392  
 strtok 398  
 strtok\_r 399  
 strtol 400  
 strtoul 403  
 strxfrm 404  
 wcsstr 475  
 wctok 480

time

asctime 39  
 asctime\_r 41  
 clock 63  
 ctime 71  
 ctime\_r 74  
 ctime64 73  
 ctime64\_r 76  
 difftime 82  
 difftime64 84  
 gmtime 161  
 gmtime\_r 165  
 gmtime64 163  
 gmtime64\_r 167  
 localtime 185  
 localtime\_r 188  
 localtime64 187  
 localtime64\_r 189  
 mktime 218  
 mktime64 220  
 strftime 370  
 wcsftime 458

trigonometric

acos 38  
 asin 42  
 atan 44  
 atan2 44  
 cos 64  
 cosh 65  
 sin 348  
 sinh 349  
 tan 409

library functions (*continued*)

trigonometric (*continued*)

tanh 410

type conversion

atof 46  
 atoi 48  
 atol 49  
 strol 400  
 strtod 392  
 strtoul 403  
 toascii 415  
 wcstod 476  
 wcstol 481  
 wcstoul 486

variable argument handling

va\_arg 423  
 va\_end 423  
 va\_start 423  
 vfprintf 425  
 vfscanf 427  
 vfwscanf 430  
 vprintf 432  
 vscanf 433  
 vsnprintf 435  
 vsprintf 436  
 vsscanf 437  
 vswscanf 441  
 vwscanf 445

library introduction 21

limits.h include file 7

llabs() subroutine

absolute value of long long  
 integer 177

lldiv() subroutine

perform long long division 179

local time corrections 185, 187

local time corrections (restartable  
 version) 188, 189

locale functions

localeconv 181  
 setlocale 339  
 strxfrm 404

locale.h include file 7

localeconv() function 181

locales

retrieve information 224  
 setting 339

localtime\_r() function 188

localtime() function 185

localtime64\_r() function 189

localtime64() function 187

locating storage 128

log() function 191

log10() function 191

logarithmic functions

log 191  
 log10 191

logic errors 43

logical record length 111

longjmp() function 193

recl 111

## M

malloc() function 195

math functions

abs 37

math functions (*continued*)

acos 38  
 asin 42  
 atan 44  
 atan2 44  
 bessel 50  
 div 86  
 erf 87  
 erfc 87  
 exp 89  
 fabs 90  
 floor 107  
 fmod 108  
 frexp 132  
 gamma 150  
 hypot 168  
 labs 177  
 ldexp 178  
 ldiv 179  
 log 191  
 log10 191  
 modf 222  
 pow 228  
 sin 348  
 sinh 349  
 sqrt 353  
 tan 409  
 tanh 410

math.h include file 8

mathematical functions 22

maximum

file name 16  
 opened files 16  
 temporary file name 16

MB\_CUR\_MAX 17

mblen() function 197

mbrlen() function 199

mbrtowc() function 201

mbsinit() function 205

mbsrtowcs() function 206

mbstowcs() function 207

mbtowc() function 211

memchr() function 212

memcmp() function 213

memcpy() function 214

memicmp() function 215

memmove() function 217

memory allocation

\_C\_TS\_malloc\_debug 77  
 \_C\_TS\_malloc\_info 79  
 calloc 55  
 free 128  
 malloc 195  
 realloc 264

memory management

\_C\_TS\_malloc\_debug 77  
 \_C\_TS\_malloc\_info 79  
 calloc 55  
 free 128  
 malloc 195  
 realloc 264

memory object functions 31

memory operations

memchr 212  
 memcmp 213  
 memcpy 214  
 memmove 217

memory operations (*continued*)

- memset 218
- wmemchr 498
- wmemcmp 499
- wmemcpy 500
- wmemmove 501
- wmemset 502

memset() function 218

miscellaneous functions

- assert 43
- getenv 154
- longjmp 193
- perror 227
- putenv 240
- rand 256
- rand\_r 256
- setjmp 338
- srand 354

mktime() function 218

mktime64() function 220

modf() function 222

monetary functions 24

monetary.h include file 8

multibyte functions

- \_wscicmp 460
- \_wcsnicmp 467
- btowc 53
- mblen 197
- mbrlen 199
- mbrtowc 201
- mbsinit 205
- mbsrtowcs 206
- mbstowcs 207
- mbtowc 211
- towctrans 417
- wcrtomb 446
- wscat 451
- wcschr 452
- wscmp 453
- wscoll 455
- wscpy 456
- wscspn 457
- wscicmp 460
- wcslen 461
- wcslocaleconv 462
- wcsncat 463
- wcsncmp 464
- wcsncpy 466
- wcsnicmp 467
- wcspbrk 468
- wcsrchr 471
- wcsrtombs 473
- wcsspn 474
- wcstombs 483
- wcswcs 488
- wcswidth 489
- wcsxfrm 490
- wctob 491
- wctomb 492
- wctrans 493
- wctype 495
- wcwidth 497

## N

NDEBUG 3, 43

nextafter() function 223

nextafterl() function 223

nexttoward() function 223

nexttowardl() function 223

nl\_langinfo() function 224

nltypes.h include file 9

nonlocal goto 193, 338

NULL pointer 14, 16, 17

## O

offsetof macro 14

opening

- message catalog 59

## P

passing

- constants 522

- variables 522

perror() function 227

pointer.h include file 9

pow() function 228

printf() function 229

printing

- error messages 227

process control

- signal 346

program termination

- abort 36

- atexit 45

- exit 88

pseudo-random integers 256

pseudorandom number functions

- rand 256

- rand\_r 256

- srand 354

ptrdiff\_t 13

pushing characters 420

putc() function 239

putchar() function 239

putenv() function 240

puts() function 241

putwc() function 242

putwchar() function 244

## Q

qsort() function 245

quick sort 245

QXXCHGDA() function 247

QXXDTOP() function 248

QXXDTOZ() function 249

QXXITOP() function 250

QXXITOZ() function 250

QXXPTOD() function 251

QXXPTOI() function 252

QXXRTVDA() function 252

QXXZTOD() function 254

QXXZTOI() function 255

## R

raise() function 255

RAND\_MAX 17

rand\_r() function 256

rand() function 256

random access 134, 138

random number generator 256, 354

read operations

- character from stdin 152

- character from stream 152

- data items from stream 126

- formatted 132, 330, 355

- line from stdin 156

- line from stream 101

- reading a character 98

- scanning 132

reading

- character 152

- data 330

- data from stream using wide

  - character 147

  - data using wide-character format

    - string 504

  - formatted data 132

  - items 126

  - line 156

  - messages 58

  - stream 101

  - wide character from stream 102, 157

  - wide-character string from

    - stream 104

realloc() function 264

reallocation 264

recfm 111

recio.h include file 9

record format 111

record input/output

- \_Racquire 257

- \_Rclose 258

- \_Rcommit 259

- \_Rdelete 261

- \_Rdevatr 263

- \_Rfeed 278

- \_Rfeov 279

- \_Rformat 280

- \_Rindara 282

- \_Riofbk 284

- \_Rlocate 286

- \_Ropen 289

- \_Ropnfbk 293

- \_Rpgmdev 294

- \_Rreadd 295

- \_Rreadf 297

- \_Rreadindv 299

- \_Rreadk 302

- \_Rreadl 305

- \_Rreadn 306

- \_Rreadnc 308

- \_Rreadp 310

- \_Rreads 312

- \_Rrelease 314

- \_Rrlsck 316

- \_Rrollbck 317

- \_Rupdate 319

- \_Rupfb 320

- \_Rwrite 322

- \_Rwrited 324

- \_Rwriterd 327

- \_Rwrread 328

record program ending 45

redirection 130

- regcomp() function 267
- regerror() function 269
- regex.h include file 12
- regexec() function 271
- regfree() function 273
- remove() function 274
- rename() function 275
- reopening streams 130
- reserving storage
  - \_C\_TS\_malloc\_debug 77
  - \_C\_TS\_malloc\_info 79
  - malloc 195
  - realloc 264
- retrieve data area 252
- retrieve locale information 224
- rewind() function 276

## S

- scanf() function 330
- searching
  - bsearch function 51
  - environment variables 154
  - strings 359, 384, 390
  - strings for tokens 398, 399
- searching and sorting functions 22
- seed 354
- send signal 255
- separate floating-point value 132
- setbuf() function 336
- setjmp.h include file 13
- setjmp() function 338
- setlocale() function 339
- setting
  - bytes to value 218
- setvbuf() function 344
- signal handling 511
- signal.h include file 13
- signal() function 346
- sin() function 348
- sine 348
- sinh() function 349
- size\_t 13
- snprintf() function 350
- sorting
  - quick sort 245
- sprintf() function 352
- sqrt() function 353
- srand() function 354
- sscanf() function 355
- standard types
  - FILE 16
- stdarg.h include file 13
- stddef.h include file 13
- stdint.h include file 14
- stdio.h include file 15
- stdlib.h include file 17
- stopping
  - program 36
- storage allocation 55
- strcasecmp() function 357
- strcat() function 358
- strchr() function 359
- strcmp() function 360
- strcmpi() function 362
- strcoll() function 363
- strcpy() function 364

- strcsprn() function 365
- strdup() function 366
- stream I/O functions 27
- stream input/output
  - fclose 91
  - feof 95
  - ferror 95
  - fflush 96
  - fgetc 98
  - fgets 101
  - fopen 109
  - fprintf 116
  - fputc 118
  - fputs 121
  - fputwc 122
  - fputws 124
  - fread 126
  - freopen 130
  - fscanf 132
  - fseek 134
  - ftell 138
  - fwrite 146
  - getc 152
  - getchar 152
  - gets 156
  - printf 229
  - putc 239
  - putchar 239
  - puts 241
  - rewind 276
  - scanf 330
  - setbuf 336
  - setvbuf 344
  - snprintf 350
  - sprintf 352
  - sscanf 355
  - swprintf 406
  - swscanf 407
  - tmpfile 414
  - ungetc 420
  - ungetwc 422
  - va\_arg 423
  - va\_end 423
  - va\_start 423
  - vfprintf 425
  - vfprintf 427
  - vfwprintf 428
  - vfwscanf 430
  - vprintf 432
  - vscanf 433
  - vsnprintf 435
  - vsprintf 436
  - vsscanf 437
  - vswprintf 439
  - vswscanf 441
  - vwprintf 443
  - vwscanf 445
  - wprintf 503
  - wscanf 504
- stream orientation 140
- streams
  - access mode 130
  - appending 109, 130
  - binary mode 130
  - buffering 336
  - changing current file position 134, 138

- streams (*continued*)
  - changing file position 276
  - formatted I/O 132, 229, 330, 352, 355
  - opening 109
  - reading characters 98, 152
  - reading data items 126
  - reading lines 101, 156
  - reopening 130
  - rewinding 276
  - text mode 130
  - translation mode 130
  - ungetting characters 420
  - updating 109, 130
  - writing characters 118, 239
  - writing data items 146
  - writing lines 241
  - writing strings 121
- strerror() function 367
- strfmon() function 368
- strftime() function 370
- stricmp() function 374
- string manipulation
  - strcasecmp 357
  - strcat 358
  - strchr 359
  - strcmp 360
  - strcoll 363
  - strcpy 364
  - strcsprn 365
  - strlen 375
  - strncasecmp 376
  - strncat 377
  - strncmp 379
  - strncpy 380
  - strpbrk 384
  - strrchr 389
  - strspn 390
  - strstr 391
  - strtod 392
  - strtok 398
  - strtok\_r 399
  - strtol 400
  - strxfrm 404
  - wcsstr 475
  - wcstok 480
- string.h include file 17
- strings
  - comparing 365, 379
  - concatenating 358
  - converting
    - to floating-point 49
    - to integer 48
    - to long integer 49
  - copying 364
  - ignoring case 360, 363, 365
  - initializing 380
  - length of 375
  - reading 101
  - searching 359, 384, 390
  - searching for tokens 398, 399
  - strstr 391
  - writing 121
- strlen() function 375
- strncasecmp() function 376
- strncat() function 377
- strncmp() function 379
- strncpy() function 380

strnicmp() function 382  
 strnset() function 383  
 strpbrk() function 384  
 strptime() function 385  
 strrchr() function 389  
 strset() function 383  
 strspn() function 390  
 strstr() function 391  
 strtod() function 392  
 strtod128() function 395  
 strtod32() function 395  
 strtod64() function 395  
 strtok\_r() function 399  
 strtok() function 398  
 strtol() function 400  
 strtoll() subroutine  
     character string to long long  
     integer 400  
 strtoul() function 403  
 strtoull() subroutine  
     character string to unsigned long long  
     integer 403  
 strxfrm() function 404  
 swprintf() function 406  
 swscanf() function 407  
 system() function 408

## T

tan() function 409  
 tangent 409  
 tanh() function 410  
 testing  
     ASCII value 171  
     character property 172, 175  
     isalnum 169  
     isalpha 169  
     iscntrl 169  
     isdigit 169  
     isgraph 169  
     islower 169  
     isprint 169  
     ispunct 169  
     isspace 169  
     isupper 169  
     isxdigit 169  
     state object for initial state 205  
     wide alphabetic character 173  
     wide alphanumeric character 173  
     wide control character 173  
     wide decimal-digit character 173  
     wide hexadecimal digit 173  
     wide lowercase character 173  
     wide non-alphanumeric  
         character 173  
     wide non-space character 173  
     wide printing character 173  
     wide uppercase character 173  
     wide whitespace character 173  
 testing state object for initial state 205  
 time  
     asctime 39  
     asctime\_r 41  
     converting from structure to  
         string 39  
     converting from structure to string  
         (restartable version) 41

time (*continued*)  
     correcting for local time 185, 187,  
         188, 189  
     ctime 71  
     ctime\_r 74  
     ctime64 73  
     ctime64\_r 76  
     difftime 82  
     difftime64 84  
     function 185, 187, 188, 189  
     functions 24  
     gmtime 161  
     gmtime\_r 165  
     gmtime64 163  
     gmtime64\_r 167  
     localtime 185  
     localtime\_r 188  
     localtime64 187  
     localtime64\_r 189  
     mktime 218  
     mktime64 220  
     strftime 370  
     time 411  
     time64 412  
 time.h include file 18  
 time() function 411  
 time64() function 412  
 tm structure 161, 163, 165, 167  
 TMP\_MAX 414  
 tmpfile() function  
     names 16  
     number of 16  
 tmpnam() function  
     file names 16  
     tmpnam() 414  
 toascii() function 415  
 tokens  
     strtok 398  
     strtok\_r 399  
     tokenize string 398  
 tolower() function 416  
 toupper() function 416  
 towctrans() function 417  
 towlower() function 418  
 toupper() function 418  
 trigonometric functions  
     acos 38  
     asin 42  
     atan 44  
     atan2 44  
     cos 64  
     cosh 65  
     sin 348  
     sinh 349  
     tan 409  
     tanh 410  
 type conversion  
     atof 46  
     atoi 48  
     atol 49  
     strtod 392  
     strtoul 400  
     strtoul 403  
     toascii 415  
     wcstod 476  
     wcstol 481  
     wcstoul 486

## U

ungetc() function 420  
 ungetwc() function 422  
 updating files 92

## V

va\_arg() function 423  
 va\_end() function 423  
 va\_start() function 423  
 variable argument functions 30  
 verify condition 43  
 vfprintf() function 425  
 vfscanf() function 427  
 vfwprintf() function 428  
 vfwscanf() function 430  
 vprintf() function 432  
 vscanf() function 433  
 vsnprintf() function 435  
 vsprintf() function 436  
 vsscanf() function 437  
 vswprintf() function 439  
 vswscanf() function 441  
 vwprintf() function 443  
 vwscanf() function 445

## W

wchar.h include file 18  
 wctomb() function 446  
 wcscat() function 451  
 wcschr() function 452  
 wcscmp() function 453  
 wcscoll() function 455  
 wcsncpy() function 456  
 wcsncpy() function 457  
 wcsftime() function 458  
 wcslen() function 461  
 wcslocaleconv() function 462  
 wcsncat() function 463  
 wcsncmp() function 464  
 wcsncpy() function 466  
 wcsprk() function 468  
 wcsptime() function 469  
 wcsrchr() function 471  
 wcsrtombs() function 473  
 wcssp() function 474  
 wcsstr() function 475  
 wcstod() function 476  
 wcstod128() function 478  
 wcstod32() function 478  
 wcstod64() function 478  
 wcstok() function 480  
 wcstol() function 481  
 wcstoll() subroutine  
     wide character to long long  
     integer 481  
 wcstombs() function 483  
 wcstoul() function 486  
 wcstoull() subroutine  
     wide-character string to unsigned long  
     long 486  
 wcs wcs() function 488  
 wcswidth() function 489  
 wcsxfrm() function 490  
 wctob() function 491

- wctomb() function 492
- wctrans() function 493
- wctype.h include file 19
- wctype() function 495
- wcwidth() function 497
- wide character string functions 34
- wmemchr() function 498
- wmemcmp() function 499
- wmemcpy() function 500
- wmemmove() function 501
- wmemset() function 502
- wprintf() function 503
- write operations
  - character to stdout 118, 239
  - character to stream 118, 239, 420
  - data items from stream 146
  - formatted 116, 229, 352
  - line to stream 241
  - printing 146
  - string to stream 121
- writing
  - character 118, 239
  - data items from stream 146
  - formatted data to a stream 116
  - string 121, 241
  - wide character 122, 242, 244
  - wide characters to a stream 143
  - wide-character string 124
- wscanf() function 504

## X

- xxcvt.h include file 19
- xxdta.h include file 19
- xxenv.h include file 19
- xxfdbk.h include file 19



---

## Readers' Comments — We'd Like to Hear from You

System i  
ILE C/C++ Runtime Library Functions  
Version 6 Release 1

Publication No. SC41-5607-03

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Submit your comments using one of these channels:

- Send your comments to the address on the reverse side of this form.
- Send a fax to the following number: United States and Canada: 1-800-937-3430  
Other countries or regions: 1-507-253-5192
- Send your comments via e-mail to: [RCHCLERK@us.ibm.com](mailto:RCHCLERK@us.ibm.com)

If you would like a response from IBM, please fill in the following information:

\_\_\_\_\_

Name

\_\_\_\_\_

Address

\_\_\_\_\_

Company or Organization

\_\_\_\_\_

Phone No.

\_\_\_\_\_

E-mail address



Fold and Tape

Please do not staple

Fold and Tape



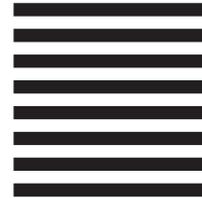
NO POSTAGE  
NECESSARY  
IF MAILED IN THE  
UNITED STATES

# BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM CORPORATION  
ATTN DEPT 542 IDCLERK  
3605 HWY 52 N  
ROCHESTER MN 55901-7829



Fold and Tape

Please do not staple

Fold and Tape





Printed in USA

SC41-5607-03

