

z/OS



Metal C Programming Guide and Reference

z/OS



Metal C Programming Guide and Reference

Note

Before using this information and the product it supports, be sure to read the general information under "Notices" on page 113.

Third Edition, September 2009

This is a major revision of SA23-2225-01.

This edition applies to Metal C Runtime Library in Version 1 Release 11 of z/OS (5694-A01) and to all subsequent releases and modifications until otherwise indicated in new editions.

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

International Business Machines Corporation
MHVRCFS, Mail Station P181
2455 South Road
Poughkeepsie, NY 12601-5400
United States of America

FAX (United States & Canada): 1+845+432-9405

FAX (Other Countries):

Your International Access Code +1+845+432-9405

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

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

World Wide Web: <http://www.ibm.com/systems/z/os/zos/webqs.html>

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

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

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

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

© Copyright International Business Machines Corporation 2007, 2009.

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

Contents

Figures.	xi
Tables.	xiii
About this document	xv
How to read syntax diagrams.	xv
Symbols	xv
Syntax items	xv
Syntax examples	xvi
Softcopy documents.	xvii
z/OS Metal C on the World Wide Web	xvii
Where to find more information	xvii
Summary of changes.	xix
Chapter 1. About IBM z/OS Metal C	1
Metal C environments	1
Programming with Metal C	2
Metal C and MVS linkage conventions.	2
Compiler-generated HLASM source code	4
Prolog and epilog code	8
Supplying your own HLASM statements.	13
Inserting HLASM instructions into the generated source code.	13
AMODE-switching support.	23
AR-mode programming support.	24
Building Metal C programs	33
Summary of useful references for the Metal C programmer	39
Chapter 2. Header files	41
builtin.h	41
ctype.h	41
float.h	41
inttypes.h	42
limits.h	44
math.h	45
metal.h	46
stdarg.h	46
stddef.h	46
stdio.h	46
Macros defined in stdio.h	46
stdint.h	47
Integer types.	47
stdlib.h	48
string.h	49
Chapter 3. C functions available to Metal C programs	51
Characteristics of Metal C runtime library functions	51
System and static object libraries	51
System library	51
Static object library	52
General library usage notes	52
abs() — Calculate integer absolute value	53
Format	53

General description	53
Returned value	53
Related Information	53
atoi() — Convert character string to integer	53
Format	53
General description	53
Returned value	53
Related Information	53
atol() — Convert character string to long	54
Format	54
General description	54
Returned value	54
Related Information	54
atoll() — Convert character string to signed long long.	54
Format	54
General description	54
Returned value	54
Related Information	54
calloc() — Reserve and initialize storage	55
Format	55
General description	55
Returned value	55
Related Information	55
__cinit() - Initialize a Metal C environment	55
Format	55
General description	55
Returned value	57
Example	57
__cterm() - Terminate a Metal C environment.	58
Format	58
General description	58
Returned value	58
Example	59
div() — Calculate quotient and remainder	59
Format	59
General description	59
Returned value	59
Related Information	59
free() — Free a block of storage	59
Format	59
General description	59
Returned value	60
Related Information	60
isalnum() to isxdigit() — Test integer value.	60
Format	60
General description	60
Returned value	61
Related Information	61
isalpha() — Test for alphabetic character classification	62
isblank() — Test for blank character classification	62
iscntrl() — Test for control classification	62
isdigit() — Test for decimal-digit classification.	62
isgraph() — Test for graphic classification	62
islower() — Test for lowercase	62
isprint() — Test for printable character classification	62
ispunct() — Test for punctuation classification	62

isspace()	— Test for space character classification	62
isupper()	— Test for uppercase letter classification	62
isxdigit()	— Test for hexadecimal digit Classification	62
labs()	— Calculate long absolute value	63
Format		63
General description		63
Returned value		63
Related Information		63
ldiv()	— Compute quotient and remainder of integral division	63
Format		63
General description		63
Returned value		63
Related Information		63
llabs()	— Calculate absolute value of long long integer	64
Format		64
General description		64
Returned value		64
Related Information		64
lldiv()	— Compute quotient and remainder of integral division for long long type	64
Format		64
General description		64
Returned value		64
Related Information		65
malloc()	— Reserve storage block	65
Format		65
General description		65
Returned value		65
Related Information		65
__malloc31()	— Allocate 31-bit storage	65
Format		65
General description		65
Returned value		66
Related Information		66
memcpy()	— Copy bytes in memory	66
Format		66
General description		66
Returned value		66
Related Information		66
memchr()	— Search buffer	66
Format		66
General description		66
Returned value		67
Related Information		67
memcmp()	— Compare bytes	67
Format		67
General description		67
Returned value		67
Related Information		67
memcpy()	— Copy buffer	68
Format		68
General description		68
Returned value		68
Related Information		68
memmove()	— Move buffer	68
Format		68
General description		68

Returned value	68
Related Information	68
memset() — Set buffer to value.	69
Format	69
General description	69
Returned value	69
Related Information	69
rand() — Generate random number	69
Format	69
General Description	69
Returned Value	69
Related Information	69
rand_r() — Pseudo-random number generator	69
Format	69
General Description	70
Returned Value	70
Related Information	70
realloc() — Change reserved storage block size.	70
Format	70
General Description	70
Returned Value	71
Related Information	71
snprintf() — Format and write data	71
Format	71
General Description	71
Returned Value	71
Related Information	71
sprintf() — Format and Write Data	72
Format	72
General Description	72
Returned Value	77
Related Information	78
srand() — Set Seed for rand() Function	78
Format	78
General Description	78
Returned Value	78
Related Information	78
sscanf() — Read and Format Data	78
Format	78
General Description	78
Returned Value	83
Related Information	83
strcat() — Concatenate Strings	83
Format	83
General Description	83
Returned Value	84
Related Information	84
strchr() — Search for Character.	84
Format	84
General Description	84
Returned Value	84
Related Information	84
strcmp() — Compare Strings	85
Format	85
General Description	85
Returned Value	85

Related Information	85
strcpy() — Copy String	85
Format	85
General Description	85
Returned Value	86
Related Information	86
strcspn() — Compare Strings	86
Format	86
General Description	86
Returned Value	86
Related Information	86
strdup() — Duplicate a String	86
Format	86
General Description	86
Returned Value	87
Related Information	87
strlen() — Determine String Length	87
Format	87
General Description	87
Returned Value	87
Related Information	87
strncat() — Concatenate Strings	87
Format	87
General Description	87
Returned Value	88
Related Information	88
strncmp() — Compare Strings	88
Format	88
General Description	88
Returned Value	88
Related Information	88
strncpy() — Copy String	89
Format	89
General Description	89
Returned Value	89
Related Information	89
strpbrk() — Find Characters in String	89
Format	89
General Description	89
Returned Value	89
Related Information	89
strrchr() — Find Last Occurrence of Character in String	90
Format	90
General Description	90
Returned Value	90
Related Information	90
strspn() — Search String	90
Format	90
General Description	90
Returned Value	90
Related Information	90
strstr() — Locate Substring	91
Format	91
General Description	91
Returned Value	91
Related Information	91

	strtod — Convert Character String to Double	91
	Format	91
	General Description	91
	Returned Value	92
	Related Information	92
	strtof — Convert Character String to Float	92
	Format	92
	General Description	92
	Returned Value	93
	Related Information	93
	strtok() — Tokenize String	94
	Format	94
	General Description	94
	Returned Value	94
	Related Information	94
	strtok_r() — Split String into Tokens	95
	Format	95
	General Description	95
	Returned Value	95
	Related Information	95
	strtol() — Convert Character String to Long	95
	Format	95
	General Description	96
	Returned Value	96
	Related Information	96
	strtold — Convert Character String to Long Double	97
	Format	97
	General Description	97
	Returned Value	97
	Related Information	98
	strtoll() — Convert String to Signed Long Long	98
	Format	98
	General Description	98
	Returned Value	99
	Related Information	99
	strtoul() — Convert String to Unsigned Integer	99
	Format	99
	General Description	99
	Returned Value	100
	Related Information	100
	strtoull() — Convert String to Unsigned Long Long	100
	Format	100
	General Description	101
	Returned Value	101
	Related Information	102
	tolower(), toupper() — Convert Character Case	102
	Format	102
	General Description	102
	Returned Value	102
	Related Information	102
	va_arg(), va_copy(), va_end(), va_start() — Access Function Arguments	102
	Format	102
	General Description	103
	Returned Value	103
	Related Information	103
	vsprintf() — Format and print data to fixed length buffer	104

Format	104
General Description.	104
Returned Value	104
Related Information.	104
vsprintf() — Format and Print Data to Buffer	104
Format	104
General Description.	105
Returned Value	105
Related Information.	105
vsscanf() — Format Input of a STDARG Argument List.	105
Format	105
General Description.	105
Returned Value	106
Related Information.	106
Appendix A. Function stack requirements	107
Appendix B. Accessibility	111
Using assistive technologies	111
Keyboard navigation of the user interface.	111
z/OS information	111
Notices	113
Policy for unsupported hardware	115
Programming Interface Information	115
Trademarks.	115
Standards	115
Index	117

Figures

1. Structure of prefix data	7
2. Generation of an optional section in the prefix data.	7
3. Prefix data with optional sections	8
4. Specification of your own prolog and epilog code for a function	9
5. SCCNSAM(MYPROLOG).	11
6. SCCNSAM(MYEPILOG)	12
7. Simple code format string	14
8. Code format string with two instructions	14
9. Code format string with two instructions, formatted for readability	14
10. Substitution of a C variable into an output <code>__asm</code> operand	15
11. HLASM source code embedded by the <code>__asm</code> statement in Figure 10 on page 15.	15
12. Substitution of a C pointer into an <code>__asm</code> operand	16
13. <code>__asm</code> operand lists.	16
14. Example of compiler-generated HLASM code for the <code>__asm</code> statement in Figure 13 on page 16	17
15. Unsuccessful attempt to specify registers	17
16. Register specification with clobbers	18
17. Unsuccessful attempt to define an <code>__asm</code> operand for both input and output	18
18. Incorrect HLASM source code from Figure 17 on page 18.	18
19. Successful definition of an <code>__asm</code> operand for both input and output.	19
20. Correct HLASM source code output from Figure 19 on page 19	19
21. The + constraint to define an <code>__asm</code> operand for both input and output.	19
22. Error: Redundant definition of an <code>__asm</code> operand	20
23. Specifying and using the WTO macro (no reentrancy)	21
24. Support for reentrancy in a code format string	21
25. Code that supplies specific DSECT mapping macros	22
26. Register specification	23
27. AMODE31 program that calls an AMODE64 program	24
28. Far pointer sizes under different addressing modes	25
29. Built-in functions for setting far-pointer components	27
30. Built-in functions for getting far-pointer components	27
31. Library functions for use only in AR-mode functions	27
32. Allocation and deallocation routines	29
33. Copying a C string pointer to a far pointer	31
34. Example of a simple dereference of a far pointer	32
35. Metal C application build process.	33
36. C source file (mycode.c) that builds a Metal C program	34
37. C compiler invocation to generate mycode.s.	34
38. C compiler invocation to generate mycode.s with debugging comments.	34
39. Command that invokes HLASM to assemble mycode.s.	35
40. Command that compiles an HLASM source file containing symbols longer than eight characters	35
41. Command that binds mycode.o and produces a Metal C program in an MVS data set	35
42. Commands that compile and link programs with different addressing modes	36
43. Job step that compiles HLQ.SOURCE.C(MYCODE)	36
44. Job step that invokes the CDAASMC JCL procedure	37
45. Job step that binds the generated HLASM object into a program	37
46. JCL that invokes the ASMLANGX utility	38

Tables

	1. Syntax examples	xvi
	2. Compiler-generated global SET symbols	10
I	3. User modifiable global SET symbols	10
	4. Language constructs that may have special impact on far pointers	25
	5. Implicit ALET associations for AR-mode-function variables	26
	6. Summary of useful references for the Metal C programmer	39
I	7. Definitions in float.h	41
	8. Definitions of Resource Limits	44
	9. csysenv argument in __cinit()	56
	10. csysenvtkn argument in __cterm()	58
	11. Flag Characters for sprintf() Family	73
	12. Precision Argument in sprintf()	75
	13. Type Characters and their Meanings	76
	14. Conversion Specifiers in sscanf()	81
	15. Stack frame requirements for Metal C runtime functions	107

About this document

The z/OS Metal C Runtime Library is an element as of z/OS V1R10 that provides a runtime library to make use of the z/OS XL C METAL compiler option.

This document provides information about a set of C header files and functions provided by Metal C Runtime Library. It contains advanced guidelines and information for developing Metal C programs.

For more information on the XL C METAL compiler option, see *z/OS XL C/C++ User's Guide*.

For more information on the Metal C Runtime Library, see <http://www.ibm.com/systems/z/zos/metalc/>.

How to read syntax diagrams

This section describes how to read syntax diagrams. It defines syntax diagram symbols, items that may be contained within the diagrams (keywords, variables, delimiters, operators, fragment references, operands) and provides syntax examples that contain these items.

Syntax diagrams pictorially display the order and parts (options and arguments) that comprise a command statement. They are read from left to right and from top to bottom, following the main path of the horizontal line.

Symbols

The following symbols may be displayed in syntax diagrams:

Symbol	Definition
▶▶—	Indicates the beginning of the syntax diagram.
—▶	Indicates that the syntax diagram is continued to the next line.
▶—	Indicates that the syntax is continued from the previous line.
—▶◀	Indicates the end of the syntax diagram.

Syntax items

Syntax diagrams contain many different items. Syntax items include:

- Keywords - a command name or any other literal information.
- Variables - variables are italicized, appear in lowercase, and represent the name of values you can supply.
- Delimiters - delimiters indicate the start or end of keywords, variables, or operators. For example, a left parenthesis is a delimiter.
- Operators - operators include add (+), subtract (-), multiply (*), divide (/), equal (=), and other mathematical operations that may need to be performed.
- Fragment references - a part of a syntax diagram, separated from the diagram to show greater detail.
- Separators - a separator separates keywords, variables or operators. For example, a comma (,) is a separator.

Note: If a syntax diagram shows a character that is not alphanumeric (for example, parentheses, periods, commas, equal signs, a blank space), enter the character as part of the syntax.

Keywords, variables, and operators may be displayed as required, optional, or default. Fragments, separators, and delimiters may be displayed as required or optional.

Item type	Definition
Required	Required items are displayed on the main path of the horizontal line.
Optional	Optional items are displayed below the main path of the horizontal line.
Default	Default items are displayed above the main path of the horizontal line.

Syntax examples

The following table provides syntax examples.

Table 1. Syntax examples


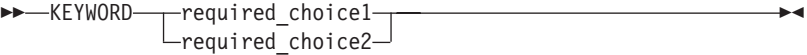
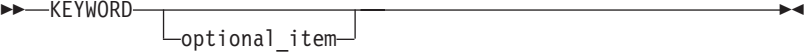
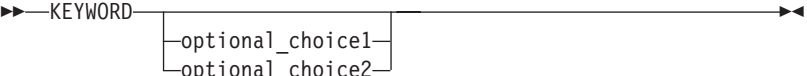
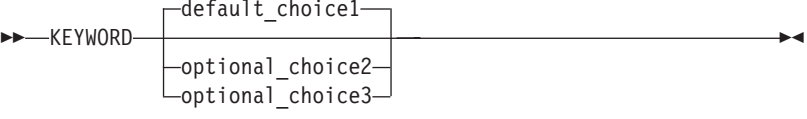

Item	Syntax example
Required item.	
Required items appear on the main path of the horizontal line. You must specify these items.	
Required choice.	
A required choice (two or more items) appears in a vertical stack on the main path of the horizontal line. You must choose one of the items in the stack.	
Optional item.	
Optional items appear below the main path of the horizontal line.	
Optional choice.	
An optional choice (two or more items) appears in a vertical stack below the main path of the horizontal line. You may choose one of the items in the stack.	
Default.	
Default items appear above the main path of the horizontal line. The remaining items (required or optional) appear on (required) or below (optional) the main path of the horizontal line. The following example displays a default with optional items.	
Variable.	
Variables appear in lowercase italics. They represent names or values.	

Table 1. Syntax examples (continued)

Item	Syntax example
Repeatable item.	
An arrow returning to the left above the main path of the horizontal line indicates an item that can be repeated.	
A character within the arrow means you must separate repeated items with that character.	
An arrow returning to the left above a group of repeatable items indicates that one of the items can be selected, or a single item can be repeated.	
Fragment.	
The fragment symbol indicates that a labeled group is described below the main syntax diagram. Syntax is occasionally broken into fragments if the inclusion of the fragment would overly complicate the main syntax diagram.	

Softcopy documents

The z/OS® Metal C publication is supplied in PDF and BookMaster® formats on the following CD: *z/OS Collection*, SK3T-4269. It is also available at <http://www.ibm.com/servers/eserver/zseries/zos/metalc/>.

To read a PDF file, use the Adobe Reader. If you do not have the Adobe Reader, you can download it from the Adobe Web site at www.adobe.com.

You can also browse the documents on the World Wide Web by visiting the z/OS library at <http://www.ibm.com/systems/z/os/zos/bkserv/>.

Note: For further information on viewing and printing softcopy documents and using BookManager®, see *z/OS Information Roadmap*.

z/OS Metal C on the World Wide Web

Additional information on z/OS Metal C is available on the World Wide Web on the z/OS Metal C home page at: <http://www.ibm.com/servers/eserver/zseries/zos/metalc/>.

This page contains late-breaking information about the z/OS Metal C product. There are links to other useful information, such as the z/OS Metal C information library and the libraries of other z/OS elements that are available on the Web. The z/OS Metal C home page also contains links to other related Web sites.

Where to find more information

Please see *z/OS Information Roadmap* for an overview of the documentation associated with z/OS.

Information updates on the web

For the latest information updates that have been provided in PTF cover letters and Documentation APARs for z/OS, see the online document at: http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/Shelves/ZDOCAPAR

This document is updated weekly and lists documentation changes before they are incorporated into z/OS publications.

The z/OS Basic Skills Information Center

The z/OS Basic Skills Information Center is a Web-based information resource intended to help users learn the basic concepts of z/OS, the operating system that runs most of the IBM mainframe computers in use today. The Information Center is designed to introduce a new generation of Information Technology professionals to basic concepts and help them prepare for a career as a z/OS professional, such as a z/OS system programmer.

Specifically, the z/OS Basic Skills Information Center is intended to achieve the following objectives:

- Provide basic education and information about z/OS without charge
- Shorten the time it takes for people to become productive on the mainframe
- Make it easier for new people to learn z/OS.

To access the z/OS Basic Skills Information Center, open your Web browser to the following Web site, which is available to all users (no login required):

<http://publib.boulder.ibm.com/infocenter/zos/basics/index.jsp>

Summary of changes

Summary of changes for SA23-2225-02 z/OS Version 1 Release 11

The book contains information previously presented in *z/OS Metal C Programming Guide and Reference*, SA23-2225-01, which supports z/OS Version 1 Release 10.

New information

- “float.h” on page 41 has been added.
- “math.h” on page 45 has been added.
- Information about system and static object libraries has been added in Chapter 3, “C functions available to Metal C programs,” on page 51.
- “strtod — Convert Character String to Double” on page 91 has been added.
- “strtof — Convert Character String to Float” on page 92 has been added.
- “strtold — Convert Character String to Long Double” on page 97 has been added.
- Table 15 on page 107 has been updated with new information.

Changed information

- “stdlib.h” on page 48 has been updated.
- “sprintf() — Format and Write Data” on page 72 has been updated.
- “sscanf() — Read and Format Data” on page 78 has been updated.

You may notice changes in the style and structure of some content in this document—for example, headings that use uppercase for the first letter of initial words only, and procedures that have a different look and format. The changes are ongoing improvements to the consistency and retrievability of information in our documents.

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

Summary of changes for SA23-2225-01 z/OS Version 1 Release 10

The book contains information previously presented in *z/OS Metal C Programming Guide and Reference*, SA23-2225-00, which supports z/OS Version 1 Release 9.

New information

- “Supplying your own HLASM statements” on page 13 has been added in “Programming with Metal C” on page 2.
- “Inserting non-executable HLASM statements into the generated source code” on page 22 has been added in “Programming with Metal C” on page 2.
- “AMODE-switching support” on page 23 has been added in “Programming with Metal C” on page 2.
- “Commands that compile and link applications that switch addressing modes” on page 35 has been added in “Programming with Metal C” on page 2.

- “Summary of useful references for the Metal C programmer” on page 39 has been added.
- The “builtins.h” on page 41 header has been added.

Changed information

- Sample prolog code and epilog code have been updated in “SCCNSAM(MYPROLOG) macro” on page 10 and “SCCNSAM(MYEPILOG) macro” on page 11.
- User-embedded HLASM statements has been changed to “Inserting HLASM instructions into the generated source code” on page 13.

This document contains terminology, maintenance, and editorial changes, including changes to improve consistency and retrievability.

Chapter 1. About IBM z/OS Metal C

Before z/OS V1R9, all z/OS XL C compiler-generated code required Language Environment®. In addition to depending on the C runtime library functions that are available only with Language Environment, the generated code depended on the establishment of an overall execution context, including the heap storage and dynamic storage areas. These dependencies prohibit you from using the XL C compiler to generate code that runs in an environment where Language Environment did not exist.

The XL C METAL compiler option, introduced in z/OS V1R9, generates code that does not require access to the Language Environment support at run time. Instead, the METAL option provides C-language extensions that allow you to specify assembly statements that call system services directly. Using these language extensions, you can provide almost any assembly macro, and your own function prologs and epilogs, to be embedded in the generated HLASM source file. When you understand how the METAL-generated code uses MVS™ linkage conventions to interact with HLASM code, you can use this capability to write freestanding programs.

Because a freestanding program does not depend on any supplied runtime environment, it must obtain the system services that it needs by calling assembler services directly. For information about how METAL-generated code uses MVS linkage conventions, see “Metal C and MVS linkage conventions” on page 2. For information about embedding assembly statements in the METAL-generated HLASM source code, see “Inserting HLASM instructions into the generated source code” on page 13.

You do not always have to provide your own libraries. IBM® supplies a subset of the XL C runtime library functions. This subset includes commonly used basic functions such as `malloc()`. For more information, see Chapter 3, “C functions available to Metal C programs,” on page 51.

Note: You can use these supplied functions or the ones that you provide yourself.

Metal C environments

Some of the functions require that an environment be created before they are called. You can create the environment by using a new function, `__cinit()`. This function will set up the appropriate control blocks and return an environment token to the caller. The caller must then ensure that GPR 12 contains this token when calling Metal C functions that require an environment. When the environment is no longer needed, a new function, `__cterm()`, can be used to perform cleanup, freeing all resources that had been obtained by using the token.

An environment created by `__cinit()` can be used in both AMODE 31 and AMODE 64. In conjunction with this, the Metal C run time maintains both a below-the-bar heap and an above-the-bar heap for each environment. Calls to `__malloc31()` always affect the below-the-bar heap. Calls made in AMODE 31 to all other functions that obtain storage will affect the below-the-bar heap; calls made in AMODE 64 affect the above-the-bar heap.

The storage key for all storage obtained on behalf of the environment is the psw key of the caller. The caller needs to ensure that the environment is always used with the same or compatible key.

Programming with Metal C

When you want to build an XL C program that can run in any z/OS environment, you can use the Metal C programming features provided by the XL C compiler as a high level language (HLL) alternative to writing the program in assembly language.

Metal C programming features facilitate direct use of operating-system services. For example, you can use the C programming language to write installation exits.

When the METAL option is in effect, the XL C compiler:

- Generates code that is independent of Language Environment.

Note: Although the compiler generates default prolog and epilog code that allows the Metal C code to run, you might be required to supply your own prolog and epilog code to satisfy runtime environment requirements.

- Generates code that follows MVS linkage conventions. This facilitates interoperations between the Metal C code and the existing code base. See “Metal C and MVS linkage conventions.”

Note: Metal C also provides a feature that improves the program’s runtime performance. See “NAB linkage extension” on page 3.

- Provides support for accessing the data stored in data spaces. See “AR-mode programming support” on page 24.
- Provides support for embedding your assembly statements into the compiler-generated code. See “Inserting HLASM instructions into the generated source code” on page 13.

If you use the METAL compiler option together with XL C optimization capabilities, you can use C to write highly optimized system-level code.

The METAL compiler option implies certain other XL C compiler options and disables others. For detailed information, see METAL option in *z/OS XL C/C++ User’s Guide*.

Metal C and MVS linkage conventions

Because Metal C follows MVS linkage conventions, it enables the compiler-generated code to interoperate directly with the existing code base to facilitate the following operations:

- Passing parameters. See “Parameter passing.”
- Returning values. See “Return values” on page 3.
- Setting up function save areas. See “Function save areas” on page 3.

For detailed information about MVS linkage conventions, see “Linkage Conventions” in *z/OS MVS Programming: Assembler Services Guide*, SA22-7605.

Parameter passing

The pointer to the parameter list is in GPR 1.

The parameter list is divided into slots.

- The size of each slot depends on the addressing mode:
 - For 31-bit mode (AMODE 31), each slot is four bytes in length.
 - For 64-bit mode (AMODE 64), each slot is eight bytes in length.

- Metal C derives the content of each slot from the function prototype, which follows C by-value semantics (that is, the value of the parameter is passed into the slot).

Notes:

1. If a parameter needs to be passed by reference, the function prototype must specify a pointer of the type to be passed.
2. Under AMODE 31 only: The high-order bit is set on the last parameter if both of the following are true:
 - The called function is a variable arguments function.
 - The last parameter passed is a pointer.

Return values

For any addressing mode, integral type values are returned in GPR 15. Under AMODE 31 only, a 64-bit integer value is returned in GPR 15 + GPR 0 (that is, the high-half of the 64-bit value is returned in GPR 15 and the low-half is returned in GPR 0). All other types are returned in a buffer whose address is passed as the first parameter.

Function save areas

GPR 13 contains the pointer to the dynamic storage area (DSA).

The DSA includes:

- 72-byte save area size for an AMODE 31 function.
- Parameter area for calling other functions. The default pointer size for a parameter or return value is based on the amode attribute of the function prototype.
- Temporary storage that is preallocated for the compiler-generated code and the user-defined automatic variables.

The save area is set up at the beginning of the DSA.

If the function calls only primary-mode functions, the save area format depends on the AMODE:

- Under AMODE 31, the save area takes the standard 18-word format.
- Under AMODE 64, the save area takes the 36-word F4SA format and the compiler will generate code to set up the F4SA signature in the second word of the save area.

If the function needs to call an AR-mode function, the save area will take the 54-word F7SA format, regardless of the addressing mode, and the compiler will generate code to set up the F7SA signature in the second word of the save area.

NAB linkage extension

Metal C code needs to use dynamic storage area (DSA) as stack space. Each time a function is called, its prolog code acquires this space and, when control is returned to the calling function, its epilog code releases the stack space.

Metal C avoids excessive acquisition and release operations by providing a mechanism that allows a called function to rely on pre-allocated stack space. This mechanism is the next available byte (NAB). All Metal C runtime library functions, as well as functions with a default prolog code, use it and expect the NAB address to be set by the calling function. The code that is generated to call a function includes the setup instructions to place the NAB address in the "Address of next save area" field in the save area. The called function simply goes to the calling

function's save area to pick up the NAB address that points to its own stack space. As a result, the called function does not need to explicitly obtain and free its own stack space.

Note: If usage of the NAB linkage extension requires more stack space than has been allocated, there will be unexpected results. The program must establish a DSA that is large enough to ensure the availability of stack space to all downstream programs. Downstream programs include all functions that are defined in the program as well as the library functions listed in Appendix A, "Function stack requirements," on page 107.

The location of the "Address of next save area" field depends on the save area format:

- In the standard 72-byte save area, it is the third word.
- In the F4SA or F7SA save area, it is the 18th doubleword.

Compiler-generated HLASM source code

When the METAL option is in effect, the XL C compiler generates code in the HLASM source code format.

Characteristics of compiler-generated HLASM source code

Any assembly instructions that you provide need to work with the instructions that are generated by the compiler. Before you provide those instructions, you need to be aware of the characteristics of compiler-generated HLASM source code.

You need to be aware that:

- Because the compiler uses relative-branching instructions, it is not necessary to establish code base registers. When the compiler detects user-embedded assembly statements, it emits a COPY instruction to bring in the IEABRC macro to assist any branching instructions that might rely on establishment of a code base register. For other instructions (such as LA or EX) that rely on the establishment of a code base register, you might need to add code to establish your own code base register. If you do so, you will need to add a DROP instruction to free the code base register you established when it is no longer needed.
- If the compiler needs to produce literals, GPR 3 will be set up as the base register to address the literals. This addressability is established after the prolog code. The literals are organized by the LTOrg instruction placed at the end of the epilog code. With the presence of user-embedded assembly statements, the compiler assumes there will be literals and establishes GPR 3 to address those literals.
- If you want code to be naturally reentrant, you must not use writable static or external variables; such variables are part of the code.
- There is only one CSECT for each compilation unit. The CSECT name is controlled by the CSECT option.
- Due to the flat name space and the case insensitivity required by HLASM, the XL C compiler prepends extra qualifiers to user names to maintain the uniqueness of each name seen by HLASM. This is referred to as *name encoding*. External symbols are not subject to the name-encoding scheme as they need to be referenced by the exact symbol names.
- The external symbols are determined by the compiler LONGNAME option.
 - If the NOLONGNAME option is in effect:
 - All external symbols are truncated to eight characters.

- All external symbols are converted to upper case.
- All "_" characters are replaced with the "@" character.
- If the LONGNAME option is in effect the compiler emits an ALIAS instruction to make the real C name externally visible. Because the length limit supported by the ALIAS instruction depends on the HLASM release, the C compiler does not enforce any length limit here.

Note: The HLASM GOFF option is necessary to allow the ALIAS instructions to be recognized. See Figure 40 on page 35.

- GPR 13 is established as the base of the stack space.
- GPR 10 and GPR 11 may be used exclusively to address static and constant data. They should not be used in the user-embedded assembly statements.
- The compiler will generate code to preserve FPR 8 through FPR 15 if they are altered by the function.
- For AMODE 31 functions: The compiler will generate code to preserve the high halves of the 64-bit GPRs if they are altered or if there are user-embedded assembly statements.
- The addressing mode is determined by the compiler option. When the compiler option LP64 is in effect, the addressing mode is AMODE 64; otherwise it is AMODE 31.

Structure of a compiler-generated HLASM source program

Each compiler-generated HLASM source program has the following elements:

- File-scope header for each compilation unit.
- For each function:
 - A function header.
 - A function body.
 - A function trailer.
- File-scope trailer for each compilation unit.

File-scope header: Statements in the file-scope header apply to the entire compilation unit and might have the following statements:

- TITLE statement to specify the product information of the compiler and the source file being compiled.
- ALIAS/EXTRN statement to declare the external symbols that are referenced in the program, if the LONGNAME compiler option is in effect.
- CSECT statement to identify the relocatable control section in the program.
- AMODE statement to specify the addressing mode.
- RMODE statement to specify the residency mode for running the module.
- Assembly statements to declare the HLASM global SET symbols used by the compiler-generated code for communicating information to the user-embedded prolog and epilog code, if the compiler detects user-embedded prolog and epilog code.
- SYSSTATE ARCHLVL=2 statement, which identifies the minimum hardware requirement.
- COPY IEABRC statement that ensures that all branch instructions are changed to relative-branching instructions, in the event that the XL C compiler encounters user-embedded assembly statements.
- Prefix data to embed a compiler signature and to record attributes about the compilation. See "Prefix data" on page 6 for details.

Function header: The function header might have the following statements or code:

- ALIAS/ENTRY statement to define the entry point by associating its C symbol with the generated HLASM name, if the LONGNAME compiler option is in effect.
- Assembly statements to set the values for the declared HLASM global SET symbols, if the compiler detects user-embedded prolog and epilog code.
- Prolog code, which might be either the default prolog code generated by the compiler or user-embedded prolog code.

Function trailer: The function trailer might include the following statements and code:

- DROP statement to clear all established base registers.
- Epilog code, which might be either the default epilog code generated by the compiler or user-embedded epilog code.
- LTORG statement to instruct the assembler to group all literals at that point in the code.
- DSECT statement that provides a map for the automatic variables.
- DSECT statement that provides a map for the parameters.

File-scope trailer: The file-scope trailer might have the following statements or areas:

- DC statements to define static variables with their initial values.
- DSECT statement to provide a map for the static variables.
- DC statements that define constants.
- ALIAS/ENTRY statement to define all external variables with their initial values.
- END statement to specify compiler product information and the compilation date.

Prefix data: Prefix data is generated to supply a signature, the timestamp of the compilation date, the compiler version, and some control flags. It is placed at the beginning of the code that follows an instruction for branching around the prefix data.

Note: Program code should reference ENTRY rather than CSECT to avoid unnecessary branching.

The prefix data consists of a 360-byte fixed section and optional sections.

There are four sets of flag bits. Flag sets 1 through 3 are reserved for future use. In flag set 4, the position of each "1" bit flag setting corresponds to the position of a specific optional section. For example, the optional section that corresponds to the leftmost "1" bit flag setting appears first, followed by the optional section that corresponds to the next "1" bit flag setting.

DC	XL8'00C300C300D50000'	Signature
DC	CL8'yyyymmdd'	Compiled Date YYYYMMDD
DC	CL6'hmmss'	Compiled Time HHMMSS
DC	XL4'41090000'	Compiler Version
DC	XL2'0000'	
DC	BL1'00000000'	Flag Set 1
DC	BL1'00000000'	Flag Set 2
DC	BL1'00000000'	Flag Set 3
DC	BL1'00000000'	Flag Set 4 1 , 2
DC	XL4'00000000'	

Notes:

1. In flag set 4, a bit setting of "1" in the first position ('1.....') would indicate the presence of a user comment string.
2. In flag set 4, a bit setting of "1" in the second position ('.1....') would indicate the presence of a service string.

Figure 1. Structure of prefix data

Figure 2 shows C code to be compiled with the SERVICE compiler option. When the SERVICE option is in effect, the string in the object module is loaded into memory at run time. If the program terminates abnormally, the string is displayed in the traceback.

```
#pragma comment(copyright,"copyright comment")
#pragma comment(user," + user comment")
int main() {
    return 0;
}
```

Figure 2. Generation of an optional section in the prefix data

Figure 3 on page 8 shows the prefix data generated for the code in Figure 2.

DC	XL8'00C300C300D50000'	Signature	
DC	CL8'yyyymmdd'	Compiled Date YYYYMMDD	
DC	CL6'hmmss'	Compiled Time HHMMSS	
DC	XL4'41090000'	Compiler Version	
DC	XL2'0000'		
DC	BL1'00000000'	Flag Set 1	
DC	BL1'00000000'	Flag Set 2	
DC	BL1'00000000'	Flag Set 3	
DC	BL1'11000000'	Flag Set 4	1 , 2
DC	XL4'00000000'		
DS	0H		3
DC	AL2(32)		4
DC	C'copyright comment + user comment'		
DC	C' '		
DS	0H		5
DC	AL2(8)		6
DC	C'service'		
DC	C' '		

Notes:

1. The first bit in flag set 4 is set to "1", which indicates that comments were embedded.
2. The second bit in flag set 4 is set to "1", which indicates the code was compiled with the SERVICE option.
3. The first optional section starts at +36 (0x24).
4. This statement indicates that the data specified in the #pragma comment directive has a length of 32 bytes.
5. The second optional section starts at +70 (0x46).
6. This statement indicates that the data specified in the service string has a length of 8 bytes.

Figure 3. Prefix data with optional sections

Prolog and epilog code

The primary functions of prolog code are:

- To save the calling function's general-purpose registers in the calling function's save area.
- To obtain the dynamic storage area for this function.
- To chain this function's save area to the calling function's save area, in accordance with the MVS linkage convention.

The primary functions of epilog code are:

- To relinquish this function's dynamic storage area.
- To restore the calling function's general-purpose registers.
- To return control to the calling function.

Note: AR-mode functions require additional prolog and epilog functions. See "AR-mode programming support" on page 24 for details.

Supplying your own prolog and epilog code

If you need the prolog and epilog code to provide additional functionality, you can use #pragma directives to instruct the compiler to use your own HLASM prolog and epilog code. Figure 4 on page 9 provides an example.

```

#pragma prolog(foo,"MYPROLOG")
#pragma epilg(foo,"MYEPILOG")
int foo() {
    return 0;
}

```

Figure 4. Specification of your own prolog and epilg code for a function

To apply the same prolog and epilg code to all your functions in the C source file, use the PROLOG and EPILOG compiler options. When you use the PROLOG and EPILOG compiler options, by default, your prolog and epilg code is applied only to the functions that have external linkage. To apply your prolog and epilg code to all functions defined in the compilation unit, use the new "all" suboption provided by z/OS V1R11 XL C compiler. For detailed information, see PROLOG and EPILOG options in *z/OS XL C/C++ User's Guide*.

The string you supplied to the PROLOG/EPILOG options or the #pragma directives must contain valid HLASM statements. The compiler does not validate the content of the string but it does take care of some formatting for you:

- If your string contains only a macro name, as shown in Figure 4, you do not need to supply leading blanks.
- If the length of your HLASM statement exceeds 71 characters, you do not need to break it into multiple lines. The compiler will handle that for you.

Your prolog code needs to ensure that:

- The primary functions of the prolog code have been performed.
- Extra DSA space is acquired, in the event that the NAB is needed for the referenced functions.
- Upon exit of your prolog code:
 - GPR 13 points at the DSA for this function.
 - GPR 1 points at the parameter list supplied by the calling function.

Your epilg code needs to ensure that:

- The primary functions of the epilg code have been performed.
- The content of GPR 15, on entry to your epilg code, is preserved. If a 64-bit integer value is returned under AMODE 31, the content of GPR 0 also needs to be preserved.

Your prolog and epilg code does not need to perform the following functions:

- Preserve the calling function's floating-point registers.
- Preserve the high-halves of 64-bit general purpose registers in AMODE 31 functions.
- Set up the NAB for the called functions.

Compiler-generated global SET symbols

When you supply your prolog and epilg code, the compiler generates the assembly instructions that set up global SET symbols for communicating compiler-collected information to your prolog and epilg code. Your prolog and epilg code can use this information to determine the code sequence generated by your macros.

Table 2 on page 10 describes global SET symbols defined by the compiler.

Table 2. Compiler-generated global SET symbols

Global SET symbol	Type	Description
&CCN_DSASZ	Arithmetic	The size of the dynamic storage area for the function.
&CCN_SASZ	Arithmetic	The size of the function save area: <ul style="list-style-type: none"> • 72 = standard format • 144 = F4SA format • 216 = F7SA format
&CCN_ARGS	Arithmetic	The number of fixed arguments expected by the function.
&CCN_RLOW	Arithmetic	The starting register number to be used in the STORE MULTIPLE instruction for saving the registers of callers if the compiler were to generate that instruction itself.
&CCN_RHIGH	Arithmetic	The ending register number to be used in the STORE MULTIPLE instruction for saving the registers of callers.
&CCN_LP64	Logical	Set to "1" if the LP64 compiler option is specified.
&CCN_NAB	Logical	Set to "1" when there are called programs that depend on the dynamic storage to be pre-allocated. In this case, the prolog code needs to add a generous amount to the size set in &CCN_DSASZ when the dynamic storage is obtained.
&CCN_ALTGPR(16)	Logical	The array representing the general purpose registers. Subscript 1 represents GPR 0 and subscript 16 represents GPR 15. A subscript is set to "1" whenever the corresponding register is altered by the compiler-generated code.
&CCN_PRCN	Character	The symbol representing the function.
&CCN_CSECT	Character	The symbol representing the CSECT in effect.
&CCN_LITN	Character	The symbol representing the LTORG generated by the compiler.
&CCN_BEGIN	Character	The symbol representing the first executable instruction of the function generated by the compiler.
&CCN_ARCHLVL	Character	The symbol representing the architecture level specified in the ARCH option.
&CCN_ASCM	Character	The ASC mode of the function: <ul style="list-style-type: none"> • A=AR mode • P=Primary mode For information about AR mode, see "AR-mode programming support" on page 24.

As of z/OS V1R11, the global SET symbols that are set by your prolog and epilog code can conditionally enable and disable code sequences generated by the compiler.

Table 3 describes global SET symbols that you can set.

Table 3. User modifiable global SET symbols

Global SET symbol	Type	Default	Description
&CCN_SASIG	Logical	1	Set to "1" to enable the save area signature generation. Set to "0" to disable the save area signature generation.

SCCNSAM(MYPROLOG) macro

Sample macros for prolog and epilog code are supplied in the SCCNSAM data set. Figure 5 on page 11 shows the sample prolog code.


```

MACRO
&NAME MYPROLOG
GBLC &CCN_PRCN
GBLC &CCN_LITN
GBLC &CCN_BEGIN
GBLC &CCN_ARCHLVL
GBLA &CCN_DSASZ
GBLA &CCN_RLOW
GBLA &CCN_RHIGH
GBLB &CCN_NAB
GBLB &CCN_LP64
LARL 15,&CCN_LITN
USING &CCN_LITN,15
GBLA &MY_DSASZ
&MY_DSASZ SETA 0
AIF (&CCN_LP64).LP64_1
STM 14,12,12(13)
AGO .NEXT_1
.LP64_1 ANOP
STMG 14,12,8(13)
.NEXT_1 ANOP
AIF (&CCN_DSASZ LE 0).DROP
&MY_DSASZ SETA &CCN_DSASZ
AIF (&CCN_DSASZ GT 32767).USELIT
AIF (&CCN_LP64).LP64_2
LHI 0,&CCN_DSASZ
AGO .NEXT_2
.LP64_2 ANOP
LGHI 0,&CCN_DSASZ
AGO .NEXT_2
.USELIT ANOP
AIF (&CCN_LP64).LP64_3
L 0,=F'&CCN_DSASZ'
AGO .NEXT_2
.LP64_3 ANOP
LGF 0,=F'&CCN_DSASZ'
.NEXT_2 AIF (NOT &CCN_NAB).GETDSA
&MY_DSASZ SETA &MY_DSASZ+1048576
LA 1,1
SLL 1,20
AIF (&CCN_LP64).LP64_4
AR 0,1
AGO .GETDSA
.LP64_4 ANOP
AGR 0,1
.GETDSA ANOP
STORAGE OBTAIN,LENGTH=(0),BNDRY=PAGE
AIF (&CCN_LP64).LP64_5
LR 15,1
ST 15,8(,13)
L 1,24(,13)
ST 13,4(,15)
LR 13,15
AGO .DROP
.LP64_5 ANOP
LGR 15,1
STG 15,136(,13)
LG 1,32(,13)
STG 13,128(,15)
LGR 13,15
.DROP ANOP
DROP 15
MEND

```

Figure 5. SCCNSAM(MYPROLOG)

SCCNSAM(MYEPILOG) macro

Sample macros for prolog and epilog code are supplied in the SCCNSAM data set. Figure 6 on page 12 shows the sample epilog code.

```

MACRO
&NAME MYEPILOG
GBLC &CCN_PRCN
GBLC &CCN_LITN
GBLC &CCN_BEGIN
GBLC &CCN_ARCHLVL
GBLA &CCN_DSASZ
GBLA &CCN_RLOW
GBLA &CCN_RHIGH
GBLB &CCN_NAB
GBLB &CCN_LP64
GBLA &MY_DSASZ
AIF (&MY_DSASZ EQ 0).NEXT_1
AIF (&CCN_LP64).LP64_1
LR 1,13
AGO .NEXT_1
.LP64_1 ANOP
LGR 1,13
.NEXT_1 ANOP
AIF (&CCN_LP64).LP64_2
L 13,4(,13)
AGO .NEXT_2
.LP64_2 ANOP
LG 13,128(,13)
.NEXT_2 ANOP
AIF (&MY_DSASZ EQ 0).NODSA
AIF (&CCN_LP64).LP64_3
ST 15,16(,13)
AGO .NEXT_3
.LP64_3 ANOP
STG 15,16(,13)
.NEXT_3 ANOP
LARL 15,&CCN_LITN
USING &CCN_LITN,15
STORAGE RELEASE,LENGTH=&MY_DSASZ,ADDR=(1)
AIF (&CCN_LP64).LP64_4
L 15,16(,13)
AGO .NEXT_4
.LP64_4 ANOP
LG 15,16(,13)
.NEXT_4 ANOP
.NODSA ANOP
AIF (&CCN_LP64).LP64_5
L 14,12(,13)
LM 1,12,24(13)
AGO .NEXT_5
.LP64_5 ANOP
LG 14,8(,13)
LMG 1,12,32(13)
.NEXT_5 ANOP
BR 14
DROP 15
MEND

```

Figure 6. SCCNSAM(MYEPILOG)

Compiler-generated default prolog and epilog code

The default prolog and epilog code generated for the main function is very much the same as the code produced by the sample prolog and epilog macros. That is, a STORAGE macro is used to obtain and release a dynamic storage area of 1 MB. For functions other than main, the prolog code simply picks up its DSA pointer (the NAB pointer) from the "Address of next save area" field in the calling function's save area.

Supplying your own HLASM statements

Before you insert your own HLASM statements into your C source file, be aware of the following information:

- The compiler does not recognize either the syntax or the semantics of the HLASM statements embedded in the C `__asm` statement. You need to ensure that the embedded HLASM statements:
 - Meet the requirements of the assembly step that follows the compilation step.
 - Function correctly when embedded in the compiler-generated HLASM source file.
- In the HLASM syntax, the first field is the label field, followed by the op-code, and the rest of the HLASM instruction. If there is no label field, you must leave a blank space at the beginning of the string. Other than this, you can code the rest of the HLASM instruction as you do in HLASM.
- You do not have to consider HLASM line-width requirements. You can code an instruction in the code format string continuously, in accordance with the limitation of the C source file. The C compiler breaks up a code format string that exceeds 71 characters in the HLASM output, inserting continuation characters as required.

Inserting HLASM instructions into the generated source code

You can use the `__asm` language extension to specify assembly instructions to be embedded within the generated HLASM source code. For example, you can embed assembly statements that invoke assembler macros to obtain system services.

Use the `__asm` statement only to embed a short sequence of assembler instructions into a C function, to perform actions that cannot be done using C statements. If you need to use a long routine, put the assembly statements into a source file, assemble it separately, and then call the routine from the C program.

Note: The compiler supports a collection of hardware built-in functions, such as `__csg`. These hardware built-in functions allow the compiler more freedom in blending embedded assembly statements with the rest of the code. For this reason, a hardware built-in function might be better than an `__asm` statement for embedding the assembly instructions that you need.

In addition to the `__asm` language extension, there are language constructs for the following purposes:

- Reserving a register for a global variable of the pointer type. See “Reserving a register for a global variable” on page 22.
- Invoking a macro in the list form. See “Specifying and using the list form of a macro” on page 20.
- Supplying your own function prologs and epilogs. See “Prolog and epilog code” on page 8.

For information about hardware built-in functions, see *Using hardware built-in functions in z/OS XL C/C++ Programming Guide*.

Using the `__asm` statement

For the complete `__asm` statement syntax, see *Inline assembly statements in z/OS XL C/C++ Language Reference*.

Within the `__asm` statement, the *code format string* specifies the assembly statement to be embedded in the compiler-generated HLASM source file. Figure 7 on page 14 provides an example of a simple code format string, enclosed in double quotation

marks, in an `__asm` statement.

```
void foo() {  
    __asm ( " AR 1,2" );  
}
```

Figure 7. Simple code format string

Treatment of the code format string

The compiler treats the code format string in an `__asm` statement similarly to the way the `printf` function treats a format string, with the following exception: Instead of printing out the string during program execution, the compiler inserts it after the generated sequence of assembly statements, before the `END` statement.

More than one assembler instruction can be put into the code format string. As shown in Figure 8, each assembler statement must be separated by the new line character `'\n'` (like the new line character that is used in a `printf` format string).

```
void foo() {  
    __asm ( " AR 1,2\n AR 1,2" );  
}
```

Figure 8. Code format string with two instructions

The example in Figure 8 will embed two `" AR 1,2"` instructions in the HLASM source code. You can make the statement more readable by breaking the string into two. In C, adjacent string literals are automatically concatenated and treated as one. The sample code in Figure 8 and Figure 9 generate the same output.

```
void foo() {  
    __asm ( " AR 1,2\n" 1  
           " AR 1,2" ); 2  
}
```

Notes:

1. The character `"\n"` is still required to delimit statements.
2. The second statement also begins with a blank space.

Figure 9. Code format string with two instructions, formatted for readability

C expressions as `__asm` operands

You can use substitution specifiers in a code format string just as you can in a `printf` format string. The substitution specifier tells the compiler to substitute the specified C expression into the corresponding *__asm operand* when it embeds the assembly statement in the HLASM source code. You must ensure that the substitution converts the code format string into a valid assembler instruction.

Note: In this document, operands used in a code format string are referred to as `__asm` operands.

An embedded assembly statement can use any C-language expression that is in scope as an `__asm` operand. The *constraint* tells the compiler what to do with the C expression that follows it.

Substitution of a C variable into an `__asm` operand: Figure 10 on page 15 shows an `__asm` statement that substitutes a C variable into an output `__asm` operand. Figure 11 on page 15 shows those assembly instructions.

```

void foo() {
    int x;
    __asm ( " ST 12,%0\n" : "=m"(x) ); ,
}

```

Notes:

1. A colon that marks the beginning of the list of output `__asm` operands follows the code format string.
2. The output `__asm` operand is `"=m"(x)`. The constraint `"m"` communicates the syntactic requirement to the XL C compiler:
 - The symbol `"="` means the (output) `__asm` operand will be modified.
 - The letter `"m"` means that the output `__asm` operand is a memory operand.
3. The C expression is the variable `x`.
4. The compiler does not know that the embedded assembly instruction is `ST`, nor does it know the HLASM syntactic requirement of the second `ST` operand.
5. The variable `x` is the first `__asm` operand in the example, and therefore corresponds to `%0` in the code format string.

Figure 10. Substitution of a C variable into an output `__asm` operand

From the code format string used in Figure 10, the XL C compiler embeds the instructions shown in Figure 11, in the generated HLASM source code

```

1      2
LA      1,@3x
ST      12,0(1)
3

```

Notes:

1. The `LA` instruction is inserted by the C compiler as a result of processing the `"=m"(x)` `__asm` operand.
2. `@3x` is the HLASM symbol name that the compiler assigned to the local variable `x`. Local C symbol names are mapped to HLASM symbol names so that each local variable has a unique name in the HLASM source file.
3. `0(1)` is substituted into `"%0"`, which specified the first `__asm` operand in the code format string in Figure 10 (`ST 12,%0`).

Figure 11. HLASM source code embedded by the `__asm` statement in Figure 10

Substitution of a C pointer into an `__asm` operand: The code format string in Figure 12 on page 16 invokes the `WTO` macro by using the execute form of the macro with a user-defined buffer.

In general, you do not control which registers are used during the operand substitution, as illustrated in Figure 12 on page 16. For an example that allows you to specify registers, see Figure 16 on page 18.

```

int main() {
    struct WTO_PARM {
        unsigned short len;
        unsigned short code;
        char text[80];
    } wto_buff = { 4+11, 0, "hello world" };

    __asm( " WTO MF=(E,(%0)) " : : "r"(&wto_buff));
    return 0;
}

```

Notes:

1. The absence of a label necessitates that a blank space begin the code format string.
2. There are no output `__asm` operands. The end of the output `__asm` operands list is marked by a colon, which is then followed by a comma-separated list of input `__asm` operands. The colon starting the list of input `__asm` operands is not necessary if there are no input operands (which is the case in Figure 10 on page 15).
3. The input `__asm` operand consists of two components:
 - A constraint "r" that tells the compiler that the operand will be stored in a GPR.
 - An expression (`&wto_buff`) that states that the operand is the address of the message text in the C structure `wto_buff`.

Figure 12. Substitution of a C pointer into an `__asm` operand

Definition of multiple `__asm` operands: In Figure 13, the compiler is instructed to store the third defined C variable (z) in a register, and then substitute that register into the third `__asm` operand `%2`.

```

void foo() {
    int x, y, z;
    __asm ( " ST 12,%0\n"
           " ST 12,%1\n"
           " AR 12,%2" : "=m"(x), "=m"(y) : "r"(z) );
}

```

Notes:

1. The code format string instructs the compiler to embed an assembly statement that substitutes the register (with contents of the C variable z) into the third `__asm` operand (`%2`).
2. The constraint "m" instructs the compiler to use memory operands for the output variables x and y.
3. The constraint "r" instructs the compiler to use a register for the input variable z.

Figure 13. `__asm` operand lists

In the compiler-generated HLASM code shown in Figure 14 on page 17, GPR 4 is assigned to the variable z.

```

L 4,@5z      1
LA 2,@4y     2
LA 1,@3x
ST 12,0(1)   3
ST 12,0(2)
AR 12,4

```

Notes:

1. The first assembly statement `L 4,@5z` is added by the compiler to get `z` into the form specified by the input `__asm` operand constraint `"r"`.
2. The next two instructions are added by the compiler to get the variables `x` and `y` into the form specified by the output `__asm` operand constraints `"=m"`.
3. The contents of the code format string are appended in the last three instructions.

Figure 14. Example of compiler-generated HLAASM code for the `__asm` statement in Figure 13 on page 16

Register specification: In general you do not have control over which registers are used during operand substitutions. The register assignment might change when you use different options or optimization levels, or when the surrounding C code is changed.

In cases where you specify explicit registers to be used in the embedded instructions, you should code a clobber list, as shown in Figure 16 on page 18. Without the clobber list, the `__asm` statement embeds incorrect assembly statements, as shown in Figure 15.

```

__asm ( " LR 0,%0\n"      /* load &p1 */
      " LR 1,%1\n"      /* load &dcB */
      " SVC 21"
      : : "r"(&p1), "r"(&dcB)); 1

```

Note: The output and input `__asm` operand lists are positional. If there are no output `__asm` operands, the colons separating the output and input operand list are still needed. Because the compiler has no knowledge of assembly instructions and does not understand the `LR` instruction, it does not know that the registers `GPR 0` and `GPR 1` are being used in the statement. Any connection between the `__asm` statement and the rest of the C code must be specified via the `__asm` operand lists. The information provided in the lists should prevent the compiler from incorrectly moving the other references surrounding the `__asm` statement. In this example, because the compiler doesn't know that `GPR 0` and `GPR 1` are being used, it will embed incorrect assembly statements.

Figure 15. Unsuccessful attempt to specify registers

To prevent the compiler from incorrectly moving the other references surrounding the `__asm` statement, add a clobber list after a colon that follows the input `__asm` operands, as shown in Figure 16 on page 18.

Note: Do not try to use the `__asm` statement to embed a long piece of assembly code with many operand specifiers and stringent register requirements. There is a limited number of registers available for the compiler to use in the operand specifiers, and in the surrounding code generation. If too many registers are clobbered, there may not be enough registers left for the `__asm` statement. The same applies if there are too many specifiers.

```

__asm ( " LR 0,%0\n"      /* load &p1 */
        " LR 1,%1\n"      /* load &dcbl */
        " SVC 21"
        : : "r"(&p1), "r"(&dcbl) : "r0", "r1");

```

1, 2

Notes:

1. This colon is not needed if there is no clobber list.
2. The clobber list specifies the registers that can be modified by the assembly instructions.

Figure 16. Register specification with clobbers

C expressions as read-write __asm operands

If you use the same __asm operand for both input and output, you must take care that you tell the compiler that the input __asm operand refers to the same variable as the corresponding output __asm operand. For example, the code format string in Figure 17 uses one register to store a single __asm operand that is used for both input and output.

Definition of __asm operands for both input and output via an operand list:

This topic describes how to use a code format string to define __asm operands that can be used for both input and output.

You can use either input and output operand strings both incorrectly (Figure 17) and correctly (Figure 19 on page 19). The code in Figure 17 is incorrect because the AR statement reads the first operand and then modifies it, but the =r constraint specifies the output aspect only.

```

__asm ( " AR %0,%1" : "=r"(x) : "r"(y) );

```

1

Note: No input operand is specified for variable x. The compiler will not know that input and output are stored in the same variable.

Figure 17. Unsuccessful attempt to define an __asm operand for both input and output

```

L 2,@4y
LA 1,@3x
AR 4,2
ST 4,0(,1)

```

1

Note: GPR 4, which is meant for input as well as output, is not loaded from variable x before the code format string is embedded because the code format string in Figure 17 specified variable x as an output operand only.

Figure 18. Incorrect HLASM source code from Figure 17

If a code format string uses a single __asm operand for both input and output, you must ensure that the embedded assembly statements will perform both of the following tasks:

- Define the variable as an input operand as well as an output operand.
- Define both an input operand and an output operand that refers to the same variable. The variable name is not sufficient for this purpose. See Figure 19 on page 19.

Figure 19 shows the code format string that will embed the correct assembly statements, shown in Figure 20.

```
__asm ( " AR %0, %1" : "=r"(x) : "r"(y), "0"(x) );
```

1 2 3 4

Notes:

1. %0 is the first operand in the code format string.
2. This example has one output __asm operand, "=r"(x).
3. Within the input __asm operand list "r"(y), "0"(x), the __asm operands are separated by a comma.
4. An input operand "0"(x) is added to the input field. The constraint of this __asm operand is the "0", which tells the compiler that:
 - This input __asm operand is the same as the output __asm operand %0. (A numeral zero in the constraint ("0") refers to %0; a numeral one in a constraint would refer to %1; and so on.)
 - The register needs to be loaded with variable x, as shown in Figure 20, before the code format string is embedded in the HLASM output.

Figure 19. Successful definition of an __asm operand for both input and output

The __asm statement in Figure 19 embeds the assembly statements in Figure 20.

```
L 2,@4y
L 4,@3x
LA 1,@3x
AR 4,2
ST 4,0(,1)
```

1

Note: The compiler inserted L 4,@3x at the beginning of the instruction sequence because the code format string in Figure 19 included both the output operand "=r"(x) and the input operand "0"(x). Together, these statements tell the compiler that the register for the first operand %0 will be used for variable x, which has a value that can be either an input or an output operand.

Figure 20. Correct HLASM source code output from Figure 19

Definition of an __asm operand for both input and output via the "+"

constraint: You can also use the "+" constraint to specify that an __asm operand is used for both input and output.

In Figure 21, the "+" constraint is used to define the variable x is used both as input and output.

```
__asm ( " AR %0, %1" : "+r"(x) : "r"(y));
```

Note: The compiler handles the "+" constraint by turning it into a "=" constraint, and then appending a matching constraint operand at the end of the input list. Therefore this example is parsed as though the operand list in Figure 19 is given.

Figure 21. The + constraint to define an __asm operand for both input and output

Note that an operand can be matched only once. When you use the "+" constraint to implicitly define matching input and output __asm operands, do not explicitly define a corresponding __asm operand.

Figure 22 shows an erroneous example of an __asm operand that is defined both implicitly and explicitly. The notes identify the unnecessary code.

```
__asm ( " AR %0, %1" : "+r"(x) : "r"(y), "0"(x));
```

1 2 3

Notes:

1. %0 is the first operand in the code format string.
2. This example has one output __asm operand, "+r"(x). The "+" constraint implicitly defines a matching input __asm operand.
3. You do not have to define __asm operand "0"(x) explicitly.

Figure 22. Error: Redundant definition of an __asm operand

Specifying and using the list form of a macro

When you specify and use the list form of a macro, you can code for reentrancy by embedding assembly statements that:

1. Allocate space on the stack (that is, use a local variable). See Figure 24 on page 21.
2. Copy the parameter field values from the list form to this allocated space.
3. Invoke the execute form of the macro that will use the allocated space.

Note: The code format string in Figure 12 on page 16 invokes the WTO macro by using the execute form of the macro with a user-defined buffer. That example does allow for reentrancy.

You should not have direct reference to symbols within your code format string as the addressability is not guaranteed. The proper way to use the macro is shown in Figure 24 on page 21, in which all __asm statements are connected through the C variable operands listmsg1 and buff.

Figure 23 on page 21 provides an example that uses the list form of a macro without considering reentrancy.

```

    1  __asm(" WTO 'hello world',MF=L" : "DS"(listmsg1));
    2
    3
int main() {
    4  __asm( " WTO  MF=(E,(%0)) " : : "r"(&listmsg1));
    5
    return 0;
}

```

Notes:

1. The first `__asm` statement invokes the macro `WTO` in the list form (`MF=L`). In order for the list form of the macro to be invoked with the values of the parameter fields defined, the `__asm` statement must be specified in the global scope.
2. The message text "hello world" is provided as a macro parameter.
3. The "DS" constraint indicates that this is a data definition, with the name of the C variable defined as the variable `listmsg1`. Because `listmsg1` is implicitly defined as a structure, it can be referenced in subsequent `__asm` statements, therefore the "DS" constraint must be specified in the output operand list. By default, the compiler allocates 256 bytes of memory for the variable `listmsg1`, which should satisfy most requirements. You can change the memory allocation size (for example, "DS:100"(`listmsg1`)). You can allocate more than 256 bytes of space.
4. The second `__asm` statement invokes the macro `WTO` in the execute form (`MF=(E,(%0))`). It takes the address of the storage defined in the list form.
5. The address of the variable `listmsg1` is defined as an input operand that is stored in a register.

Figure 23. Specifying and using the `WTO` macro (no reentrancy)

Support of reentrancy requirements: If the execute form of the macro needs to change the fields provided in the list form, the assembly statements embedded by the `__asm` statement in Figure 23 will be incorrect when support for reentrancy is required. The proper way to use the macro is shown in Figure 24.

```

__asm(" WTO 'hello world',MF=L" : "DS"(listmsg1)); 1

int main() {
    __asm(" WTO 'hello world',MF=L" : "DS"(buff)); 2
    buff = listmsg1; 3

    __asm( " WTO  MF=(E,(%0)) " : : "r"(&buff));
    return 0;
}

```

Notes:

1. The first `__asm` statement uses the list form of the macro `WTO` to define the variable `listmsg1`.
2. The second `__asm` statement, specified within function scope with a "DS" constraint, will allocate stack space for the variable `buff` but will not actually initialize the parameter values.
3. The size of this variable should match that of the corresponding `__asm` statement in global scope. An assignment copies the actual parameter values from the list form to this buffer.

Figure 24. Support for reentrancy in a code format string

Inserting non-executable HLASM statements into the generated source code

You can use the `#pragma insert_asm` directive to supply your own non-executable HLASM statements to the generated source code. The primary purpose of this directive is that you can use it to include the DSECT mapping macros that are required by your embedded assembly statements. The syntax is `#pragma insert_asm("string ")`.

The `#pragma insert_asm` directive causes the compiler to insert *string* at an appropriate place in the generated HLASM code. When you use multiple `#pragma insert_asm` directives, they are placed in the same order as they appear in your C source code.

Notes:

1. The `#pragma insert_asm` directive can be used with a `_Pragma` operator. If you use the `_Pragma` operator, you must put a slash ("/") character before the double quotation marks that surround the string literal. For example: `_Pragma ("insert_asm(\"MYSTRING\")")`.
2. If the pragma is ignored (for example, when the METAL option is not specified), the compiler emits a message.

Example: Using the #pragma insert_asm directive to map specific DSECT

information: Figure 25 uses the `#pragma insert_asm` directive to get the system CVTUSER field to address your specified CVT extension data. Because the CVTPTR and CVTUSER fields are defined in the CVT mapping macro, you can use the `#pragma insert_asm` directive to map specific DSECT information.

```
void foo() {
    void * user_cvt;
    __asm(" L 2,CVTPTR\n"
        " L 2,CVTUSER-CVT(2)\n"
        " ST 2,%0"
        : "m"(user_cvt) : "r2");
}
#pragma insert_asm(" CVT    DSECT=YES,LIST=NO")
```

Figure 25. Code that supplies specific DSECT mapping macros

Reserving a register for a global variable

The *register* storage class specifier is the C-language extension that allows you to specify, for the entire compilation unit, a general purpose register (GPR) for a global variable, as shown in Figure 26 on page 23.

When you use a code format string to specify a GPR for a global variable, be aware that:

- Only GPR 0 through GPR 15 can be specified for storage of a global variable.
- The variable must be declared as a pointer type.
- A declaration with register specifier cannot have an initializer.

For more information, see The register storage class specifier in *z/OS XL C/C++ Language Reference*.

```

register int * p 1__asm(2"r5");

```

Notes:

1. The variable declaration `int * p` defines the variable as a pointer type.
2. The register `"r5"` is not initialized.

Figure 26. Register specification

AMODE-switching support

Within a Metal C application, AMODE 31 and AMODE 64 programs can call each other.

To take advantage of the Metal C AMODE-switching support, be aware of the following information:

- The called and calling programs must be in separate source files. Mixing addressing modes within a single C source file is not supported.
- The save area format for the called program is determined by the AMODE and ASC mode of the called program, that is, 72-byte for AMODE 31 programs, F4SA for AMODE 64 programs, F7SA for AR mode programs. The ability for tracing the save areas chain will be interrupted across AMODE switches.
- The parameter list is prepared according to the AMODE of the called program, that is, 4-byte slots for AMODE 31 programs and 8-byte slots for AMODE64 programs.
- The AMODE of the called program can be specified by the new `amode31` and `amode64` type attributes. For detailed information, see The `amode31` | `amode64` type attribute in *z/OS XL C/C++ Language Reference*.
- The calling program switches the addressing mode before the call and switches back to its own addressing mode on return from the call.
- The implicit sizes of types `long` and `pointer` in the function prototype are determined by the addressing mode of the called program.

Example of an AMODE31 program that calls an AMODE64 program

In Figure 27 on page 24, AMODE 31 program `main` in `a31.c` makes calls to AMODE 64 programs `a64a1` and `a64a2` in `a64a.c`. For the commands that compile and link `a31.c` and `a64a.c`, see “Commands that compile and link applications that switch addressing modes” on page 35.

```

a31.c

long a64a1 (long j, int k, short s) __attribute__((amode64));
int a64a2 (long j, int k, short s) __attribute__((amode64));
int main () {
    int a = 40;
    return a64a1(99LL, a, 4) + a64a2(-120LL, -60, -18);
}

a64a.c

long a64a1 (long a, int b, short c) {
    return -(a+b+c);
}

int a64a2 (long a, int b, short c) {
    return -(a+b+c);
}

```

Figure 27. AMODE31 program that calls an AMODE64 program

AR-mode programming support

With the METAL option, an AR-mode function can access data stored in data spaces by using the hardware access registers. For more information about AR-mode see *z/OS MVS Programming: Assembler Services Guide*, SA22-7605. A non-AR-mode function is said to be in *primary mode*.

This information describes the compiler options, language constructs, and built-in functions that support AR-mode programming.

AR-mode function declaration

You can declare a function to be an AR-mode function with the `armode` attribute. The syntax is:

```
void armode_func() __attribute__((armode));
```

You can also use the `ARMODE` compiler option to declare that all functions in the source program to be AR-mode functions. If you use the `ARMODE` compiler option and you want to single out the functions in the source program to be primary mode functions you can declare the function with the `noarmode` attribute. The syntax is:

```
void nonarmode_func() __attribute__((noarmode));
```

Far pointer declaration, reference, and dereference

The ability to reference data stored in different data spaces is achieved through a C language extension to pointer types called far pointer types. A far pointer type is declared by adding the `__far` qualifier. The syntax is

```
int * __far my_far_pointer;
```

A far pointer can be declared in a function of any mode (AR mode or primary mode). But only an AR-mode function can directly or indirectly dereference a far pointer. In other words, only an AR-mode program can access data stored in data spaces with far pointers.

Note: For an example, see Figure 34 on page 32.

Regardless of the mode of the function, a far pointer can be manipulated in the following ways:

- It can be passed as a parameter.
- It can be received as a function return value.
- It can be compared with another pointer.
- It can be cast as another pointer type.
- It can be used in pointer arithmetic expressions.

A far pointer consists of ALET and an offset. Although an ALET is always 32 bits in length, the size of a far pointer is twice the size of a regular pointer. The layout of a far pointer in memory depends on the AMODE of the function:

- Under AMODE 31, a far pointer occupies eight bytes.
 - The ALET occupies the first four bytes.
 - The offset occupies the last four bytes.
- Under AMODE 64, a far pointer occupies 16 bytes.
 - The first four bytes are unused.
 - The ALET occupies the second four bytes.
 - The offset occupies the last eight bytes.

This difference in pointer size is illustrated in Figure 28.

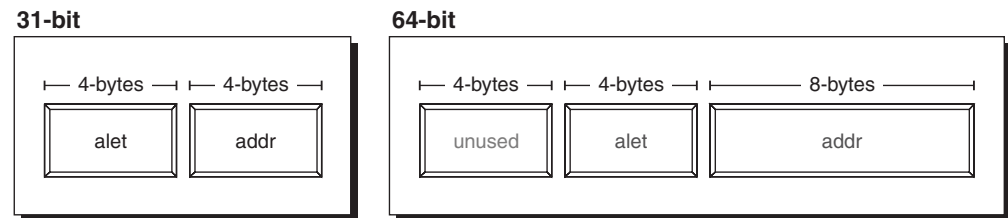


Figure 28. Far pointer sizes under different addressing modes

C language constructs and far pointers

Table 4 describes the effects of language constructs that might have special impact on far pointers.

Table 4. Language constructs that may have special impact on far pointers

Language Construct	Effect
Implicit or explicit cast from normal to far pointer	Because the normal pointer is assumed to point to primary address space, the ALET of the far pointer is set to 0.
Explicit cast from far pointer to normal pointer	The offset of the far pointer is extracted and used to form the normal pointer. Unless the ALET of the far pointer was 0, the normal pointer is likely to be invalid.
Operators !=, ==	If either operand is a far pointer, the other operand is implicitly cast to a far pointer before the operands are compared. The comparison is performed on both the ALET and offset components of a far pointer.
Operators <, <=, >, >=	Only the offset of the far pointer is used in the comparison. Unless the ALETs of the far pointers were the same, the result might be meaningless.
Compare to NULL	Because of the implicit cast of NULL to a far pointer, the != and == operators compare both the ALET and the offset to zero. A test of !(p>NULL) is not sufficient to ensure that the ALET is also 0.

Table 4. Language constructs that may have special impact on far pointers (continued)

Language Construct	Effect
Pointer arithmetic	The effects of pointer arithmetic are applied to the offset component of a far pointer only. The ALET component remains unchanged.
Address of Operator, operand of &	<p>The result is a normal pointer, except in the following cases:</p> <ul style="list-style-type: none"> If the operand of & is the result of an indirection operator (*), the type of & is the same as the operand of the indirection operator. If the operand of & is the result of the arrow operator (->, structure member access), the type of & is the same as the left operand of the arrow operator.

Implicit ALET association

In addition to explicitly specifying ALETs that use far pointers to access data in data spaces, the compiler must associate those ALETs with all the memory references contained in the AR-mode function.

In a non-AR-mode function, all variable references are to primary data space (ALET 0). In an AR-mode function, the compiler manages access registers (ARs) so that every memory reference uses an ALET associated with the variable type to reach the appropriate data space. Table 5 lists the ALET associations for different types of variables.

Table 5. Implicit ALET associations for AR-mode-function variables

Variable type	Implied ALET
File-scope variable	ALET 0 (primary data space)
Stack variables (function local variable)	The ALET that is in AR 13 at the time of function entry. This points to the stack frame.
Parameters (function formal parameters)	The ALET that is in AR 1 at the time of function entry. This points to the parameter list.
Data pointed to by regular pointers	ALET 0 (primary data space).
Data pointed by far pointer	ALET contained in far pointer.

Far pointer construction

The Metal C Runtime Library does not provide functions for allocating or deallocating alternative data spaces. You can use the DSPSERV and ALESERV HLASM macros to allocate space and obtain a valid ALET and offset. For an example, see Figure 32 on page 29. For more information, see *z/OS MVS Programming: Assembler Services Guide*, SA22-7605.

Built-in functions that manage far-pointer components

The compiler provides built-in functions for setting and getting the individual components of far pointers. Whenever you use these built-in functions, you must:

- Define the macro `_MI.BUILTIN` to "1".
- Include the header file `builtins.h`.

Figure 29 on page 27 lists the constructors.


```
void * __far __set_far_ALET_offset(unsigned int alet, void * offset);
void * __far __set_far_ALET(unsigned int alet, void * __far offset); 1
void * __far __set_far_offset(void * __far alet, void * offset); 2
```

Notes:

1. The `__set_far_ALET` function does not modify the far-pointer parameter offset. It simply uses it to provide the offset component of the far pointer being constructed. Its return value is the constructed far pointer.
2. Similarly, the `__set_far_offset` function that uses the far-pointer parameter ALET is not modified; it simply provides the ALET for the far pointer being constructed.

Figure 29. Built-in functions for setting far-pointer components

Figure 30 lists the extractors.

```
unsigned int __get_far_ALET(void * __far p);
void * __get_far_offset(void * __far p);
```

Figure 30. Built-in functions for getting far-pointer components

For information about ARMODE built-in functions, see Using hardware built-in functions in *z/OS XL C/C++ Programming Guide*.

Library functions that manipulate data stored in data spaces

The XL C compiler provides far versions of some of the standard C string and memory library functions. The far versions can be called by either AR-mode or primary-mode functions. If these functions are called by an AR mode function, the compiler will generate inline code for them.

Whenever you use these functions, you must:

- Define the macro `_MI.BUILTIN` to "1".
- Include the header file `builtins.h`.

The semantics of these functions, listed in Figure 31, are identical to the standard version.

```
void * __far __far_memcpy(void * __far s1, const void * __far s2, size_t n);
int __far memcmp(const void * __far s1, const void * __far s2, size_t n);
void * __far __far_memset(void * __far s, int c, size_t n);
void * __far __far_memchr(const void * __far s, int c, size_t n);
char * __far __far_strcpy(char * __far s1, const char * __far s2); See Note
char * __far __far_strncpy(char * __far s1, const char * __far s2, size_t n);
int __far strcmp(const char * __far s1, const char * __far s2);
int __far strncmp(const char * __far s1, const char * __far s2, size_t n);
char * __far __far_strcat(char * __far s1, const char * __far s2);
char * __far __far_strncat(char * __far s1, const char * __far s2, size_t n);
char * __far __far_strchr(const char * __far s, int c);
char * __far __far_strrchr(const char * __far s, int c);
size_t __far strlen(const char * __far s);
```

Note: For an example that illustrates the use of this function, see Figure 33 on page 31.

Figure 31. Library functions for use only in AR-mode functions

AR-mode function linkage conventions

AR mode functions follow the same linkage conventions as do primary-mode functions, with the following additional requirements:

- Any function that calls an AR-mode function must supply the 54-word F7SA save area for saving the access registers.
- The AR-mode function must preserve the calling function's access registers.
- The AR-mode function is responsible for switching into AR mode on entry and switching back to calling function's ASC mode on exit.

Note: A primary-mode function does not switch the ASC mode when calling an AR-mode function.

- An AR-mode function must switch to primary mode before calling a primary mode function.
- A far pointer is passed and returned as a struct that is based on the layout for the calling function's AMODE.

Default prolog and epilog code for AR-mode functions

If the calling function is in non-AR mode, the DSA and parameter areas are assumed to be located in the primary address space.

For AR-mode functions, the default prolog code generates additional instructions that:

- Save the calling function's access registers in the F7SA save area.
- Save the ASC mode of the calling function in the F7SA save area.
- Switch to AR mode.
- Prime AR 1 and AR 13 with LAE instructions.

For AR-mode functions, the default epilog code generates additional instructions that:

- Restore the calling function's access registers.
- Restore the ASC mode of the calling function.

Data space allocation and deallocation

Figure 32 on page 29 provides examples of routines for allocating and deallocating data space.

```

#define _MI_BUILTN 1
#include builtins.h
#include string.h

/*****
/* Allocation/Deallocation example routines */
*****/

int alloc_data_space(void * __far *ret, char dstok[8], long size_blocks, char name[8])
{
    __asm("DSPARMS DSPSERV MF=L\n" : "XL:DS:64"(DSPARMS));
    __asm("ALPARMS ALESERV MF=L\n" : "XL:DS:16"(ALPARMS));
    int res,res2;
    struct _myparms /* To reduce number of operands to __asm */
    {
        unsigned origin; /* +0 */
        unsigned blocks; /* +4 */
        unsigned alet; /* +8 */
        char name[8]; /* +12 */
        char dstok[8]; /* +20 */
    } myparms;

    strncpy(myparms.name,name,8);
    myparms.blocks = size_blocks;

    __asm(
        "        DSPSERV CREATE,GENNAME=COND,NAME=12(%1),OUTNAME=12(%1),"
        "STOKEN=20(%1),ORIGIN=0(%1),BLOCKS=(4(%1)),MF=(E,(%2))\n"
        "        ST          15,%0\n"
        : "=m"(res)
        : "a"(&myparms), "a"(&DSPARMS)
        : "r0" , "r1", "r14", "r15");

    if(res==0)
    {
        __asm(
            "        ALESERV ADD,STOKEN=20(%1),ALET=8(%1),MF=(E,(%2))\n"
            "        ST          15,%0\n"
            : "=m"(res2) : "a"(&myparms), "a"(&ALPARMS) : "r0" , "r1", "r14", "r15");

        if(res2!=0)
        {
            __asm(
                "        DSPSERV DELETE,STOKEN=20(%1),MF=(E,(%2))\n"
                "        ST          15,%0\n"
                : "=m"(res2) : "a"(&myparms), "a"(&DSPARMS) : "r0" , "r1", "r14", "r15");
            return -res2;
        }
    }
    else
    {
        return res;
    }

    *ret = __set_far_ALET_offset(myparms.alet,(void *)myparms.origin);
    strncpy(dstok,myparms.dstok,8);
    strncpy(name,myparms.name,8);
    return 0;
}

```

Figure 32. Allocation and deallocation routines (Part 1 of 3)

```

void * __far allocate_far(long size)
{
    void * __far ret;

    ret = NULL;
    if(size > 0)
    {
        int blocks = (size+4095)/4096;
        char name[8];
        char dstok[8];
        strncpy(name,"Z",8); /* provide a prefix */
        alloc_data_space(&ret, dstok, blocks, name);
    }
    return ret;
}

void delete_data_space(void * __far p, char dstok[8])
{
    __asm("DSPARMS DSPSERV MF=L\n" : "XL:DS:64"(DSPARMS));
    __asm("ALPARMS ALESERV MF=L\n" : "XL:DS:16"(ALPARMS));
    int alet;

    if(p!=NULL)
    {
        alet = __get_far_ALET(p);
        __asm(
            "ALESERV DELETE,ALET=0(%0),MF=(E,(%1))\n"
            : "a"(&alet), "a"(&ALPARMS) : "r0" , "r1", "r14", "r15");

        __asm(
            "DSPSERV DELETE,STOKEN=0(%0),MF=(E,(%1))\n"
            : "a"(dstok), "a"(&DSPARMS) : "r0" , "r1", "r14", "r15");
    }
}

int get_data_space_token(void * __far p, char *dstok)
{
    __asm("ALPARMS ALESERV MF=L\n" : "XL:DS:16"(ALPARMS));
    unsigned alet;
    int res;

    if(p!=NULL)
    {
        alet = __get_far_ALET(p);
        __asm(
            "ALESERV EXTRACT,ALET=0(%1),STOKEN=0(%2),MF=(E,(%3))\n"
            "ST 15,%0\n"
            : "=m"(res) : "a"(&alet), "a"(dstok), "a"(&ALPARMS) : "r0" , "r1", "r14", "r15");

        return res;
    }
    return -1;
}

```

Figure 32. Allocation and deallocation routines (Part 2 of 3)

```

void * __far free_far(void * __far p)
{
    int x;
    void * __far ret;

    if(p != NULL)
    {
        char dstok[8];
        x = get_data_space_token(p,dstok);
        if(x==0)
        {
            delete_data_space(p, dstok);
        }
    }
    return NULL;
}

```

Figure 32. Allocation and deallocation routines (Part 3 of 3)

Copying a string pointer to a far pointer

Figure 33 provides an example of using a built-in function to copy a C string pointer to a far pointer.

```

/*****
/* __far_strcpy example
*****/

char * __far far_strcpy_example() __attribute__((armode));
char * __far far_strcpy_example()
{
    char * __far far_string;
    char * near_string;

    near_string = "Hello World!\n";

    far_string = allocate_far(1024);

    __far_strcpy(far_string,near_string);

    return far_string; /* Assume caller will free allocated data space */
}

```

Figure 33. Copying a C string pointer to a far pointer

Far pointer dereference

The Metal C Runtime Library does not provide functions for allocating or deallocating alternative data spaces. Figure 34 provides an example of code that dereferences a far pointer.

```

/*****
/* Simple dereference example
*****/

char get_ith_character(char *__far s, int i) __attribute__((armode));
char get_ith_character(char *__far s, int i)
{
    return s[i];
}

int main()
{
    char c;
    char *__far far_string;

    far_string = far_strcpy_example();

    c = get_ith_character(far_string,1);

    free_far(far_string);

    return c;
}

```

Figure 34. Example of a simple dereference of a far pointer

Building Metal C programs

XL C programs are normally built by compiling C source files into object files that are linked into a program. When building Metal C programs, this process has an extra step: C source files are first compiled into HLASM source files and then assembled into object files, as shown in Figure 35.

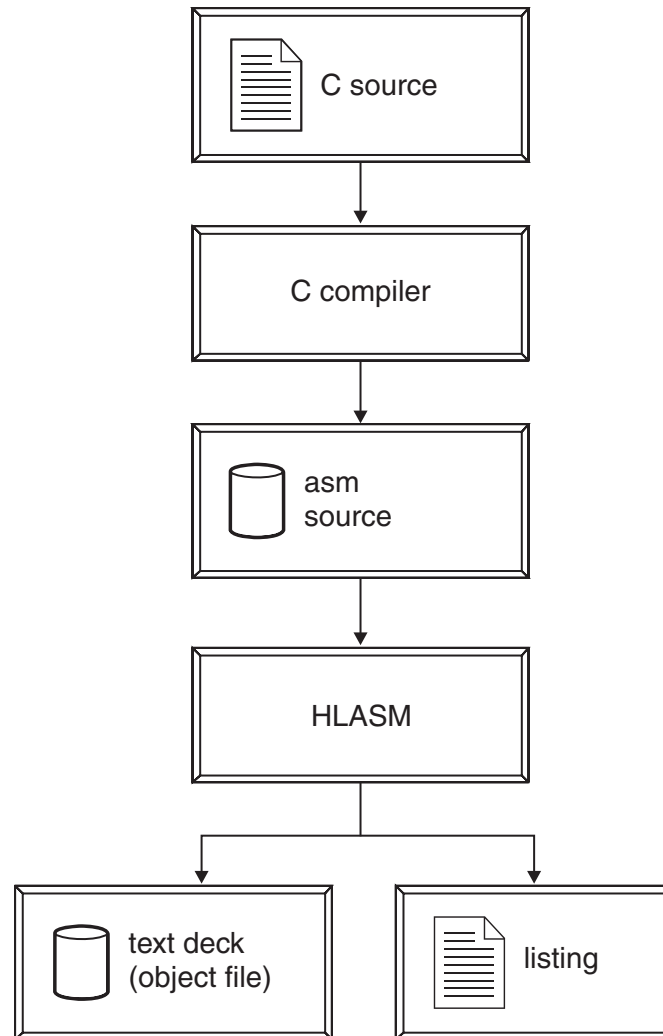


Figure 35. Metal C application build process

Examples of building Metal C applications

A set of examples illustrates how to build a Metal C program by using either z/OS UNIX System Services commands or MVS JCL procedures. In these examples:

- MYADD is the name of the entry point in the C program.
- The name of the C source file used to generate the HLASM source file is mycode.c.
- The name of the HLASM source file is mycode.s if it is generated under z/OS UNIX System Services.
- The name of the HLASM source file is HLQSOURCE.ASM(MYCODE) if HLQSOURCE.(MYCODE) is generated under MVS.

C source file

Figure 36 shows a C source file `mycode.c` that can be used to generate an HLAASM source file. The name of the generated HLAASM source file is `mycode.s` under z/OS UNIX System Services.

```
int myadd(void) {
    int a , b;
    a = 1;
    b = 2;
    __asm(" AR  %0,%1 "
          : "=r"(a)
          : "r"(b), "0"(a)
    );
    return a;
}
```

Figure 36. C source file (`mycode.c`) that builds a Metal C program

Building a Metal C program using z/OS UNIX System Services

There are three steps for building a Metal C program under z/OS UNIX System Services:

1. Use the **xlc** command to generate an HLAASM source file.
2. Use the **as** command to generate the object file.
3. Use the **ld** command to generate the program.

Generating an HLAASM source file using the xlc command: To generate an HLAASM source file from a C source file, the **xlc** command must be invoked with the **-qmetal** option and the **-S** flag.

Note: Without the **-S** flag, the **xlc** utility invokes the compiler with the **OBJECT** option, which is in conflict with the **METAL** option. This causes the compiler to emit a severe error message and stop processing.

The generated HLAASM source file has the same name as the C source file with the suffix derived from the `ssuffix` attribute in the **xlc** configuration file. The default suffix is `s`, so in the examples in this section, the HLAASM source file name is `mycode.s`.

```
xlc -S -qmetal mycode.c
```

Figure 37. C compiler invocation to generate `mycode.s`

A successful compilation will produce `mycode.s`. The compiler can generate debugging information in DWARF format even for Metal C programs. To generate DWARF debugging information, the **-g** flag must be specified when the C sources are compiled with the **METAL** option, as shown in Figure 38.

```
xlc -S -qmetal -g mycode.c
```

Figure 38. C compiler invocation to generate `mycode.s` with debugging comments

A successful compilation will produce `mycode.s` with the debugging information appended as a series of comments at the end of the file.

Generating an object file from the HLAASM source using the z/OS UNIX System Services `as` command: The generated object file does not have to be a z/OS UNIX file. The **as** command can write the object file directly to an MVS data set, as shown in Figure 39 on page 35.


```
as mycode.s -o '//HLQ.ASM.OBJ(MYCODE)'
```

Figure 39. Command that invokes HLASM to assemble mycode.s

A successful compilation will produce mycode.o.

Notes:

1. If the **-g** flag was used to compile the C source, debugging information in DWARF format can be produced by specifying the **as** command **-g** flag.
2. To specify the data set name, use the **-o** flag on the command line, as shown in Figure 39.

If the C source file was compiled with the LONGNAME compiler option, the generated HLASM source file will contain symbols that are more than eight characters in length. In that case, the HLASM GOFF option must be specified. Use the **as** utility **-m** flag to specify HLASM options, as shown in Figure 40.

```
as -mgoff mycodelong.s
```

Figure 40. Command that compiles an HLASM source file containing symbols longer than eight characters

A successful compilation will produce mycodelong.o.

Creating a program with the z/OS UNIX System Services **ld command:** Use the z/OS UNIX System Services **ld** command to link the object file produced by the **as** command into a program. The program does not have to be a z/OS UNIX file. The **ld** utility can write the program directly to a specified MVS data set.

Common **ld** command options that control the bind step are:

- **-e** to specify the entry point.
- **-o** to specify the name of the program created by the **ld** utility.
- **-V** to direct the binder listing to stdout.
- **-b** to specify other binder-specific options.

Note: If you compile your C source file with the LONGNAME option, you must use **-b case=mixed** and the **-e** option must specify the entry point in its original case, as shown in Figure 41.

```
ld -b case=mixed -e MYADD -o '//LOAD(mycode)' mycode.o
```

Figure 41. Command that binds mycode.o and produces a Metal C program in an MVS data set

A successful bind produces HLQ.LOAD(MYCODE) with entry point MYADD.

Commands that compile and link applications that switch addressing modes:

Figure 42 on page 36 shows the commands that compile and link the programs in Figure 27 on page 24.

```
xlc -S -qmetal a31.c
xlc -S -qmetal -q64 a64a.c
as -a=a31.lst -mgoff a31.s
as -a=a64a.lst -mgoff a64a.s
ld -o a.out a31.o a64a.o -e//MAIN
```

1
2
3
3
4

Notes:

1. To generate an HLASM source file from a C source file, the **xlc** command must be invoked with the **-qmetal** option and the **-S** flag.
2. The called program **a64a.c** is an external function in a separate source file.
3. The **-mgoff** command is used to compile an HLASM source file containing symbols longer than eight characters.
4. The z/OS UNIX System Services **ld** command links the object file produced by the **as** command into a program. The **-e** command specifies the entry point.

Figure 42. Commands that compile and link programs with different addressing modes

Building Metal C programs using JCL

When you build Metal C programs using JCL, you cannot use standard JCL procedures that combine the compilation step with the link step (or link and run steps) because compiling Metal C programs produces HLASM source files that must be assembled by HLASM before they can be linked.

If DWARF debugging information is not required, you can use standard HLASM JCL procedures to compile HLASM source files generated from C source files. If the DWARF debugging information is required, you can use the CDAASMC JCL procedure to assemble the HLASM source files.

After successful completion of the assembly step (with or without DEBUG processing), you can use an appropriate binder invocation JCL procedure to produce an program.

Note: Binder invocation JCL procedures are available in the CEE.SCEEPROC data set.

Compilation of HLQ.SOURCE.C(MYCODE):

```
//PROC JCLLIB ORDER=(CBC.SCCNPRC)
//*-----
//* Invoke METAL C compiler
//*-----
//METALCMP EXEC EDCC,
//      INFILE='HLQ.SOURCE.C(MYCODE)',
//      OUTFILE='HLQ.SOURCE.ASM(MYCODE),DISP=SHR',
//      CPARM='METAL'
```

Figure 43. Job step that compiles HLQ.SOURCE.C(MYCODE)

Assembly of HLQ.SOURCE.ASM(MYCODE):

```
////PROCLIB JCLLIB ORDER=(CEE.SCEEPROC)
//*****
//* ASSEMBLE USING CDAHLASM utility
//*****
//METALASM EXEC CDAASMC,
//    HPARM=' ',
//    INFILE='HLQ.SOURCE.ASM(MYCODE)',
//    OUTFILE='HLQ.OBJ(MYCODE),DISP=SHR',
//    DWARFOUT='HLQ.DWARF(MYCODE),DISP=SHR',
//    ADATAOUT='HLQ.ADATA(MYCODE),DISP=SHR'
//CDAHOPT DD *
//    PHASEID
/*
```

1
2
3

Figure 44. Job step that invokes the CDAASMC JCL procedure

Notes:

1. The DWARF debug side file is stored in DD:SYSDWARF. If the DWARFOUT option is omitted, DWARF debugging information is written into a temporary data set.
2. The ADATA debug side file is stored in DD:SYSADATA. If the ADATAOUT option is omitted, ADATA debugging information is written into a temporary data set.
3. CDAHOPT can be used to specify CDAHLASM options.

Bind of HLQ.OBJ(MYCODE) into a Metal C program:

```
//*-----
/* BIND STEP:
//*-----
//BIND EXEC PGM=IEWL,
//    PARM='AMODE=31,MAP,CASE=MIXED'
//STEPLIB DD DSN=CEE.SCEERUN2,DISP=SHR
//    DD DSN=CEE.SCEERUN,DISP=SHR
//SYSLMOD DD DSNAME=HLQ.LOAD(MYCODE),DISP=SHR
//SYSPRINT DD SYSOUT=*
//OBJECT DD DSN=HLQ.OBJ,DISP=SHR
//SYSLIN DD *
//    INCLUDE OBJECT(MYCODE)
//    ENTRY MYADD
/*
```

1

2

Figure 45. Job step that binds the generated HLASM object into a program

Notes:

1. The program is written to SYSLMOD.
2. The entry point can be specified using the ENTRY binder control statement.

Generation of debugging information

When the NOMETAL compiler option is in effect (the default), the XL C compiler either generates debugging information as a separate binary file in DWARF format, or embeds debugging information within the object file in ISD format. When the METAL compiler option is specified, debugging information in both ADATA and DWARF format can be generated. The ADATA debug format allows debugging of the generated HLASM source. The DWARF debug format allows debugging of the original C source.

CDAASMC JCL procedure to generate debugging information: The **as** command is a z/OS UNIX System Services utility that invokes the HLASM

assembler and can produce debugging information in DWARF format. CDAASMC is the JCL procedure provided with the XL C compiler to do the same thing in a batch environment.

Note: If you wish to use the HLASM ASMLANGX debugging utility, you must first assemble your source with the ADATA assembler option. The CDAASMC JCL procedure allows you to generate both ADATA and DWARF debugging information.

The cataloged CDAASMC JCL procedure invokes CDAHLASM.

Debugging information for the IDF debugger: The Interactive Debug Facility (IDF) is a symbolic debugging tool for assembly language programs. It uses information from the load module file to determine the locations of a program's control sections and external symbols.

Optionally, IDF can make use of additional information to help disassemble the program. The additional information can be generated by specifying the assembler TEST option and the linkage editor TEST option.

Note: The Linkage Editor TEST option can make the final load module file quite large. If you prefer to suppress them, either omit the linkage editor TEST option or specify the NOTEST option.

The Linkage Editor TEST option increases the size of the load module file, so do not use it for production modules.

ADATA debugging information: The ASMLANGX utility extracts source level information from the ADATA debugging information. The output is an extract file. Although you can create extract files as sequential files, they are typically stored in a PDS.

The recommended format for the extract file is:

```
RECFM(VB) LRECL(1562) BLKSIZE(27998)
```

```
//ASMLANGX EXEC PGM=ASMLANGX,REGION=4096K,  
// PARM='member (ASM LOUD ERROR'  
//SYSADATA DD DISP=SHR,DSN=h1q..SYSADATA  
//ASMLANGX DD DISP=OLD,DSN=h1q..ASMLANGX
```

1
2
3

Notes:

1. The PDS member name of the input and output file is passed as a parameter. For sequential files, this name is ignored.
2. The SYSADATA DD statement specifies the input data set name.
3. The ASMLANGX DD statement specifies the output data set name.

Figure 46. JCL that invokes the ASMLANGX utility

IDF debugger invocation

If you want to use an interactive utility to debug your program, invoke the IDF debugger by performing the following steps:

1. Specify the problem load module and the extract file that contains the debugging information by entering the following commands.

```
ALLOC FI (ASMLANGX) DS('h1q.ASMLANGX') SHR  
TSOLIB ACT DS('h1q.LOAD')
```

2. Invoke IDF by entering the following command:
ASMIDF MYCODE
3. Press F9 to get the **Program Source and Disassembly** view.

Summary of useful references for the Metal C programmer

Table 6 lists topics of interest to the Metal C programmer and, for each topic, lists information found in this document, as well as external references.

Table 6. Summary of useful references for the Metal C programmer

Information	Internal reference	External references
The base linkage conventions that are used by the generated modules.	"Metal C and MVS linkage conventions" on page 2	For detailed information about MVS linkage conventions, see <i>Linkage Conventions in z/OS MVS Programming: Assembler Services Guide</i> , SA22-7605.
The Metal C Runtime Library.	Chapter 2, "Header files," on page 41	For additional information about the Metal runtime library, see http://www.ibm.com/systems/z/zos/metalc/ .
Using assembler statements within a C program.	<ul style="list-style-type: none"> "Inserting HLASM instructions into the generated source code" on page 13 "Inserting non-executable HLASM statements into the generated source code" on page 22 	<p>For detailed information about HLASM programming, see <i>HLASM MVS & VM Programmer's Guide</i>.</p> <p>For detailed information about inline assembly statements, see <i>Inline assembly statements in z/OS XL C/C++ Language Reference</i>.</p> <p>For more information about callable system services, see <i>z/OS MVS Programming: Callable Services for High-Level Languages</i>.</p>
Using the METAL option.	"Programming with Metal C" on page 2	Note: For detailed information about the METAL option and how it interacts with other XL C compiler options, see METAL option in <i>z/OS XL C/C++ User's Guide</i> .
Making access registers available to the Metal C application.	"AR-mode programming support" on page 24	For detailed information about using access registers, see <i>z/OS MVS Programming: Extended Addressability Guide</i> .
Providing prolog and epilog code to customize the environment.	<ul style="list-style-type: none"> "Compiler-generated global SET symbols" on page 9 "Supplying your own prolog and epilog code" on page 8 	<i>Not applicable.</i>

Table 6. Summary of useful references for the Metal C programmer (continued)

Information	Internal reference	External references
Building the application by using JCL procedures.	"Building Metal C programs using JCL" on page 36	<i>Not applicable.</i>
Building the application by using z/OS UNIX System Services.	"Building a Metal C program using z/OS UNIX System Services" on page 34	<i>Not applicable.</i>
Generating the appropriate debugging information.	"Generation of debugging information" on page 37	<i>Not applicable.</i>
Invoking the IDF debugger.	"IDF debugger invocation" on page 38	For specific information about IDF, see http://www.ibm.com/software/awdtools/debugtool/ .

Chapter 2. Header files

Header files for the Metal C Runtime Library are located in the z/OS UNIX file system directory: `/usr/include/metal/`. To use these headers with a Metal C compile, you must inform the compiler that this directory is to be searched. There are a number of ways to do this.

If you are compiling in batch, you have the following options:

- Use the `SEARCH` compiler option.
`SEARCH(/usr/include/metal/)`
- Use the `PATH` keyword on a `DD` statement for the `SYSLIB` `DD` for the compiler step.
`SYSLIB DD PATH='/usr/include/metal/'`

If you are compiling in the z/OS UNIX System Services environment, specify the `NOSEARCH` compiler option along with one of the following:

- Use the `-I` option of the `xlC` utility.
`-I /usr/include/metal/`
- Use the `cinc` attribute in the `xlC` configuration file.
`cinc = /usr/include/metal/`

builtins.h

The `builtins.h` header contains a list of built-in functions supported by the compiler. A built-in function is inline code that is generated in place of an actual function call. For more information about the built-in functions, see *Using hardware built-in functions in z/OS XL C/C++ Programming Guide* and “AR-mode programming support” on page 24.

ctype.h

The `ctype.h` header file declares functions used in character classification. The `ctype.h` header file declares the following functions.

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

Note: All the functions in the previous table use code page IBM-1047.

float.h

The `float.h` header file contains definitions of constants listed in ANSI 2.2.4.2.2. The constants describe the characteristics of the internal representations of the three floating-point data types: `float`, `double`, and `long double`. Table 7 lists the definitions contained by `float.h`.

Table 7. Definitions in `float.h`

Constant	Description
<code>FLT_RADIX</code>	The radix for a z/OS XL C Metal C application. For <code>FLOAT(IEEE)</code> , the value is 2.

Table 7. Definitions in *float.h* (continued)

Constant	Description
FLT_MANT_DIG DBL_MANT_DIG LDBL_MANT_DIG	The number of hexadecimal digits stored to represent the significand of a fraction.
FLT_DIG DBL_DIG LDBL_DIG	The number of decimal digits, <i>q</i> , such that any floating-point number with <i>q</i> decimal digits can be rounded into a floating-point number with <i>p</i> radix FLT_RADIX digits, and back again, without any change to the <i>q</i> decimal digits.
FLT_MIN_10_EXP DBL_MIN_10_EXP LDBL_MIN_10_EXP	The minimum negative integer such that 10 raised to that power is in the range of normalized floating-point numbers.
FLT_MAX_EXP DBL_MAX_EXP LDBL_MAX_EXP	The maximum integer such that FLT_RADIX raised to that power minus 1 is a representable finite floating-point number.
FLT_MAX_10_EXP DBL_MAX_10_EXP LDBL_MAX_10_EXP	The maximum integer such that 10 raised to that power is in the range of representable finite floating-point numbers.
FLT_MAX DBL_MAX LDBL_MAX	The maximum representable finite floating-point number.
FLT_EPSILON DBL_EPSILON LDBL_EPSILON	The difference between 1.0 and the least value greater than 1.0 that is representable in the given floating-point type.
FLT_MIN DBL_MIN LDBL_MIN	The minimum normalized positive floating-point number.
DECIMAL_DIG	The minimum number of decimal digits needed to represent all the significant digits for type long double.
FLT_EVAL_METHOD	Describes the evaluation mode for floating point operations. This value is 1, which evaluates <ul style="list-style-type: none"> • All operations and constants of types float and double to type double. • All operations and constants of long double to type long double.

inttypes.h

The following macros are defined in *inttypes.h*. Each expands to a character string literal containing a conversion specifier which can be modified by a length modifier that can be used in the *format* argument of a formatted input/output function when converting the corresponding integer type. These macros have the general form of PRI or SCN, followed by the conversion specifier, followed by a name corresponding to a similar type name in *<inttypes.h>*. In these names, the suffix number represents the width of the type. For example, *PRIdFAST32* can be used in a format string to print the value of an integer of type **int_fast32_t**.

Compile requirement:

In the following list all macros with the suffix MAX or 64 require long long to be available.

Macros for printf family for signed integers.

PRId8	PRId16	PRId32	PRId64
PRIdLEAST8	PRIdLEAST16	PRIdLEAST32	PRIdLEAST64
PRIdFAST8	PRIdFAST16	PRIdFAST32	PRIdFAST64
PRIdMAX			
PRIdPTR			
PRId8	PRId16	PRId32	PRId64
PRIdLEAST8	PRIdLEAST16	PRIdLEAST32	PRIdLEAST64
PRIdFAST8	PRIdFAST16	PRIdFAST32	PRIdFAST64
PRIdMAX			
PRIdPTR			

Compile requirement:

In the following list all macros with the suffix MAX or 64 require long long to be available.

Macros for printf family for unsigned integers.

PRId8	PRId16	PRId32	PRId64
PRIdLEAST8	PRIdLEAST16	PRIdLEAST32	PRIdLEAST64
PRIdFAST8	PRIdFAST16	PRIdFAST32	PRIdFAST64
PRIdMAX			
PRIdPTR			
PRId8	PRId16	PRId32	PRId64
PRIdLEAST8	PRIdLEAST16	PRIdLEAST32	PRIdLEAST64
PRIdFAST8	PRIdFAST16	PRIdFAST32	PRIdFAST64
PRIdMAX			
PRIdPTR			
PRId8	PRId16	PRId32	PRId64
PRIdLEAST8	PRIdLEAST16	PRIdLEAST32	PRIdLEAST64
PRIdFAST8	PRIdFAST16	PRIdFAST32	PRIdFAST64
PRIdMAX			
PRIdPTR			
PRId8	PRId16	PRId32	PRId64
PRIdLEAST8	PRIdLEAST16	PRIdLEAST32	PRIdLEAST64
PRIdFAST8	PRIdFAST16	PRIdFAST32	PRIdFAST64
PRIdMAX			
PRIdPTR			
PRId8	PRId16	PRId32	PRId64
PRIdLEAST8	PRIdLEAST16	PRIdLEAST32	PRIdLEAST64
PRIdFAST8	PRIdFAST16	PRIdFAST32	PRIdFAST64
PRIdMAX			
PRIdPTR			

Compile requirement:

In the following list all macros with the suffix MAX or 64 require long long to be available.

Macros for sscanf family for signed integers.

SCNd8	SCNd16	SCNd32	SCNd64
SCNdLEAST8	SCNdLEAST16	SCNdLEAST32	SCNdLEAST64
SCNdFAST8	SCNdFAST16	SCNdFAST32	SCNdFAST64
SCNdMAX			
SCNdPTR			
SCNi8	SCNi16	SCNi32	SCNi64
SCNiLEAST8	SCNiLEAST16	SCNiLEAST32	SCNiLEAST64
SCNiFAST8	SCNiFAST16	SCNiFAST32	SCNiFAST64
SCNiMAX			
SCNiPTR			

Compile requirement:

In the following list all macros with the suffix MAX or 64 require long long to be available.

Macros for sscanf family for unsigned integers.

SCNo8	SCNo16	SCNo32	SCNo64
SCNoLEAST8	SCNoLEAST16	SCNoLEAST32	SCNoLEAST64
SCNoFAST8	SCNoFAST16	SCNoFAST32	SCNoFAST64
SCNoMAX			
SCNoPTR			
SCNu8	SCNu16	SCNu32	SCNu64
SCNuLEAST8	SCNuLEAST16	SCNuLEAST32	SCNuLEAST64
SCNuFAST8	SCNuFAST16	SCNuFAST32	SCNuFAST64
SCNuMAX			
SCNuPTR			
SCNx8	SCNx16	SCNx32	SCNx64
SCNxLEAST8	SCNxLEAST16	SCNxLEAST32	SCNxLEAST64
SCNxFAST8	SCNxFAST16	SCNxFAST32	SCNxFAST64
SCNxMAX			
SCNxPTR			

limits.h

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

Table 8. Definitions of Resource Limits

CHAR_BIT	8
----------	---

Table 8. Definitions of Resource Limits (continued)

CHAR_MAX	127 (_CHAR_SIGNED)
CHAR_MAX	255
CHAR_MIN	(-128) (_CHAR_SIGNED)
CHAR_MIN	0
INT_MAX	2147483647
INT_MIN	(-2147483647 - 1)
LLONG_MAX	(9223372036854775807LL)
LLONG_MIN	(-LLONG_MAX-1)
LONG_MAX	2147483647
LONGLONG_MAX	(9223372036854775807LL)
LONG_MIN	(-2147483647L - 1)
LONGLONG_MIN	(-LONGLONG_MAX - 1)
MB_LEN_MAX	4
SCHAR_MAX	127
SCHAR_MIN	(-128)
SHRT_MAX	32767
SHRT_MIN	(-32768)
SSIZE_MAX	2147483647
UCHAR_MAX	255
UINT_MAX	4294967295
ULONG_MAX	4294967295U
ULONGLONG_MAX	(18446744073709551615ULL)
ULLONG_MAX	(18446744073709551615ULL)
USHRT_MAX	65535

math.h

The math.h header file contains macro declarations for use with floating-point support:

No feature test macro is required.

Object-like Macros

HUGE_VAL

A very large positive number that expands to a double expression.

HUGE_VALF

A very large positive number that expands to a float expression.

HUGE_VALL

A very large positive number that expands to a long double expression.

INFINITY

A constant expression of type float representing positive infinity.

NAN

A constant expression of type float representing a quiet NaN.

metal.h

The metal.h header file contains function prototypes and data definitions related to the Metal C runtime library, including the `__cinit()` and `__cterm()` functions.

The metal.h header file also includes `__csysenv_s`, which is the structure used to describe the characteristics of a Metal C environment. For more information about the `__csysenv_s` structure, see “`__cinit()` - Initialize a Metal C environment” on page 55.

Note: The metal.h header file is automatically included by any Metal C runtime library header file, so it is not necessary to explicitly include it if a header file is being used.

stdarg.h

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

<code>va_arg()</code>	<code>va_copy()</code>	<code>va_start()</code>	<code>va_end()</code>
-----------------------	------------------------	-------------------------	-----------------------

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

The stdarg.h header file defines `va_list` as `char *va_list`.

stddef.h

The stddef.h header file defines the following types:

ptrdiff_t

The signed long type of the result of subtracting two pointers.

size_t

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

ssize_t

similar to `size_t`, but must be a signed type.

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

stdio.h

The stdio.h header file declares the following functions.

<code>snprintf()</code>	<code>sprintf()</code>	<code>sscanf()</code>	<code>vsnprintf()</code>	<code>vsprintf()</code>
<code>vsscanf()</code>				

Macros defined in stdio.h

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

`NULL` A pointer which never points to a data object.

stdint.h

The `stdint.h` header defines integer types, limits of specified width integer types, limits of other integer types, and macros for integer constant expressions.

Note: For the exact width integer types, minimum-width integer types, limits of specified width integer types, and exact width integer constants, *bit sizes* N with the values 8, 16, 32, and 64 are supported.

Requirement: Use of the bit size 64 and greatest-width integer types or macros require LP64 or the long long data type to be available.

Integer types

The following exact width integer types are defined.

- `intN_t`
- `uintN_t`

The following minimum-width integer types are defined.

- `int_leastN_t`
- `uint_leastN_t`

The following fastest minimum-width integer types are defined. These types are the fastest to operate with among all integer types that have at least the specified width.

- `int_fastN_t`
- `uint_fastN_t`

The following greatest-width integer types are defined. These types hold the value of any signed/unsigned integer type.

Note: Requires long long to be available.

- `intmax_t`
- `uintmax_t`

The following integer types capable of holding object pointers are defined.

- `intptr_t`
- `uintptr_t`

Object-like macros for limits of integer types

Macros for limits of exact width integer types.

- `INTN_MAX`
- `INTN_MIN`
- `UINTN_MAX`

Macros for limits of minimum width integer types.

- `INT_LEASTN_MAX`
- `INT_LEASTN_MIN`
- `UINT_LEASTN_MAX`

Macros for limits of fastest minimum width integer types.

- INT_FASTN_MAX
- INT_FASTN_MIN
- UINT_FASTN_MAX

Macros for limits of greatest width integer types.

- INTMAX_MAX
- INTMAX_MIN
- UINTMAX_MAX

Macros for limits of pointer integer types.

- INTPTR_MAX
- INTPTR_MIN
- UINTPTR_MAX

Macros for limits of ptrdiff_t.

- PTRDIFF_MAX
- PTRDIFF_MIN

Macro for limit of size_t.

- SIZE_MAX

Function-like macros for integer constants

Macros for minimum width integer constants.

- INTN_C(value)
- UINTN_C(value)

Macros for greatest-width integer constants:

- INTMAX_C(value)
- UINTMAX_C(value)

stdlib.h

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

abs()[1]	atoi()	atol()	atoll()	calloc()
div()	free()	labs()	ldiv()	llabs()
lldiv()	malloc()	__malloc31()	rand()	rand_r()
realloc()	srand()	strtod()	strtof()	strtol()
strtold()	strtoll()	strtoul()	strtoull()	

[1] Built-in function.

Two type definitions are added to stdlib.h for the Compare and Swap functions cs() and cds(). The structures defined are cs_t and cds_t.

The type size_t is declared in the header file. It is used for the type of the value returned by sizeof. For more information on the types size_t, see “stddef.h” on page 46.

The `stdlib.h` declares `div_t`, `ldiv_t`, and `lldiv_t`, which define the structure types that are returned by `div()`, `ldiv()`, and `lldiv()`.

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

<code>NULL</code>	The <code>NULL</code> pointer constant (also defined in <code>stddef.h</code>).
<code>RAND_MAX</code>	Expands to an integer representing the largest number that the <code>rand()</code> or <code>rand_r()</code> function can return.

string.h

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

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

[1] Built-in function.

The `string.h` header file also defines the macro `NULL` and the type `size_t`. For more information see “`stddef.h`” on page 46.

Chapter 3. C functions available to Metal C programs

This chapter describes the Metal C runtime library functions.

The linkage conventions used by the XL C METAL compiler option govern use of the C functions that are available to XL C-compiled freestanding programs. For more information, see “Metal C and MVS linkage conventions” on page 2.

When you use any of these supplied C functions, be aware of the information provided in “Characteristics of compiler-generated HLASM source code” on page 4.

Characteristics of Metal C runtime library functions

Linkage to each function is through the default linkage provided by the METAL option of the C compiler. This assumes that GPR 13 points to a stack frame in a contiguous stack, and that the forward pointer in the stack frame contains the address of the next available byte in the stack. The stack frame requirements for each function are documented in Appendix A, “Function stack requirements,” on page 107 so that the caller knows how much space to reserve.

The library functions support AMODE 31 and AMODE 64.

The library functions (with the exception of a few AR mode supporting functions) expect the ASC mode to be Primary on entry. The AR mode support part of Metal C ensures that this is enforced; however, if calling these library functions from within HLASM embedded statements or their own HLASM programs, you need to manage ASC mode to meet this requirement.

The library functions support IEEE floating point numbers.

The library uses code page IBM-1047 and the En_US locale definitions to perform its functions.

System and static object libraries

The Metal C runtime library supports two versions of its library functions: a system library and a static object library. The behavior of the functions within the two versions is the same. What differs is where the functions are located and how the Metal C application interacts with them.

System library

The system library is a version of the Metal C runtime library that exists within the system's link pack area, and is made available during the system IPL process. It is suggested that you use the system library if the Metal C application is run on a level of z/OS that supports the runtime library, and the application runs after the library has been made available. This library has the added advantage of not requiring application module re-links when service is applied to the library.

To use the system library version, simply include the desired Metal C runtime library headers in the Metal C application source code. The default behavior of the headers is to generate code within the application that calls this system library. No additional binding is needed in order for these function calls to work.

Static object library

The Static object library is a version of the Metal C runtime library that gets directly bound with a Metal C application load module. The resulting application is self-contained with respect to the library; all library function calls from the application result in the functions bound within the load module to be driven.

It is suggested that you use the static object library if the Metal C application meets either of the following requirements:

- The application is run on a supported level of z/OS that does not support the system library (before z/OS V1.9).
- The application is run during system IPL before the system library has been made available.

The static object library functions are provided in two system data sets: SYS1.SCCR3BND and SYS1.SCCR6BND. SYS1.SCCR3BND is used with Metal C applications that have been compiled using ILP32 and run AMODE 31. SYS1.SCCR6BND is used with Metal C applications that have been compiled using LP64 and run AMODE 64.

In order to use the static object library, you must take the following steps:

1. Define the `__METAL_STATIC` feature test macro before including the headers in your Metal C program, and then compile the program. For example:

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

This will cause library function calls in the program to generate external references to the functions contained within the SCCRnBND data sets.

2. Bind the compiled object with the corresponding SCCRnBND data set. How this is done depends on the environment in which the binding takes place:

- Batch: When using the binder from a batch job, use the CALL option, and use the SYSLIB DD to identify the static object library data set that you want to bind with.
- Unix System Services shell: From the shell, it is suggested that the ld shell command be used to bind the application with the library functions. This avoids conflicts with the Language Environment stubs that the c89 family of commands may introduce. Use the -S option to identify the static object library data set that you want to bind with. For example:

```
-S //'SYS1.SCCR3BND''
```

Note: When service is applied to the static object library, the Metal C application must be re-linked to pick up the changes.

General library usage notes

- A Metal C application can use either the system library or the static object library, but not both. The mixing of system library calls and static object library calls within the same application is not supported.
- All static objects bound to the application load module must be at compatible service levels.
- Metal C runtime library functions are not supported under Language Environment and must not be used within a Language Environment program, because equivalent functions are already available.

abs() — Calculate integer absolute value

Format

```
#include <stdlib.h>

int abs(int n);
```

General description

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

For the integer version of `abs()`, the minimum allowable integer is `INT_MIN+1`. (`INT_MIN` is a macro that is defined in the `limits.h` header file.) For example, with the Metal C compiler, `INT_MIN+1` is `-2147483647`.

Returned value

The returned value is the absolute value, if the absolute value is possible to represent.

Otherwise the input value is returned.

Related Information

- “limits.h” on page 44
- “stdlib.h” on page 48
- “labs() — Calculate long absolute value” on page 63

atoi() — Convert character string to integer

Format

```
#include <stdlib.h>

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

General description

The `atoi()` function converts the initial portion of the string pointed to by *nptr* to a ‘int’. This is equivalent to

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

Returned value

If successful, `atoi()` returns the converted int value represented in the string.

If unsuccessful, `atoi()` returns an undefined value.

Related Information

- “stdlib.h” on page 48
- “atol() — Convert character string to long” on page 54
- “atoll() — Convert character string to signed long long” on page 54
- “strtol() — Convert Character String to Long” on page 95
- “strtoll() — Convert String to Signed Long Long” on page 98
- “strtoul() — Convert String to Unsigned Integer” on page 99
- “strtoull() — Convert String to Unsigned Long Long” on page 100

atol() — Convert character string to long

Format

```
#include <stdlib.h>

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

General description

The `atol()` function converts the initial portion of the string pointed to by `nptr` to a 'long int'. This is equivalent to `strtol(nptr, (char **)NULL, 10)`

Returned value

If successful, `atol()` returns the converted long int value represented in the string.

If unsuccessful, `atol()` returns an undefined value.

Related Information

- “`stdlib.h`” on page 48
- “`atoi()` — Convert character string to integer” on page 53
- “`atoll()` — Convert character string to signed long long”
- “`strtol()` — Convert Character String to Long” on page 95
- “`strtoll()` — Convert String to Signed Long Long” on page 98
- “`strtoul()` — Convert String to Unsigned Integer” on page 99
- “`strtoull()` — Convert String to Unsigned Long Long” on page 100

atoll() — Convert character string to signed long long

Format

```
#define _ISOC99_SOURCE
#include <stdlib.h>
long long atoll(const char *nptr);
```

Compile Requirement

Use of this function requires the long long data type. See *z/OS XL C/C++ Language Reference* for information on how to make long long available.

General description

The `atoll()` function converts the initial portion of the string pointed to by `nptr` to a '**long long** int'. This is equivalent to `strtoll(nptr, (char **)NULL, 10)`.

Returned value

If successful, `atoll()` returns the converted signed **long long** value, represented in the string. If unsuccessful, it returns an undefined value.

Related Information

- “`stdlib.h`” on page 48
- “`atoi()` — Convert character string to integer” on page 53
- “`atol()` — Convert character string to long”
- “`strtol()` — Convert Character String to Long” on page 95
- “`strtoll()` — Convert String to Signed Long Long” on page 98

- “strtoul() — Convert String to Unsigned Integer” on page 99
- “strtoull() — Convert String to Unsigned Long Long” on page 100

calloc() — Reserve and initialize storage

Format

```
#include <stdlib.h>

void *calloc(size_t num, size_t size);
```

General 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.

The `calloc()` function returns a pointer to the reserved space. The storage space to which the returned value points is aligned for storage of any type of object.

Note: Use of this function requires that an environment has been set up through the `__cinit()` function. When the function is called, GPR 12 must contain the environment token created by the `__cinit()` call.

Returned value

If successful, `calloc()` returns the pointer to the area of memory reserved.

If there is not enough space to satisfy the request or if *num* or *size* is 0, `calloc()` returns NULL.

Related Information

- “stdlib.h” on page 48
- “free() — Free a block of storage” on page 59
- “malloc() — Reserve storage block” on page 65
- “__malloc31() — Allocate 31-bit storage” on page 65
- “realloc() — Change reserved storage block size” on page 70

__cinit() - Initialize a Metal C environment

Format

```
#include <metal.h>

__csysenv_t __cinit(struct __csysenv_s * csysenv);
```

General description

The `__cinit()` function establishes a Metal C environment based on the characteristics in the input *csysenv* structure. This environment is used when calling Metal C functions that require an environment, such as those related to storage management (`malloc()`, `free()`, and so on). Storage for the environment structures is obtained by using the attributes specified in the input *csysenv* structure.

Use of this function requires the long long data type. See *z/OS XL C/C++ Language Reference* for information about how to make long long data type available

The environment token created by __cinit() can be used from both AMODE 31 and AMODE 64 programs. Calls to __malloc31() always affect the below-the-bar heap. Calls made while in AMODE 31 to all other functions that obtain storage affect the below-the-bar heap; calls made while in AMODE 64 affect the above-the-bar heap.

Table 9. csysenv argument in __cinit()

Argument	Description
csysenv	A structure describing the characteristics of the environment to be created.

The details on the csysenv (__csysenv_s) structure is shown as follows:

```

struct __csysenv_s {
    int      __cseversion;      /* Control block version number      */
                                /* Must be set to __CSE_VERSION_1    */

    int      __cseheap31init;   /* for 31 bit storage                 */
    __ptr31(void, __csetcbowner) /* owning TCB for resources           */
                                /* default: TCB mode - caller tcb,   */
                                /* SRB. XMEM - CMRO TCB              */

    int      __cseheap31inc;    /* Reserved field                     */
    char      __csettkowner[16]; /* TCB token of owning TCB for       */
                                /* above the bar storage              */
                                /* default: caller tcbtoken          */
                                /* SRB mode: tcbtoken must be       */
                                /* specified                         */

    unsigned int
        __cseheap31initsize; /* Minimum size, in bytes, to obtain */
                                /* for the initial AMODE 31 heap storage. */
                                /* If 0, defaults to 32768 bytes        */

    unsigned int
        __cseheap31incsize; /* Minimum size, in bytes, to obtain */
                                /* when expanding the AMODE 31 heap.    */
                                /* If 0, defaults to 32768 bytes        */

#ifdef __LL
    unsigned long long
        __cseheap64init; /* Minimum size, in MB, to obtain */
                                /* for the initial AMODE 64 heap storage. */
                                /* If 0, defaults to 1 MB              */

    unsigned long long
        __cseheap64inc; /* Minimum size, in MB, to obtain */
                                /* When expanding the AMODE 64 heap.    */
                                /* If 0, defaults to 1MB              */

    unsigned long long
        __cseheap64usertoken; /* usertoken for use with ?iarv64 */
                                /* to obtain above the bar storage      */

#else
    unsigned int
        __cseheap64init_hh;
    unsigned int
        __cseheap64initsize; /* Minimum size, in MB, to obtain */
                                /* for the initial AMODE 64 heap storage. */
                                /* If 0, defaults to 1 MB              */

    unsigned int
        __cseheap64incsize_hh;
    unsigned int
        __cseheap64incsize; /* Minimum size, in MB, to obtain */
                                /* When expanding the AMODE 64 heap.    */
                                /* If 0, defaults to 1MB              */

    unsigned int
        __cseheap64usertoken_hh;
    unsigned int

```

```

        __cseheap64usertoken; /* usertoken for use with ?iarv64
                               to obtain above the bar storage */
#endif

    unsigned int          /* AMODE 64 Storage Attributes */
        __cseheap64fprot:1, /* On, AMODE 64 heap storage is to be
                               fetch protected
                               Off, storage is not fetch
                               protected */
        __cseheap64cntlauth:1; /* On, AMODE 64 heap storage has
                               CONTROL=AUTH attribute
                               Off, storage is CONTROL=UNAUTH */
    int          __csereserved1[7]; /* Reserved for future use */
};

```

Note: The entire __csysenv_s structure must be cleared to binary zeros before initializing specific fields within it.

Returned value

If successful, __cinit() returns an environment token that is used on subsequent calls to Metal C functions that require an environment. If unable to create an environment, __cinit() returns 0.

Example

```

#include <metal.h>
#include <stdlib.h>
#include <string.h>

#ifdef _LP64
    register void * myenvtkn __asm("r12");
#else
    register void * myenvtknr12 __asm("r12");
    __csysenv_t    myenvtkn;
#endif

void mymtlfcn(void) {
    struct __csysenv_s mysysenv;

    void * mystg;
    void * my31stg;

    /******
    /* Initialize the csysenv structure.
    /******
    memset(&mysysenv, 0x00, sizeof(mysysenv));

    mysysenv.__cseversion = __CSE_VERSION_1;

    mysysenv.__csubpool = 129;

    /******
    /* Set heap initial and increment sizes.
    /******
    mysysenv.__cseheap31initsize = 131072;
    mysysenv.__cseheap31incrsz = 8192;
    mysysenv.__cseheap64initsize = 20;
    mysysenv.__cseheap64incrsz = 1;
#ifdef _LP64
    /******
    /* Create a Metal C environment.
    /******
    myenvtkn = (void * ) __cinit(&mysysenv);

```

```

#else
/*****
/* Create a Metal C environment.          */
*****/
myenvtkn = __cinit(&mysysenv);

/*****
/* Save the high half of R12 and then set R12 to  */
/* the 8 byte environment token.                */
*****/
__asm(" LG      12,%0\n"
      :
      : "m"(myenvtkn)
      : "r12"
      );
#endif

/*****
/* Call functions that require an environment.  */
*****/
mystg = malloc(1048576);
my31stg = __malloc31(100);

/*****
/* Clean up the environment.                  */
*****/
__cterm((__csysenv_t) myenvtkn);
}

```

In order to share the environment token to other source files there are 2 options:

- Compile all Metal C files that make up the program by using the `RESERVED_REG("r12")` compiler option. This reserves register 12 so that the environment token will remain untouched by the compiled code.
- Pass `myenvtkn` by using other methods, and for any source that needs to use the environment token declare the global register variable as in this example and assign the environment token to it.

Output None.

__cterm() - Terminate a Metal C environment

Format

```

#include <metal.h>
void __cterm(__csysenv_t csysenvtkn);

```

General description

The `__cterm()` function terminates a Metal C environment, freeing all resources obtained on behalf of the environment.

Table 10. *csysenvtkn* argument in `__cterm()`

Argument	Description
<i>csysenvtkn</i>	The environment token representing the environment to be terminated.

Returned value

None.

Example

See the example provided for the `__cinit()` function.

Output None.

div() — Calculate quotient and remainder

Format

```
#include <stdlib.h>

div_t div(int numerator, int denominator);
```

General description

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

Returned value

The `div()` function returns a structure of type `div_t`, containing both the quotient `int quot` and the remainder `int rem`. This structure is defined in `stdlib.h`. If the returned value cannot be represented, the behavior of `div()` is undefined. If *denominator* is 0, a divide by 0 exception is raised.

Related Information

- “`stdlib.h`” on page 48
- “`ldiv()` — Compute quotient and remainder of integral division” on page 63
- “`lldiv()` — Compute quotient and remainder of integral division for long long type” on page 64

free() — Free a block of storage

Format

```
#include <stdlib.h>

void free(void *ptr);
```

General description

The `free()` function frees a block of storage pointed to by *ptr*. The *ptr* variable points to a block previously reserved with a call to `calloc()`, `malloc()`, or `realloc()`. The number of bytes freed is the number of bytes specified when you reserved (or reallocated, in the case of `realloc()`), the block of storage. If *ptr* is `NULL`, `free()` simply returns without freeing anything. Since *ptr* is passed by value `free()` will not set *ptr* to `NULL` after freeing the memory to which it points.

Notes:

1. Use of this function requires that an environment has been set up by using the `__cinit()` function. When the function is called, GPR 12 must contain the environment token created by the `__cinit()` call.
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 an `abend`.

free

Returned value

`free()` returns no value.

Related Information

- “`stdlib.h`” on page 48
- “`calloc()` — Reserve and initialize storage” on page 55
- “`malloc()` — Reserve storage block” on page 65
- “`__malloc31()` — Allocate 31-bit storage” on page 65
- “`realloc()` — Change reserved storage block size” on page 70

`isalnum()` to `isxdigit()` — Test integer value

Format

```
#include <ctype.h>

int isalnum(int c);
int isalpha(int c);
int isblank(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
```

General description

The functions listed in the previous section, which are all declared in `ctype.h`, test a given integer value. The valid integer values for *c* are those representable as an *unsigned char* or EOF.

Here are descriptions of each function in this group.

<code>isalnum()</code>	Test for an upper- or lowercase letter, or a decimal digit, as defined by code page IBM-1047.
<code>isalpha()</code>	Test for an alphabetic character, as defined by code page IBM-1047.
<code>isblank()</code>	Test for a blank character, as defined by code page IBM-1047.
<code>iscntrl()</code>	Test for any control character, as defined by code page IBM-1047.
<code>isdigit()</code>	Test for a decimal digit, as defined by code page IBM-1047.
<code>isgraph()</code>	Test for a printable character excluding space, as defined by code page IBM-1047.
<code>islower()</code>	Test for a lowercase character, as defined by code page IBM-1047.
<code>isprint()</code>	Test for a printable character including space, as defined by code page IBM-1047.
<code>ispunct()</code>	Test for any non-alphanumeric printable character, excluding space, as defined by code page IBM-1047.
<code>isspace()</code>	Test for a white space character, as defined by code page IBM-1047.

<code>isupper()</code>	Test for an uppercase character, as defined by code page IBM-1047.
<code>isxdigit()</code>	Test for a hexadecimal digit, as defined by code page IBM-1047.

Returned value

If the integer satisfies the test condition, these functions return nonzero.

If the integer does not satisfy the test condition, these functions return 0.

Related Information

- “ctype.h” on page 41
- “tolower(), toupper() — Convert Character Case” on page 102

isalpha() — Test for alphabetic character classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 60.

isblank() — Test for blank character classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 60.

iscntrl() — Test for control classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 60.

isdigit() — Test for decimal-digit classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 60.

isgraph() — Test for graphic classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 60.

islower() — Test for lowercase

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 60.

isprint() — Test for printable character classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 60.

ispunct() — Test for punctuation classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 60.

isspace() — Test for space character classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 60.

isupper() — Test for uppercase letter classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 60.

isxdigit() — Test for hexadecimal digit Classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 60.

labs() — Calculate long absolute value

Format

```
#include <stdlib.h>

long int labs(long int n);
```

General description

The `labs()` function calculates the absolute value of its long integer argument *n*. The result is undefined when the argument is equal to `LONG_MIN`, the smallest available long integer (-2 147 483 648). The value `LONG_MIN` is defined in the `limits.h` header file.

Returned value

The `labs()` function returns the absolute value of the long integer argument *n*.

Related Information

- “`limits.h`” on page 44
- “`stdlib.h`” on page 48
- “`abs()` — Calculate integer absolute value” on page 53
- “`llabs()` — Calculate absolute value of long long integer” on page 64

ldiv() — Compute quotient and remainder of integral division

Format

```
#include <stdlib.h>

ldiv_t ldiv(long int numerator, long int denominator);
```

General description

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

Returned 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 returned value is undefined. If *denominator* is 0, a divide by 0 exception is raised.

Related Information

- “`stdlib.h`” on page 48
- “`div()` — Calculate quotient and remainder” on page 59
- “`lldiv()` — Compute quotient and remainder of integral division for long long type” on page 64

llabs() — Calculate absolute value of long long integer

Format

```
#include <stdlib.h>

long long llabs(long long int n);
```

Compile Requirement

Use of this function requires the long long data type. See *z/OS XL C/C++ Language Reference* for information on how to make long long available.

General description

The llabs() function calculates the absolute value of its long long integer argument *n*. The result is undefined when the argument is equal to LONGLONG_MIN, the smallest available long long integer (-9 223 372 036 854 775 808). The value LONGLONG_MIN is defined in the limits.h header file.

Returned value

The llabs() function returns the absolute value of the long long integer argument *n*.

Related Information

- “stdlib.h” on page 48
- “limits.h” on page 44
- “abs() — Calculate integer absolute value” on page 53
- “labs() — Calculate long absolute value” on page 63

lldiv() — Compute quotient and remainder of integral division for long long type

Format

```
#include <stdlib.h>

lldiv_t lldiv (long long number, long long denom);
```

Compile Requirement

Use of this function requires the long long data type. See *z/OS XL C/C++ Language Reference* for information on how to make long long available.

General description

The lldiv() function calculates the quotient and remainder of the division of numerator by denominator.

Returned value

The lldiv() function returns a structure of type lldiv_t, containing both the quotient long long quot and the remainder long long rem.

If the value cannot be represented, the returned value is undefined. If *denominator* is 0, a divide by 0 exception is raised.

Related Information

- “stdlib.h” on page 48
- “div() — Calculate quotient and remainder” on page 59
- “ldiv() — Compute quotient and remainder of integral division” on page 63

malloc() — Reserve storage block

Format

```
#include <stdlib.h>

void *malloc(size_t size);
```

General description

The malloc() function reserves a block of storage of *size* bytes. Unlike the calloc() function, the content of the storage allocated is indeterminate. The storage to which the returned value points is always aligned for storage of any type of object.

Note: Use of this function requires that an environment has been set up through the __cinit() function. When the function is called, GPR12 must contain the environment token created by the __cinit() call.

Returned value

If successful, malloc() returns a pointer to the reserved space.

If not enough storage is available, or if *size* was specified as 0, malloc() returns NULL.

Related Information

- “stdlib.h” on page 48
- “calloc() — Reserve and initialize storage” on page 55
- “free() — Free a block of storage” on page 59
- “__malloc31() — Allocate 31-bit storage”
- “realloc() — Change reserved storage block size” on page 70

__malloc31() — Allocate 31-bit storage

Format

```
#include <stdlib.h>

void *__malloc31(size_t size);
```

General description

The __malloc31() function reserves a block of storage of *size* bytes from 31-bit addressable storage. The content of the storage allocated is indeterminate. The storage space to which the returned value points is always suitably aligned for storage of any type of object.

Note: Use of this function requires that an environment has been set up by using the __cinit() function. When the function is called, GPR 12 must contain the environment token created by the __cinit() call.

Returned value

If successful, __malloc31() returns a pointer to the reserved space.

If not enough storage is available, or if *size* was specified as 0, __malloc31() returns NULL.

Related Information

- “stdlib.h” on page 48
- “calloc() — Reserve and initialize storage” on page 55
- “free() — Free a block of storage” on page 59
- “malloc() — Reserve storage block” on page 65
- “realloc() — Change reserved storage block size” on page 70

memccpy() — Copy bytes in memory

Format

```
#include <string.h>

void *memccpy(void *__restrict__s1, const void *__restrict__s2, int c, size_t n);
```

General description

The memccpy() function copies bytes from memory area *s2* into memory area *s1*, stopping after the first occurrence of byte *c* (converted to an unsigned char) is copied, or after *n* bytes are copied, whichever comes first.

Returned value

If successful, memccpy() returns a pointer to the byte after the copy of *c* in *s1*.

If *c* was not found in the first *n* bytes of *s2*, memccpy() returns a NULL pointer.

Related Information

- “string.h” on page 49
- “memchr() — Search buffer”
- “memcmp() — Compare bytes” on page 67
- “memcpy() — Copy buffer” on page 68
- “memmove() — Move buffer” on page 68
- “memset() — Set buffer to value” on page 69
- “strcpy() — Copy String” on page 85

memchr() — Search buffer

Format

```
#include <string.h>

void *memchr(const void *buf, int c, size_t count);
```

General description

The memchr() built-in function searches the first *count* bytes pointed to by *buf* for the first occurrence of *c* converted to an unsigned character. The search continues until it finds *c* or examines *count* bytes.

Returned value

If successful, `memchr()` returns a pointer to the location of *c* in *buf*.

If *c* is not within the first *count* bytes of *buf*, `memchr()` returns NULL.

Related Information

- “string.h” on page 49
- “memcpy() — Copy bytes in memory” on page 66
- “memcmp() — Compare bytes”
- “memcpy() — Copy buffer” on page 68
- “memmove() — Move buffer” on page 68
- “memset() — Set buffer to value” on page 69
- “strchr() — Search for Character” on page 84

memcmp() — Compare bytes

Format

```
#include <string.h>

int memcmp(const void *buf1, const void *buf2, size_t count);
```

General description

The `memcmp()` built-in function compares the first *count* bytes of *buf1* and *buf2*.

The relation is determined by the sign of the difference between the values of the leftmost first pair of bytes that differ. The values depend on EBCDIC encoding. This function is *not* locale sensitive.

Returned value

Indicates the relationship between *buf1* and *buf2* as follows:

Value	Meaning
< 0	The contents of the buffer pointed to by <i>buf1</i> less than the contents of the buffer pointed to by <i>buf2</i>
= 0	The contents of the buffer pointed to by <i>buf1</i> identical to the contents of the buffer pointed to by <i>buf2</i>
> 0	The contents of the buffer pointed to by <i>buf1</i> greater than the contents of the buffer pointed to by <i>buf2</i>

Related Information

- “string.h” on page 49
- “memcpy() — Copy bytes in memory” on page 66
- “memchr() — Search buffer” on page 66
- “memcpy() — Copy buffer” on page 68
- “memmove() — Move buffer” on page 68
- “memset() — Set buffer to value” on page 69
- “strcmp() — Compare Strings” on page 85

memcpy() — Copy buffer

Format

```
#include <string.h>

void *memcpy(void * __restrict __dest, const void * __restrict __src, size_t count);
```

General description

The `memcpy()` built-in function copies *count* bytes from the object pointed to by *src* to the object pointed to by *dest*. For `memcpy()`, the source characters may be overlaid if copying takes place between objects that overlap. Use the `memmove()` function to allow copying between objects that overlap.

Returned value

The `memcpy()` function returns the value of *dest*.

Related Information

- “string.h” on page 49
- “memccpy() — Copy bytes in memory” on page 66
- “memchr() — Search buffer” on page 66
- “memmove() — Move buffer”
- “memset() — Set buffer to value” on page 69
- “strcpy() — Copy String” on page 85

memmove() — Move buffer

Format

```
#include <string.h>

void *memmove(void *dest, const void *src, size_t count);
```

General description

The `memmove()` function copies *count* bytes from the object pointed to by *src* to the object pointed to by *dest*. The function allows copying between possibly overlapping objects as if the *count* bytes of the object pointed to by *src* must first be copied into a temporary array before being copied to the object pointed to by *dest*.

Returned value

The `memmove()` function returns the value of *dest*.

Related Information

- “string.h” on page 49
- “memccpy() — Copy bytes in memory” on page 66
- “memchr() — Search buffer” on page 66
- “memcpy() — Copy buffer”
- “memset() — Set buffer to value” on page 69

memset() — Set buffer to value

Format

```
#include <string.h>

void *memset(void *dest, int c, size_t count);
```

General description

The `memset()` built-in function sets the first *count* bytes of *dest* to the value *c* converted to an unsigned int.

Returned value

`memset()` returns the value of *dest*.

Related Information

- “string.h” on page 49
- “memcpy() — Copy bytes in memory” on page 66
- “memchr() — Search buffer” on page 66
- “memcpy() — Copy buffer” on page 68
- “memmove() — Move buffer” on page 68

rand() — Generate random number

Format

```
#include <stdlib.h>

int rand(void);
```

General Description

The `rand()` function generates a pseudo-random integer in the range 0 to `RAND_MAX`. Use the `srand()` function before calling `rand()` to set a seed for the random number generator. If you do not make a call to `srand()`, the default seed is 1.

Note: Use of this function requires that an environment has been set up by using the `__cinit()` function. When the function is called, GPR 12 must contain the environment token created by the `__cinit()` call.

Returned Value

The `rand()` function returns the calculated value.

Related Information

- “stdlib.h” on page 48
- “rand_r() — Pseudo-random number generator”
- “srand() — Set Seed for rand() Function” on page 78

rand_r() — Pseudo-random number generator

Format

```
#include <stdlib.h>

int rand_r(unsigned int *seed);
```

General Description

The `rand_r()` function generates a sequence of pseudo-random integers in the range 0 to **RAND_MAX**. (The value of the **RAND_MAX** macro will be at least 32767.)

If `rand_r()` is called with the same initial value for the object pointed to by *seed* and that object is not modified between successive returns and calls to `rand_r()`, the same sequence shall be generated.

Returned Value

The `rand_r()` function returns a pseudo-random integer.

Related Information

- “`stdlib.h`” on page 48
- “`rand()` — Generate random number” on page 69
- “`srand()` — Set Seed for `rand()` Function” on page 78

realloc() — Change reserved storage block size

Format

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size);
```

General 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 give all bits of each element an initial value of 0.

If *size* is 0 and *ptr* is not `NULL`, the storage pointed to by *ptr* is freed and `NULL` is returned.

If you use `realloc()` with a pointer that does not point to a *ptr* created previously by `malloc()`, `calloc()`, or `realloc()`, or if you pass *ptr* to storage already freed, you get undefined behavior—typically an exception.

If you ask for more storage, the contents of the extension are undefined and are not guaranteed to be 0.

The storage to which the returned value points is aligned for storage of any type of object.

Note: Use of `realloc()` requires that an environment has been set up by using the `__cinit()` function. When the function is called, GPR 12 must contain the environment token created by the `__cinit()` call.

Returned Value

If successful, `realloc()` returns a pointer to the reallocated storage block. The storage location of the block might be moved. Thus, the returned value is not necessarily the same as the *ptr* argument to `realloc()`.

The returned value is `NULL` if *size* is 0. If there is not enough storage to expand the block to the given size, the original block is unchanged and a `NULL` pointer is returned.

Related Information

- “`stdlib.h`” on page 48
- “`calloc()` — Reserve and initialize storage” on page 55
- “`free()` — Free a block of storage” on page 59
- “`malloc()` — Reserve storage block” on page 65
- “`__malloc31()` — Allocate 31-bit storage” on page 65

snprintf() — Format and write data

Format

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

```
int snprintf(char *__restrict__ s, size_t n, const char *__restrict__ format, ...);
```

General Description

The `snprintf()` function formats and writes output to an array (specified by argument *s*). If *n* is zero, nothing is written, and *s* may be a null pointer. Otherwise, output characters beyond the *n*-1st are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array. If copying takes place between objects that overlap, the behavior is undefined.

Note: Use of `snprintf()` requires that an environment has been set up by using the `__cinit()` function. When the function is called, GPR 12 must contain the environment token created by the `__cinit()` call.

Returned Value

The `snprintf()` function returns the number of characters that would have been written had *n* been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than *n*.

Related Information

- “`stdio.h`” on page 46
- “`sprintf()` — Format and Write Data” on page 72
- “`scanf()` — Read and Format Data” on page 78

sprintf() — Format and Write Data

Format

```
#include <stdio.h>

int sprintf(char *__restrict__buffer, const char *__restrict__format-string, ...);
```

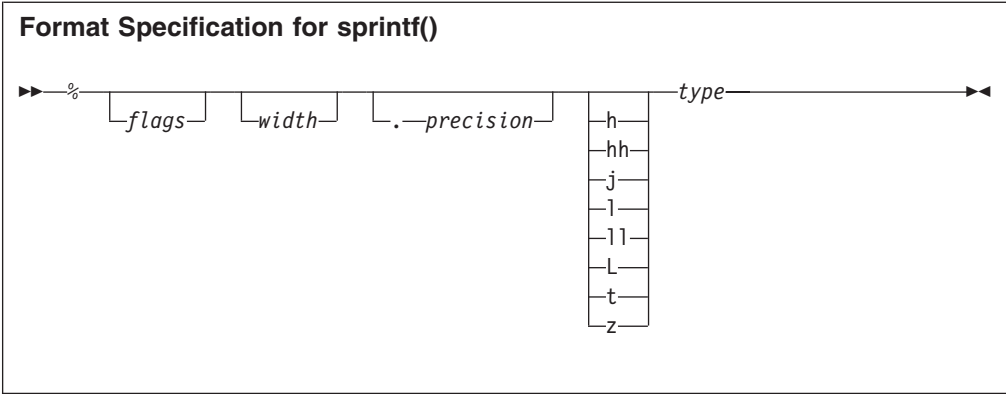
General Description

The `sprintf()` function formats and stores a series of characters and values in the array pointed to by *buffer*. Any *argument-list* is converted and put out according to the corresponding format specification in the *format-string*. If the strings pointed to by *buffer* and *format-string* overlap, behavior is undefined.

The *format-string* consists of ordinary characters, escape sequences, and conversion specifications. The ordinary characters are copied in order of their appearance. Conversion specifications, beginning with a percent sign (%) or the sequence (%n\$) where n is a decimal integer in the range [1,NL_ARGMAX], determine the output format for any *argument-list* following the *format-string*. The *format-string* can contain multibyte characters beginning and ending in the initial shift state. When the *format-string* includes the use of the optional prefix *ll* to indicate the size expected is a long long datatype then the corresponding value in the argument list should be a long long datatype if correct output is expected.

- If the %n\$ conversion specification is found, the value of the nth *argument* after the *format-string* is converted and output according to the conversion specification. Numbered arguments in the argument list can be referenced from *format-string* as many times as required.
- The *format-string* can contain either form of the conversion specification, that is, % or %n\$ but the two forms cannot be mixed within a single *format-string* except that %% can be mixed with the %n\$ form. When numbered conversion specifications are used, specifying the 'nth' argument requires that the first to (n-1)th arguments are specified in the *format-string*.

The *format-string* is read from left to right. When the first format specification is found, the value of the first *argument* after the *format-string* is converted and output according to the format specification. The second format specification causes the second *argument* after the *format-string* to be converted and output, and so on through the end of the *format-string*. If there are more arguments than there are format specifications, the extra arguments are evaluated and ignored. The results are undefined if there are not enough arguments for all the format specifications. The format specification is illustrated below.



Each field of the format specification is a single character or number signifying a particular format option. The *type* character, which appears after the last optional format field, determines whether the associated argument is interpreted as a character, a string, a number, or pointer. The simplest format specification contains only the percent sign and a *type* character (for example, `%s`).

The percent sign

If a percent sign (`%`) is followed by a character that has no meaning as a format field, the character is simply copied to the *buffer*. For example, to print a percent sign character, use `%%`.

The flag characters

The *flag* characters in Table 11 are used for the justification of output and printing of thousands of grouping characters, signs, blanks, decimal-points, octal prefixes, and hexadecimal prefixes. Note that more than one *flag* can appear in a format specification. This is an optional field.

Table 11. Flag Characters for `sprintf()` Family

Flag	Meaning	Default
'	The integer portion of the result of a decimal conversion (<code>%i,%d,%u, %f,%g</code> or <code>%G</code>) will be formatted with the thousands' grouping characters.	No grouping.
–	Left-justify the result within the field width.	Right-justify.
+	Prefix the output value with a sign (+ or –) if the output value is of a signed type.	Sign appears only for negative signed values (–).
<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.

Table 11. Flag Characters for sprintf() Family (continued)

Flag	Meaning	Default
#	When used with the o, x, or X formats, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively. For o conversion, it increases the precision, if necessary, to force the first digit of the result to be a zero. If the value and precision are both 0, a single 0 is printed. For e, E, f, F, g, and G conversion specifiers, the result always contains a decimal-point, even if no digits follow the decimal-point. Without this flag, a decimal-point appears in the result of these conversions only if a digit follows it. For g and G conversion specifiers, do not remove trailing zeros from the result as they normally are. For other conversion specifiers, the behavior is undefined.	No prefix.
0	When used with the d, i, o, u, x, X, e, E, f, F, g, or G conversion specifiers, leading zeros are used to pad to the field width. If the 0 and - flags both appear, the 0 flag is ignored. For d, i, o, u, x, and X conversion specifiers, if a precision is specified, the 0 flag is ignored. If the 0 and ' flags both appear, the grouping characters are inserted before zero padding. For other conversions, the behavior is undefined.	Space padding.

The code point for the # character varies between the EBCDIC encoded character sets. The Metal C runtime library expects the # character to use the code point for encoded character set IBM-1047.

The # flag should not be used with c, d, i, u, s, or p types.

The Width of the Output

Width is a nonnegative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the specified *width*, blanks are added on the left or the right (depending on whether the - flag is specified) until the minimum width is reached.

Width never causes a value to be truncated; if the number of characters in the output value is greater than the specified *width*, or *width* is not given, all characters of the value are output (subject to the *precision* specification).

The *width* specification can be an asterisk (*); if it is, an argument from the argument list supplies the value. The *width* argument must precede the value being formatted in the argument list. This is an optional field.

If *format-string* contains the %n\$ form of conversion specification, *width* can be indicated by the sequence *m\$, where m is a decimal integer in the range [1,NL_ARGMAX] giving the position of an integer argument in the argument list containing the field width.

The Precision of the Output

The *precision* specification is a nonnegative decimal integer preceded by a period. It specifies the number of characters to be output, or the number of decimal places. Unlike the *width* specification, the *precision* can cause truncation of the output value.

The *precision* specification can be an asterisk (*); if it is, an argument from the argument list supplies the value. The *precision* argument must precede the value being formatted in the argument list. The *precision* field is optional.

If *format-string* contains the %n\$ form of conversion specification, *precision* can be indicated by the sequence *m\$, where m is a decimal integer in the range [1,NL_ARGMAX] giving the position of an integer argument in the argument list containing the field precision.

The interpretation of the *precision* value and the default when the *precision* is omitted depend upon the *type*, as shown in Table 12.

Table 12. Precision Argument in sprintf()

Type	Meaning	Default
d i o u x X	<i>Precision</i> specifies the minimum number of digits to be output. If the number of digits in the argument is less than <i>precision</i> , the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds <i>precision</i> .	Default <i>precision</i> is 1. If <i>precision</i> is 0, or if the period (.) appears without a number following it, the <i>precision</i> is set to 0. When <i>precision</i> is 0, conversion of the value zero results in no characters.
c	No effect.	The character is output.
s	<i>Precision</i> specifies the maximum number of characters to be output. Characters in excess of <i>precision</i> are not output.	Characters are output until a NULL character is encountered.
e E f F	<i>Precision</i> specifies the number of digits to be output after the decimal-point. The last digit output is rounded.	Default <i>precision</i> is 6. If <i>precision</i> is 0 or the period appears without a number following it, no decimal-point is output.
g G	<i>Precision</i> specifies the maximum number of significant digits output.	All significant digits are output.

Optional prefix

Used to indicate the size of the argument expected:

- h A prefix with the integer types d, i, o, u, x, X means the integer is 16 bits long.
- hh Specifies that a following d, i, o, u, x, or X conversion specifier applies to a signed char or unsigned char argument (the argument will have been promoted according to the integer promotions, but its value shall be

converted to signed char or unsigned char before printing); or that a following n conversion specifier applies to a pointer to a signed char argument.

- j Specifies that a following d, i, o, u, x, or X conversion specifier applies to an intmax_t or uintmax_t argument; or that a following n conversion specifier applies to a pointer to an intmax_t argument.
- 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 the integer types d, i, o, u, x, X means the integer is 64 bits long.
- L A prefix with e, E, f, g, or G types that specifies that the argument is long double.
- t Specifies that a following d, i, o, u, x, or X conversion specifier applies to a ptrdiff_t or the corresponding unsigned type argument; or that a following n conversion specifier applies to a pointer to a ptrdiff_t argument.
- z Specifies that a following d, i, o, u, x, or X conversion specifier applies to a size_t or the corresponding signed integer type argument; or that a following n conversion specifier applies to a pointer to a signed integer type corresponding to a size_t argument.

Table 13 below shows the meaning of the type characters used in the precision argument.

Table 13. Type Characters and their Meanings

Type	Argument	Output Format
d, i	Integer	Signed decimal integer.
u	Integer	Unsigned decimal integer.
o	Integer	Unsigned octal integer.
x	Integer	Unsigned hexadecimal integer, using abcdef.
X	Integer	Unsigned hexadecimal integer, using ABCDEF.
c	Character	Single character.
s	String	Characters output up to the first NULL character (\0) or until <i>precision</i> is reached.
n	Pointer to integer	Number of characters successfully output so far to the <i>stream</i> or buffer; this value is stored in the integer whose address is given as the argument.
p	Pointer	Pointer to void converted to a sequence of printable characters. See the individual system reference guides for the specific format.

Table 13. Type Characters and their Meanings (continued)

Type	Argument	Output Format
f, F	Double	<p>Signed value having the form [-]dddd.dddd, where dddd is one or more decimal digits. The number of digits before the decimal-point depends on the magnitude of the number. The number of digits after the decimal-point is equal to the requested precision. If the precision is explicitly zero and no # is present, no decimal-point appears. If a decimal-point appears, at least one digit appears before it.</p> <p>Convert a double argument representing an infinity in [+/-]inf: a plus or minus sign with the character sequence inf, followed by a white space character (space, tab, or newline), a NULL character (\0), or EOF.</p> <p>Convert a double argument representing a NaN in one of the styles:</p> <ul style="list-style-type: none"> [+/-]nan(n) for a signaling nan. [+/-]nanq(n) for a quiet nan, where n is an integer and $1 \leq n \leq \text{INT_MAX}-1$. <p>The value of n is determined by the fraction bits of the NaN argument value. For a signaling NaN value, NaN fraction bits are reversed (left to right) to produce bits (right to left) of an even integer value, 2^n. For a quiet NaN value, NaN fraction bits are reversed (left to right) to produce bits (right to left) of an odd integer value, 2^n-1.</p> <p>The F conversion specifier produces INF, NANS, or NANQ instead of inf, nan, or nanq respectively.</p>
e, E	Double	<p>Signed value having the form [-]d.dddde[sign]ddd:</p> <ul style="list-style-type: none"> d is a single-decimal digit. dddd is one or more decimal digits. ddd is 2 or more decimal digits. sign is + or -. <p>If the <i>precision</i> is zero and no # flag is present, no decimal-point appears. The conversion specifier produces a number with E instead of e to introduce the exponent.</p> <p>A double argument representing an infinity or NaN is converted in the style of an f or F conversion specifier.</p>
g, G	Double	<p>Signed value output in f or e format (or in the F or E format in the case of a G conversion specifier). The e or E format is used only when the exponent of the value is less than -4 or greater than or equal to the <i>precision</i>. Trailing zeros are truncated, and the decimal-point appears only if one or more digits follow it or a # flag is present.</p> <p>A double argument representing an infinity or NaN is converted in the style of an f or F conversion specifier.</p>

Returned Value

If successful, sprintf() returns the number of characters output. The ending NULL character is not counted.

If unsuccessful, sprintf() returns a negative value.

Related Information

- “stdio.h” on page 46
- “snprintf() — Format and write data” on page 71
- “scanf() — Read and Format Data”

srand() — Set Seed for rand() Function

Format

```
#include <stdlib.h>

void srand(unsigned int seed);
```

General Description

The `srand()` function uses its argument *seed* as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to `rand()`. If `srand()` is not called, the `rand()` seed is set as if `srand(1)` was called at program start. Any other value for *seed* sets the generator to a different starting point. The `rand()` function generates pseudo-random numbers.

Some people find it convenient to use the return value of the `time()` function as the argument to `srand()`, as a way to ensure random sequences of random numbers.

Note: Use of `srand()` requires that an environment has been set up by using the `__cinit()` function. When the function is called, GPR 12 must contain the environment token created by the `__cinit()` call.

Returned Value

`srand()` returns no values.

Related Information

- “stdlib.h” on page 48
- “rand() — Generate random number” on page 69
- “rand_r() — Pseudo-random number generator” on page 69

sscanf() — Read and Format Data

Format

```
#include <stdio.h>

int sscanf(const char *__restrict__buffer, const char *__restrict__format-string, ...);
```

General Description

The `sscanf()` function reads data from *buffer* into the locations given by argument-list. If the strings pointed to by *buffer* and *format-string* overlap, behavior is undefined.

Each entry in the argument list must be a pointer to a variable of a type that matches the corresponding conversion specification in *format-string*. If the types do not match, the results are undefined.

The *format-string* controls the interpretation of the argument list. The *format-string* can contain multibyte characters beginning and ending in the initial shift state.

The format string pointed to by *format-string* can contain one or more of the following:

- White space characters, as specified by `isspace()`, such as blanks and newline characters. A white space character causes `sscanf()` to read, but not to store, all consecutive white space characters in the input up to the next character that is not white space. One white space character in *format-string* matches any combination of white space characters in the input.
- Characters that are not white space, except for the percent sign character (%). A non-white space character causes `sscanf()` to read, but not to store, a matching non-white space character. If the next character in the input stream does not match, the function ends.
- Conversion specifications which are introduced by the percent sign (%) or the sequence (%n\$) where n is a decimal integer in the range [1,NL_ARGMAX]. A conversion specification causes `sscanf()` to read and convert characters in the input into values of a conversion specifier. The value is assigned to an argument in the argument list.

`sscanf()` reads *format-string* from left to right. Characters outside of conversion specifications are expected to match the sequence of characters in the input stream; the matched characters in the input stream are scanned but not stored. If a character in the input stream conflicts with *format-string*, the function ends, terminating with a “matching” failure. The conflicting character is left in the input stream as if it had not been read.

When the first conversion specification is found, the value of the first *input field* is converted according to the conversion specification and stored in the location specified by the first entry in the argument list. The second conversion specification converts the second input field and stores it in the second entry in the argument list, and so on through the end of *format-string*.

When the %n\$ conversion specification is found, the value of the *input field* is converted according to the conversion specification and stored in the location specified by the nth argument in the argument list. Numbered arguments in the argument list can only be referenced once from *format-string*.

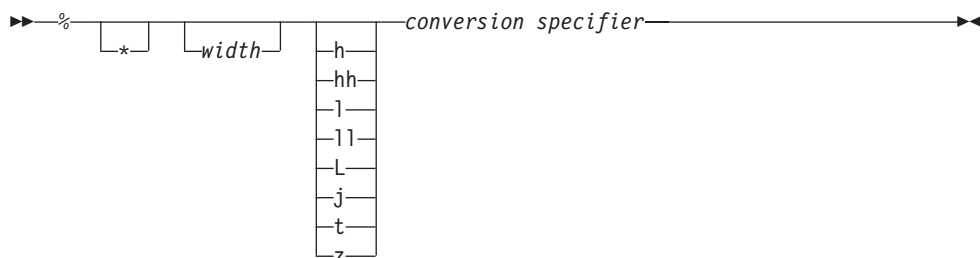
The *format-string* can contain either form of the conversion specification, that is, % or %n\$ but the two forms cannot be mixed within a single *format-string* except that %% or %* can be mixed with the %n\$ form.

An *input field* is defined as:

- All characters until a white space character (space, tab, or newline) is encountered
- All characters until a character is encountered that cannot be converted according to the conversion specification
- All characters until the field *width* is reached.

If there are too many arguments for the conversion specifications, the extra arguments are evaluated but otherwise ignored. The results are undefined if there are not enough arguments for the conversion specifications.

Syntax of Conversion Specification for sscanf()



Each field of the conversion specification is a single character or a number signifying a particular format option. The *conversion specifier*, which appears after the last optional format field, determines whether the input field is interpreted as a character, a string, or a number. The simplest conversion specification contains only the percent sign and a *conversion specifier* (for example, %s).

Each field of the format specification is discussed in detail below.

Other than conversion specifiers, avoid using the percent sign (%), except to specify the percent sign: %%. Currently, the percent sign is treated as the start of a conversion specifier. Any unrecognized specifier is treated as an ordinary sequence of characters. If, in the future, z/OS XL C/C++ permits a new conversion specifier, it could match a section of your format string, be interpreted incorrectly, and result in undefined behavior. See Table 14 on page 81 for a list of conversion specifiers.

An asterisk (*) following the percent sign suppresses assignment of the next input field, which is interpreted as a field of the specified *conversion specifier*. The field is scanned but not stored.

width is a positive decimal integer controlling the maximum number of characters to be read. No more than *width* characters are converted and stored at the corresponding *argument*.

Fewer than *width* characters are read if a white space character (space, tab, or newline), or a character that cannot be converted according to the given format occurs before *width* is reached.

The optional prefix l shows that you use the long version of the following *conversion specifier*, while the prefix h indicates that the short version is to be used. The corresponding *argument* should point to a long or double object (for the l character), a long double object (for the L character), or a short object (with the h character). The l and h modifiers can be used with the d, i, o, x, and u *conversion specifiers*. The l and h modifiers are ignored if specified for any other *conversion specifier*.

Optional prefix

Used to indicate the size of the argument expected:

h Specifies that a following d, i, o, u, x, X, or n conversion specifier applies to an argument with type pointer to short or unsigned short.

l	hh	Specifies that a following d, i, o, u, x, X or n conversion specifier applies to an argument with type pointer to signed char or unsigned char.
	j	Specifies that a following d, i, o, u, x, X or n conversion specifier applies to an argument with type pointer to intmax_t or uintmax_t.
l	l	Specifies that a following e, E, f, F, g, or G conversion specifier applies to an argument with type pointer to double.
	ll	Specifies that a following d, i, o, u, x, X or n conversion specifier applies to an argument with type pointer to long long or unsigned long long.
l	L	Specifies that a following e, E, f, g, or G conversion specifier applies to an argument with type pointer to long double.
	t	Specifies that a following d, i, o, u, x, X or n conversion specifier applies to an argument with type pointer to ptrdiff_t or the corresponding unsigned type.
l	z	Specifies that a following d, i, o, u, x, X or n conversion specifier applies to an argument with type pointer to size_t or the corresponding signed integer type.

The *type* characters and their meanings are in Table 14.

Table 14. Conversion Specifiers in *sscanf()*

Conversion Specifier	Type of Input Expected	Type of Argument
d	Decimal integer	Pointer to int
o	Octal integer	Pointer to unsigned int
x X	Hexadecimal integer	Pointer to unsigned int
i	Decimal, hexadecimal, or octal integer	Pointer to int
u	Unsigned decimal integer	Pointer to unsigned int
c	Sequence of one or more characters as specified by field width; white space characters that are ordinarily skipped are read when %c is specified. No terminating null is added.	Pointer to char large enough for input field.
s	Like c, a sequence of bytes of type char (signed or unsigned), except that white space characters are not allowed, and a terminating null is always added.	Pointer to character array large enough for input field, plus a terminating NULL character (\0) that 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 either fscanf() or to scanf().

Table 14. Conversion Specifiers in sscanf() (continued)

Conversion Specifier	Type of Input Expected	Type of Argument
p	Pointer to void converted to series of characters. For the specific format of the input, see the individual system reference guides.	Pointer to void.
[<p>A non-empty sequence of bytes to be matched against a set of expected bytes (the <i>scanset</i>), which form the conversion specification. White space characters that are ordinarily skipped are read when %[is specified.</p> <p>Consider the following situations:</p> <p>[^bytes]. In this case, the scanset contains all bytes that do not appear between the circumflex and the right square bracket.</p> <p>[abc] or [^]abc.] In both these cases the right square bracket is included in the scanset (in the first case:]abc and in the second case, <i>not</i>]abc)</p> <p>[a–z] In EBCDIC The – is in the scanset, the characters b through y are <i>not</i> in the scanset; in ASCII The – is <i>not</i> in the scanset, the characters b through y are.</p> <p>The code point for the square brackets ([and]) and the caret (^) vary among the EBCDIC encoded character sets. The default C locale expects these characters to use the code points for encoded character set Latin-1 / Open Systems 1047. Conversion proceeds one byte at a time: there is no conversion to wide characters.</p>	Pointer to the initial byte of an array of char, signed char, or unsigned char large enough to accept the sequence and a terminating byte, which will be added automatically.
e	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	Pointer to float
E		
f	(e or E) followed by a possibly signed integer value.	
F		
g		
G		

The format string passed to sscanf() must be encoded as IBM-1047.

To read strings not delimited by space characters, substitute a set of characters in square brackets ([]) for the s (string) conversion specifier. The corresponding input field is read up to the first character that does not appear in the bracketed character set. If the first character in the set is a logical not (~), the effect is reversed: the input field is read up to the first character that does appear in the rest of the character set.

To store a string without storing an ending NULL character (\0), use the specification %ac, where a is a decimal integer. In this instance, the c conversion specifier means that the argument is a pointer to a character array. The next a characters are read from the input stream into the specified location, and no NULL character is added.

The input for a %x conversion specifier is interpreted as a hexadecimal number.

The `sscanf()` 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 is one, is considered unread and is the first character of the next input field or the first character in subsequent read operations on the input stream.

The `sscanf` family functions match `e`, `E`, `f`, `F`, `g` or `G` conversion specifiers to floating-point number substrings in the input stream. The `sscanf` family functions convert each input substring matched by an `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier to a float, double or long double value depending on the size modifier before the `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier.

Many z/OS Metal C formatted input functions, including the `sscanf` family of functions, use the IEEE binary floating-point format and recognize special infinity and NaN floating-point number input sequences.

- The special sequence for infinity input is `[+/-]inf` or `[+/-]INF`, where `+` or `-` is optional.
- The special sequence of NaN input is either `[+/-]nan(n)` for a signaling nan or `[+/-]nanq(n)` for a quiet nan, where `n` is an integer and `1 <= n <= INT_MAX-1`. If `(n)` is omitted, `n` is assumed to be 1. The value of `n` determines what IEEE binary floating-point NaN fraction bits are produced by the formatted input functions. For a signaling NaN, these functions produce NaN fraction bits (left to right) by reversing the bits (right to left) of the even integer value `2*n`. For a quiet NaN, they produce NaN fraction bits (left to right) by reversing the bits (right to left) of the odd integer value `2*n-1`.

Returned Value

The `sscanf()` function returns the number of input items that were successfully matched and assigned. The returned value does not include conversions that were performed but not assigned (for example, suppressed assignments). The functions return EOF if there is an input failure before any conversion, or if EOF is reached before any conversion. Thus a returned value of 0 means that no fields were assigned: there was a matching failure before any conversion.

Related Information

- “`stdio.h`” on page 46
- “`snprintf()` — Format and write data” on page 71
- “`sprintf()` — Format and Write Data” on page 72
- “`vsnprintf()` — Format and print data to fixed length buffer” on page 104
- “`vsscanf()` — Format Input of a STDARG Argument List” on page 105

strcat() — Concatenate Strings

Format

```
#include <string.h>

char *strcat(char * __restrict__string1, const char * __restrict__string2);
```

General Description

The `strcat()` built-in function concatenates *string2* with *string1* and ends the resulting string with the NULL character. In other words, `strcat()` appends a copy of the string

strcat

pointed to by *string2*—including the terminating NULL byte— to the end of a string pointed to by *string1*, with its last byte (that is, the terminating NULL byte of *string1*) overwritten by the first byte of the appended string.

Do not use a literal string for a *string1* value, although *string2* may be a literal string.

If the storage of *string1* overlaps the storage of *string2*, the behavior is undefined.

Returned Value

The `strcat()` built-in function returns the value of *string1*, the concatenated string.

Related Information

- “string.h” on page 49
- “strchr() — Search for Character”
- “strcmp() — Compare Strings” on page 85
- “strcpy() — Copy String” on page 85
- “strcspn() — Compare Strings” on page 86
- “strncat() — Concatenate Strings” on page 87

strchr() — Search for Character

Format

```
#include <string.h>

char *strchr(const char *string, int c);
```

General Description

The `strchr()` built-in function finds the first occurrence of *c* converted to `char`, in the string **string*. The character *c* can be the NULL character (`\0`); the ending NULL character of *string* is included in the search.

The `strchr()` function operates on NULL-terminated strings. The string argument to the function *must* contain a NULL character (`\0`) marking the end of the string.

Returned Value

If successful, `strchr()` returns a pointer to the first occurrence of *c* (converted to a character) in *string*.

If the character is not found, `strchr()` returns a NULL pointer.

Related Information

- “string.h” on page 49
- “memchr() — Search buffer” on page 66
- “strcat() — Concatenate Strings” on page 83
- “strcmp() — Compare Strings” on page 85
- “strcpy() — Copy String” on page 85
- “strcspn() — Compare Strings” on page 86
- “strncmp() — Compare Strings” on page 88
- “strpbrk() — Find Characters in String” on page 89
- “strrchr() — Find Last Occurrence of Character in String” on page 90
- “strspn() — Search String” on page 90

strcmp() — Compare Strings

Format

```
#include <string.h>

int strcmp(const char *string1, const char *string2);
```

General Description

The `strcmp()` built-in function compares the string pointed to by *string1* to the string pointed to by *string2*. The string arguments to the function must contain a NULL character (`\0`) marking the end of the string.

The relation between the strings is determined by subtracting: *string1*[*i*] – *string2*[*i*], as *i* increases from 0 to *strlen* of the smaller string. The sign of a nonzero return value is determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type `unsigned char`) that differ in the strings being compared. This function is *not* locale-sensitive.

Returned Value

`strcmp()` returns a value indicating the relationship between the strings, as listed below.

Value	Meaning
< 0	String pointed to by <i>string1</i> less than string pointed to by <i>string2</i>
= 0	String pointed to by <i>string1</i> equivalent to string pointed to by <i>string2</i>
> 0	String pointed to by <i>string1</i> greater than string pointed to by <i>string2</i>

Related Information

- “string.h” on page 49
- “memcmp() — Compare bytes” on page 67
- “strcspn() — Compare Strings” on page 86
- “strncmp() — Compare Strings” on page 88
- “strpbrk() — Find Characters in String” on page 89
- “strrchr() — Find Last Occurrence of Character in String” on page 90
- “strspn() — Search String” on page 90

strcpy() — Copy String

Format

```
#include <string.h>

char *strcpy(char * __restrict__ string1, const char * __restrict__ string2);
```

General Description

The `strcpy()` built-in function copies *string2*, including the ending NULL character, to the location specified by *string1*. The *string2* argument to `strcpy()` must contain a NULL character (`\0`) marking the end of the string. You cannot use a literal string for a *string1* value, although *string2* may be a literal string. If the two objects overlap, the behavior is undefined.

strcpy

Returned Value

The `strcpy()` function returns the value of *string1*.

Related Information

- “string.h” on page 49
- “memcpy() — Copy buffer” on page 68
- “strcat() — Concatenate Strings” on page 83
- “strchr() — Search for Character” on page 84
- “strcmp() — Compare Strings” on page 85
- “strcspn() — Compare Strings”
- “strncpy() — Copy String” on page 89
- “strpbrk() — Find Characters in String” on page 89
- “strrchr() — Find Last Occurrence of Character in String” on page 90
- “strspn() — Search String” on page 90

strcspn() — Compare Strings

Format

```
#include <string.h>

size_t strcspn(const char *string1, const char *string2);
```

General Description

The `strcspn()` function computes the length of the initial portion of the string pointed to by *string1* that contains no characters from the string pointed to by *string2*.

Returned Value

The `strcspn()` function returns the calculated length of the initial portion found.

Related Information

- “string.h” on page 49
- “strcat() — Concatenate Strings” on page 83
- “strchr() — Search for Character” on page 84
- “strcmp() — Compare Strings” on page 85
- “strcpy() — Copy String” on page 85
- “strncmp() — Compare Strings” on page 88
- “strpbrk() — Find Characters in String” on page 89
- “strrchr() — Find Last Occurrence of Character in String” on page 90
- “strspn() — Search String” on page 90

strdup() — Duplicate a String

Format

```
#include <string.h>

char *strdup(const char *string);
```

General Description

The `strdup()` function creates a duplicate of the string pointed to by *string*.

Note: Use of this function requires that an environment has been set up by using the `__cinit()` function. When the function is called, GPR 12 must contain the environment token created by the `__cinit()` call.

Returned Value

If successful, `strdup()` returns a pointer to a new string which is a duplicate of *string*.

Otherwise, `strdup()` returns a NULL pointer.

Note: The caller of `strdup()` should free the storage obtained for the string.

Related Information

- “string.h” on page 49
- “free() — Free a block of storage” on page 59
- “malloc() — Reserve storage block” on page 65

strlen() — Determine String Length

Format

```
#include <string.h>

size_t strlen(const char *string);
```

General Description

The `strlen()` built-in function determines the length of string pointed to by *string*, excluding the terminating NULL character.

Returned Value

The `strlen()` function returns the length of *string*.

Related Information

- “string.h” on page 49
- “strncat() — Concatenate Strings”
- “strncmp() — Compare Strings” on page 88
- “strncpy() — Copy String” on page 89

strncat() — Concatenate Strings

Format

```
#include <string.h>

char *strncat(char * __restrict_string1, const char * __restrict_string2, size_t count);
```

General Description

The `strncat()` built-in function appends the first *count* characters of *string2* to *string1* and ends the resulting string with a NULL character (`\0`). If *count* is greater than the length of *string2*, `strncat()` appends only the maximum length of *string2* to *string1*. The first character of the appended string overwrites the terminating NULL character of the string pointed to by *string1*.

If copying takes place between overlapping objects, the behavior is undefined.

Returned Value

The `strncat()` function returns the value *string1*, the concatenated string.

Related Information

- “string.h” on page 49
- “strcat() — Concatenate Strings” on page 83
- “strncmp() — Compare Strings”
- “strncpy() — Copy String” on page 89
- “strpbrk() — Find Characters in String” on page 89
- “strrchr() — Find Last Occurrence of Character in String” on page 90
- “strspn() — Search String” on page 90

strncmp() — Compare Strings

Format

```
#include <string.h>

int strncmp(const char *string1, const char *string2, size_t count);
```

General Description

The `strncmp()` built-in function compares at most the first *count* characters of the string pointed to by *string1* to the string pointed to by *string2*.

The string arguments to the function should contain a NULL character (`\0`) marking the end of the string.

The relation between the strings is determined by the sign of the difference between the values of the leftmost first pair of characters that differ. The values depend on character encoding. This function is *not* locale sensitive.

Returned Value

The `strncmp()` function returns a value indicating the relationship between the substrings, as follows:

Value	Meaning
-------	---------

- | | |
|-----|---|
| < 0 | String pointed to by <i>substring1</i> less than string pointed to by <i>substring2</i> |
| = 0 | String pointed to by <i>substring1</i> equivalent to string pointed to by <i>substring2</i> |
| > 0 | String pointed to by <i>substring1</i> greater than string pointed to by <i>substring2</i> |

Related Information

- “string.h” on page 49
- “memcmp() — Compare bytes” on page 67
- “strcmp() — Compare Strings” on page 85
- “strcspn() — Compare Strings” on page 86
- “strncat() — Concatenate Strings” on page 87
- “strncpy() — Copy String” on page 89
- “strpbrk() — Find Characters in String” on page 89
- “strrchr() — Find Last Occurrence of Character in String” on page 90
- “strspn() — Search String” on page 90

strncpy() — Copy String

Format

```
#include <string.h>

char *strncpy(char * __restrict__string1, const char * __restrict__string2, size_t count);
```

General Description

The `strncpy()` built-in function copies at most *count* characters of *string2* to *string1*. If *count* is less than or equal to the length of *string2*, a NULL character (`\0`) is *not* appended to the copied string. If *count* is greater than the length of *string2*, the *string1* result is padded with NULL characters (`\0`) up to length *count*.

If copying takes place between objects that overlap, the behavior is undefined.

Returned Value

The `strncpy()` function returns *string1*.

Related Information

- “string.h” on page 49
- “memcpy() — Copy buffer” on page 68
- “strcpy() — Copy String” on page 85
- “strncat() — Concatenate Strings” on page 87
- “strncmp() — Compare Strings” on page 88
- “strpbrk() — Find Characters in String”
- “strrchr() — Find Last Occurrence of Character in String” on page 90
- “strspn() — Search String” on page 90

strpbrk() — Find Characters in String

Format

```
#include <string.h>

char *strpbrk(const char *string1, const char *string2);
```

General 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*.

Returned Value

If successful, `strpbrk()` returns a pointer to the character.

If *string1* and *string2* have no characters in common, `strpbrk()` returns a NULL pointer.

Related Information

- “string.h” on page 49
- “strchr() — Search for Character” on page 84
- “strcspn() — Compare Strings” on page 86
- “strncmp() — Compare Strings” on page 88
- “strrchr() — Find Last Occurrence of Character in String” on page 90
- “strspn() — Search String” on page 90

strchr() — Find Last Occurrence of Character in String

Format

```
#include <string.h>

char *strchr(const char *string, int c);
```

General Description

The `strchr()` function finds the last occurrence of `c` (converted to a char) in *string*. The ending NULL character is considered part of the *string*.

Returned Value

If successful, `strchr()` returns a pointer to the last occurrence of `c` in *string*.

If the given character is not found, `strchr()` returns a NULL pointer.

Related Information

- “string.h” on page 49
- “memchr() — Search buffer” on page 66
- “strchr() — Search for Character” on page 84
- “strcspn() — Compare Strings” on page 86
- “strncmp() — Compare Strings” on page 88
- “strpbrk() — Find Characters in String” on page 89
- “strspn() — Search String”

strspn() — Search String

Format

```
#include <string.h>

size_t strspn(const char *string1, const char *string2);
```

General Description

The `strspn()` function calculates the length of the maximum initial portion of the string pointed to by *string1* that consists entirely of the characters contained in the string pointed to by *string2*.

Returned Value

The `strspn()` function returns the length of the substring found.

Related Information

- “string.h” on page 49
- “strcat() — Concatenate Strings” on page 83
- “strchr() — Search for Character” on page 84
- “strcmp() — Compare Strings” on page 85
- “strcpy() — Copy String” on page 85
- “strcspn() — Compare Strings” on page 86
- “strpbrk() — Find Characters in String” on page 89
- “strchr() — Find Last Occurrence of Character in String”

strstr() — Locate Substring

Format

```
#include <string.h>

char *strstr(const char *string1, const char *string2);
```

General Description

The `strstr()` function finds the first occurrence of the string pointed to by *string2* (excluding the NULL character) in the string pointed to by *string1*.

Returned Value

If successful, `strstr()` returns a pointer to the beginning of the first occurrence of *string2* in *string1*.

If *string2* does not appear in *string1*, `strstr()` returns NULL.

If *string2* points to a string with zero length, `strstr()` returns *string1*.

Related Information

- “string.h” on page 49
- “strchr() — Search for Character” on page 84
- “strcmp() — Compare Strings” on page 85
- “strcspn() — Compare Strings” on page 86
- “strncmp() — Compare Strings” on page 88
- “strpbrk() — Find Characters in String” on page 89
- “strrchr() — Find Last Occurrence of Character in String” on page 90
- “strspn() — Search String” on page 90

strtod — Convert Character String to Double

Format

```
#include <stdlib.h>

double strtod(const char * __restrict__nptr, char ** __restrict__endptr);
```

General Description

The `strtod()` function converts part of a character string, pointed to by *nptr*, to a double. The parameter *nptr* points to a sequence of characters that can be interpreted as a numerical value of the type double.

The `strtod()` function breaks the string into three parts:

1. An initial, possibly empty, sequence of white-space characters, as specified by `isspace()`.
2. A subject sequence interpreted as a floating-point constant or representing infinity or a NAN.
3. A final string of one or more unrecognized characters, including the terminating null byte of the input string.

The subject string is the longest string that matches the expected form.

The expected form of the subject sequence is an optional plus or minus sign with one of the following parts:

strtod

- A non-empty sequence of decimal digits optionally containing a radix character followed by an optional exponent part. A radix character is the character that separates the integer part of a number from the fractional part.
- A 0x or 0X, a non-empty sequence of hexadecimal digits optionally containing a radix character, a base 2 decimal exponent part with a p or P as prefix, a plus or minus sign, and then a sequence of at least one decimal digit, for example, [-]0xh.hhhhp+/-d.
- An INF, ignoring case.
- One of NANQ or NANQ(n), ignoring case.
- One of NANS or NANS(n), ignoring case.
- One of NAN or NAN(n), ignoring case.

See “sscanf() — Read and Format Data” on page 78 for a description of special infinity and NAN sequences recognized by z/OS Metal C.

The pointer to the last string that was successfully converted is stored in the object pointed to by *endptr*, if *endptr* is not a NULL pointer. If the subject string is empty or it does not have the expected form, no conversion is performed. The value of *nptr* is stored in the object pointed to by *endptr*.

Returned Value

If successful, strtod() returns the value of the floating-point number in IEEE Binary Floating-Point format.

In an overflow, strtod() returns +/-HUGE_VAL. In an underflow, it returns 0. If no conversion is performed, strtod() returns 0.

Related Information

- “stdlib.h” on page 48
- “atoi() — Convert character string to integer” on page 53
- “atol() — Convert character string to long” on page 54
- “sscanf() — Read and Format Data” on page 78
- “strtol() — Convert Character String to Long” on page 95
- “strtof — Convert Character String to Float”
- “strtold — Convert Character String to Long Double” on page 97
- “strtoul() — Convert String to Unsigned Integer” on page 99
- “vsscanf() — Format Input of a STDARG Argument List” on page 105

strtof — Convert Character String to Float

Format

```
#include <stdlib.h>
float strtof(const char * __restrict__ nptr, char ** __restrict__ endptr);
```

General Description

The strtof() function converts part of a character string, pointed to by *nptr*, to a float. The parameter *nptr* points to a sequence of characters that can be interpreted as a numerical value of the type float.

The strtof() function breaks the string into three parts:

1. An initial, possibly empty, sequence of white-space characters, as specified by isspace().

2. A subject sequence interpreted as a floating-point constant or representing infinity or a NAN.
3. A final string of one or more unrecognized characters, including the terminating null byte of the input string.

The subject string is the longest string that matches the expected form.

The expected form of the subject sequence is an optional plus or minus sign with one of the following parts:

- A non-empty sequence of decimal digits optionally containing a radix character followed by an optional exponent part. A radix character is the character that separates the integer part of a number from the fractional part.
- A 0x or 0X, a non-empty sequence of hexadecimal digits optionally containing a radix character, a base 2 decimal exponent part with a p or P as prefix, a plus or minus sign, and then a sequence of at least one decimal digit, for example, [-]0xh.hhhhp+/-d.
- An INF, ignoring case.
- One of NANQ or NANQ(n), ignoring case.
- One of NANS or NANS(n), ignoring case.
- One of NAN or NAN(n), ignoring case.

In z/OS Metal C, represent the radix character as a period (.).

See “sscanf() — Read and Format Data” on page 78 for a description of special infinity and NAN sequences recognized by z/OS Metal C.

The pointer to the last string that was successfully converted is stored in the object pointed to by *endptr*, if *endptr* is not a NULL pointer. If the subject string is empty or it does not have the expected form, no conversion is performed. The value of *nptr* is stored in the object pointed to by *endptr*.

Returned Value

If successful, `strtof()` returns the value of the floating-point number in IEEE Binary Floating-Point format.

In an overflow, `strtof()` returns `+/-HUGE_VALF`. In an underflow, it returns 0. If no conversion is performed, `strtof()` returns 0.

Related Information

- “stdlib.h” on page 48
- “atoi() — Convert character string to integer” on page 53
- “atol() — Convert character string to long” on page 54
- “sscanf() — Read and Format Data” on page 78
- “strtod — Convert Character String to Double” on page 91
- “strtol() — Convert Character String to Long” on page 95
- “strtold — Convert Character String to Long Double” on page 97
- “strtoul() — Convert String to Unsigned Integer” on page 99
- “vsscanf() — Format Input of a STDARG Argument List” on page 105

strtok() — Tokenize String

Format

```
#include <string.h>

char *strtok(char * __restrict__string1, const char * __restrict__string2);
```

General Description

The `strtok()` function breaks a character string, pointed to by *string*, into a sequence of tokens. The tokens are separated from one another by the characters in the string pointed to by *string2*.

The token starts with the first character not in the string pointed to by *string2*. If such a character is not found, there are no tokens in the string. `strtok()` returns a NULL pointer. The token ends with the first character contained in the string pointed to by *string2*. If such a character is not found, the token ends at the terminating NULL character. Subsequent calls to `strtok()` will return the NULL pointer. If such a character *is* found, then it is overwritten by a NULL character, which terminates the token.

If the next call to `strtok()` specifies a NULL pointer for *string1*, the tokenization resumes at the first character following the found and overwritten character from the previous call. For example:

```
/* Here are two calls */
strtok(string, " ")
strtok(NULL, " ")

/* Here is the string they are processing */
      abc defg hij
first call finds  ↑
                  ↑ second call starts
```

Note: To use the `strtok()` function, set up an environment by using the `__cinit()` function. When the function is called, GPR 12 must contain the environment token created by the `__cinit()` call.

Returned Value

The first time `strtok()` is called, it returns a pointer to the first token in *string1*. In later calls with the same token string, `strtok()` returns a pointer to the next token in the string. A NULL pointer is returned when there are no more tokens. All tokens are NULL-terminated.

Related Information

- “string.h” on page 49
- “strcat() — Concatenate Strings” on page 83
- “strchr() — Search for Character” on page 84
- “strcmp() — Compare Strings” on page 85
- “strcpy() — Copy String” on page 85
- “strcspn() — Compare Strings” on page 86
- “strspn() — Search String” on page 90

strtok_r() — Split String into Tokens

Format

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

char *strtok_r(char *s, const char *sep, char **lasts);
```

General Description

The function `strtok_r()` considers the NULL-terminated string `s` as a sequence of zero or more text tokens separated by spans of one or more characters from the separator string `sep`. The argument `lasts` points to a user-provided pointer which points to stored information necessary for `strtok_r()` to continue scanning the same string.

In the first call to `strtok_r()`, `s` points to a NULL-terminated string, `sep` to a NULL-terminated string of separator characters and the value pointed to by `lasts` is ignored. The function `strtok_r()` returns a pointer to the first character of the first token, writes a NULL character into `s` immediately following the returned token, and updates the pointer to which `lasts` points.

In subsequent calls, `s` is a NULL pointer and `lasts` will be unchanged from the previous call so that subsequent calls will move through the string `s`, returning successive tokens until no tokens remain. The separator string `sep` may be different from call to call. When no token remains in `s`, a NULL pointer is returned.

Note: To use the `strtok_r()` function, set up an environment by using the `__cinit()` function. When the function is called, GPR 12 must contain the environment token created by the `__cinit()` call.

Returned Value

If successful, `strtok_r()` returns a pointer to the token found.

When no token is found, `strtok_r()` returns a NULL pointer.

Related Information

- “string.h” on page 49
- “strcat() — Concatenate Strings” on page 83
- “strchr() — Search for Character” on page 84
- “strcmp() — Compare Strings” on page 85
- “strcpy() — Copy String” on page 85
- “strcspn() — Compare Strings” on page 86
- “strspn() — Search String” on page 90

strtol() — Convert Character String to Long

Format

```
#include <stdlib.h>

long int strtol(const char * __restrict_nptr, char ** __restrict_endptr, int base);
```

General Description

The `strtol()` function converts *nptr*, a character string, to a long int value.

The function decomposes the entire string into three parts:

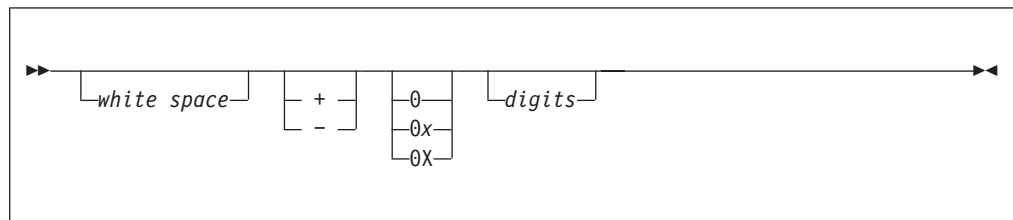
1. A sequence of white space characters as defined by the IBM-1047 codepage.
2. A sequence of characters interpreted as integer in some base notation. This is the *subject sequence*.
3. A sequence of unrecognized characters.

The base notation is determined by *base*, if *base* is greater than zero. If *base* is zero, the base notation is determined by the format of the sequence of characters that follow an optional plus—or optional minus—sign.

- | | |
|----|---|
| 10 | Sequence starts with nonzero decimal digit. |
| 8 | Sequence starts with 0, followed by a sequence of digits with values from 0 to 7. |
| 16 | Sequence starts with either 0x or 0X, followed by digits, and letters A through F or a through f. |

If the base is greater than zero, the subject sequence contains decimal digits and letters, possibly preceded by either a plus or a minus sign. The letters a (or A) through z (or Z) represent values from 10 through 36, but only those letters whose value is less than the value of the base are allowed.

When you use the `strtol()` function, *nptr* should point to a string with the following form:



The pointer to the converted characters, even if conversion was unsuccessful, is stored in the object pointed to by *endptr*, if *endptr* is not a NULL pointer.

Returned Value

If successful, `strtol()` returns the converted long int value.

If unsuccessful, `strtol()` returns 0 if no conversion could be performed. If the correct value is outside the range of representable values, `strtol()` returns `LONG_MAX` or `LONG_MIN`, according to the sign of the value. If the value of *base* is not supported, `strtol()` returns 0.

Related Information

- “stdlib.h” on page 48
- “atoi() — Convert character string to integer” on page 53
- “atol() — Convert character string to long” on page 54
- “atoll() — Convert character string to signed long long” on page 54
- “sscanf() — Read and Format Data” on page 78
- “strtoul() — Convert String to Unsigned Integer” on page 99

strtold — Convert Character String to Long Double

Format

```
#include <stdlib.h>
long double strtold(const char *__restrict__ nptr, char **__restrict__ endptr);
```

General Description

The `strtold()` function converts part of a character string, pointed to by *nptr*, to long double. The parameter *nptr* points to a sequence of characters that can be interpreted as a numerical value of the type long double.

The `strtold()` function breaks the string into three parts:

1. An initial, possibly empty, sequence of white-space characters, as specified by `isspace()`.
2. A subject sequence interpreted as a floating-point constant or representing infinity or a NAN.
3. A final string of one or more unrecognized characters, including the terminating null byte of the input string.

The function then attempts to convert the subject string into the floating-point number, and returns the result.

The expected form of the subject sequence is an optional plus or minus sign with one of the following parts:

- A non-empty sequence of decimal digits optionally containing a radix character followed by an optional exponent part. A radix character is the character that separates the integer part of a number from the fractional part.
- A `0x` or `0X`, a non-empty sequence of hexadecimal digits optionally containing a radix character, a base 2 decimal exponent part with a `p` or `P` as prefix, a plus or minus sign, and then a sequence of at least one decimal digit, for example, `[-]0xh.hhhhp+/-d`.
- An INF, ignoring case.
- One of NANQ or NANQ(n), ignoring case.
- One of NANS or NANS(n), ignoring case.
- One of NAN or NAN(n), ignoring case.

See “`sscanf()` — Read and Format Data” on page 78 for a description of special infinity and NAN sequences recognized by z/OS Metal C.

The pointer to the last string that was successfully converted is stored in the object pointed to by *endptr*, if *endptr* is not a NULL pointer. If the subject string is empty or it does not have the expected form, no conversion is performed. The value of *nptr* is stored in the object pointed to by *endptr*.

Returned Value

If successful, `strtold()` returns the value of the floating-point number in IEEE Binary Floating-Point format.

In an overflow, `strtold()` returns `+/-HUGE_VAL`. In an underflow, it returns 0. If no conversion is performed, `strtold()` returns 0.

Related Information

- “stdlib.h” on page 48
- “atoi() — Convert character string to integer” on page 53
- “atol() — Convert character string to long” on page 54
- “sscanf() — Read and Format Data” on page 78
- “strtod — Convert Character String to Double” on page 91
- “strtof — Convert Character String to Float” on page 92
- “strtol() — Convert Character String to Long” on page 95
- “strtoul() — Convert String to Unsigned Integer” on page 99
- “vsscanf() — Format Input of a STDARG Argument List” on page 105

strtoll() — Convert String to Signed Long Long

Format

```
#include <stdlib.h>

long long strtoll(const char * __restrict__ nptr, char ** __restrict__ endptr, int base);
```

Compile Requirement

Use of this function requires the long long data type. See *z/OS XL C/C++ Language Reference* for information on how to make long long available.

General Description

The `strtoll()` function converts *nptr*, a character string, to a signed long long value.

The function decomposes the entire string into three parts:

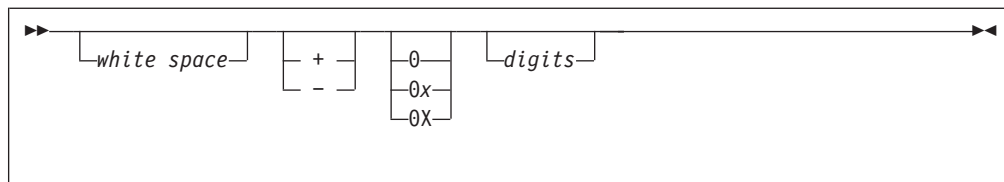
1. A sequence of white space characters as defined by the IBM-1047 codepage.
2. A sequence of characters interpreted as an unsigned integer in some base notation. This is the *subject sequence*.
3. A sequence of unrecognized characters.

The base notation is determined by *base*, if *base* is greater than zero. If *base* is zero, the base notation is determined by the format of the sequence of characters that follow an optional plus or optional minus sign.

- | | |
|----|---|
| 10 | Sequence starts with nonzero decimal digit. |
| 8 | Sequence starts with 0, followed by a sequence of digits with values from 0 to 7. |
| 16 | Sequence starts with either 0x or 0X, followed by digits, and letters A through F or a through f. |

If the base is greater than zero, the subject sequence contains decimal digits and letters, possibly preceded by either a plus or a minus sign. The letters a (or A) through z (or Z) represent values from 10 through 36, but only those letters whose value is less than the value of the base are allowed.

When you are using `strtoll()`, *nptr* should point to a string with the following form:



The pointer to the converted characters, even if conversion was unsuccessful, is stored in the object pointed to by *endptr*, if *endptr* is not a NULL pointer.

Returned Value

If successful, `strtol()` returns the converted signed long long value, represented in the string.

If unsuccessful, `strtol()` returns 0 if no conversion could be performed. If the correct value is outside the range of representable values, `strtol()` returns `LLONG_MAX` (`LLONG_MAX`) or `LLONG_MIN` (`LLONG_MIN`), according to the sign of the value. If the value of base is not supported, `strtol()` returns 0.

Related Information

- “`stdlib.h`” on page 48
- “`atoi()` — Convert character string to integer” on page 53
- “`atol()` — Convert character string to long” on page 54
- “`atoll()` — Convert character string to signed long long” on page 54
- “`sscanf()` — Read and Format Data” on page 78
- “`strtoul()` — Convert String to Unsigned Integer”

strtoul() — Convert String to Unsigned Integer

Format

```
#include <stdlib.h>

unsigned long int strtoul(const char * __restrict__ string1, char ** __restrict__ string2, int base);
```

General Description

The `strtoul()` function converts *string1*, a character string, to an unsigned long int value.

The function decomposes the entire string into three parts:

1. A sequence of white space characters as defined by the IBM-1047 codepage.
2. A sequence of characters interpreted as an unsigned integer in some base notation. This is the *subject sequence*.
3. A sequence of unrecognized characters.

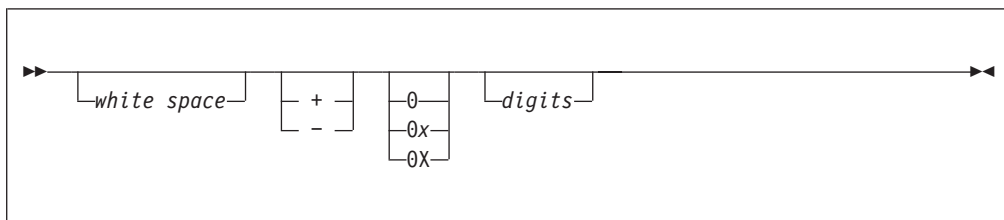
The base notation is determined by *base*, if *base* is greater than zero. If *base* is zero, the base notation is determined by the format of the sequence of characters that follow an optional plus or optional minus sign.

- | | |
|----|---|
| 10 | Sequence starts with nonzero decimal digit. |
| 8 | Sequence starts with 0, followed by a sequence of digits with values from 0 to 7. |

- 16 Sequence starts with either 0x or 0X, followed by digits, and letters A through F or a through f.

If the base is greater than zero, the subject sequence contains decimal digits and letters, possibly preceded by either a plus or a minus sign. The letters a (or A) through z (or Z) represent values from 10 through 36, but only those letters whose value is less than the value of the base are allowed. The function stops reading the string at the first character that it cannot recognize as part of a number. This character can be the first numeric character greater than or equal to the *base*. The `strtoul()` function sets *string2* to point to the end of the resulting output string if a conversion is performed and provided that *string2* is not a NULL pointer.

When you are using the `strtoul()` function, *string1* should point to a string with the following form:



If *base* is in the range of 2-36, it becomes the base of the number. If *base* is 0, the prefix determines the base (8, 16, or 10): the prefix 0 means base 8; the prefix 0x or 0X means base 16; using any other digit without a prefix means decimal.

The pointer to the converted characters, even if conversion was unsuccessful, is stored in the object pointed to by *string2*, if *string2* is not a NULL pointer.

Returned Value

If successful, `strtoul()` returns the converted unsigned long int value, represented in the string.

If unsuccessful, `strtoul()` returns 0 if no conversion could be performed. If the correct value is outside the range of representable values, `strtoul()` returns `ULONG_MAX`. If the value of *base* is not supported, `strtoul()` returns 0.

Related Information

- “`stdlib.h`” on page 48
- “`atoi()` — Convert character string to integer” on page 53
- “`atol()` — Convert character string to long” on page 54
- “`atoll()` — Convert character string to signed long long” on page 54
- “`scanf()` — Read and Format Data” on page 78
- “`strtol()` — Convert Character String to Long” on page 95

strtoull() — Convert String to Unsigned Long Long

Format

```
#include <stdlib.h>

unsigned long long strtoull(register const char * __restrict__ nptr, char ** __restrict__ endptr, int base);
```

Compile Requirement

Use of this function requires the long long data type. See *z/OS XL C/C++ Language Reference* for information on how to make long long available.

General Description

The `strtolll()` function converts *nptr*, a character string, to an unsigned long long value.

The function decomposes the entire string into three parts:

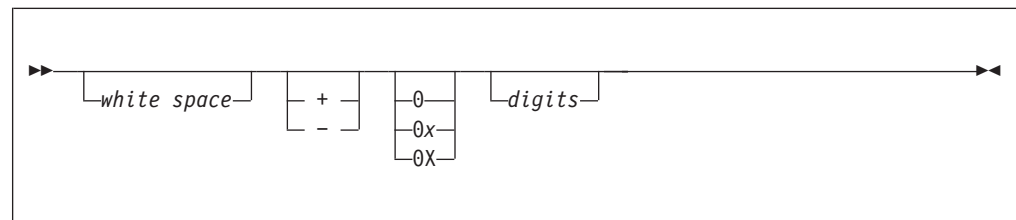
1. A sequence of white space characters as defined by the IBM-1047 codepage.
2. A sequence of characters interpreted as an unsigned integer in some base notation. This is the *subject sequence*.
3. A sequence of unrecognized characters.

The base notation is determined by *base*, if *base* is greater than zero. If *base* is zero, the base notation is determined by the format of the sequence of characters that follow an optional plus or optional minus sign.

- 10 Sequence starts with nonzero decimal digit.
- 8 Sequence starts with 0, followed by a sequence of digits with values from 0 to 7.
- 16 Sequence starts with either 0x or 0X, followed by digits, and letters A through F or a through f.

If the base is greater than zero, the subject sequence contains decimal digits and letters, possibly preceded by either a plus or a minus sign. The letters a (or A) through z (or Z) represent values from 10 through 36, but only those letters whose value is less than the value of the base are allowed. The function stops reading the string at the first character that it cannot recognize as part of a number. This character can be the first numeric character greater than or equal to the *base*. The `strtolll()` function sets *endptr* to point to the end of the resulting output string if a conversion is performed and provided that *endptr* is not a NULL pointer.

When you are using the `strtolll()` function, *nptr* should point to a string with the following form:



If *base* is in the range of 2-36, it becomes the base of the number. If *base* is 0, the prefix determines the base (8, 16 or 10): the prefix 0 means base 8; the prefix 0x or 0X means base 16; using any other digit without a prefix means decimal.

The pointer to the converted characters, even if conversion was unsuccessful, is stored in the object pointed to by *endptr*, if *endptr* is not a NULL pointer.

Returned Value

If successful, `strtolll()` returns the converted unsigned long long value, represented in the string.

strtoull

If unsuccessful, strtoull() returns 0 if no conversion could be performed. If the correct value is outside the range of representable values, strtoull() returns ULLONG_MAX (ULONG_LONG_MAX). If the value of base is not supported, strtoull() returns 0.

Related Information

- “stdlib.h” on page 48
- “atoi() — Convert character string to integer” on page 53
- “atol() — Convert character string to long” on page 54
- “atoll() — Convert character string to signed long long” on page 54
- “sscanf() — Read and Format Data” on page 78
- “strtol() — Convert String to Signed Integer” on page 99

tolower(), toupper() — Convert Character Case

Format

```
#include <ctype.h>

int tolower(int c); /* Convert c to lowercase if appropriate */
int toupper(int c); /* Convert c to uppercase if appropriate */
```

General Description

The tolower() function converts *c* to a lowercase letter, if possible. Conversely, the toupper() function converts *c* to an uppercase letter, if possible.

Returned Value

If successful, tolower() and toupper() return the corresponding character, as defined in the IBM-1047 code page, if such a character exists.

If unsuccessful, tolower() and toupper() return the unchanged value *c*.

Related Information

- “ctype.h” on page 41
- “isalnum() to isxdigit() — Test integer value” on page 60

va_arg(), va_copy(), 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);
```

C99

```
#define _ISOC99_SOURCE
#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);
void va_copy(va_list dest, va_list src);
```

General Description

The `va_arg()`, `va_end()`, and `va_start()` macros access the arguments to a function when it takes a fixed number of required arguments and a variable number of optional arguments. You declare required arguments as ordinary parameters to the function and access the arguments through the parameter names.

The `va_start()` macro initializes the *arg_ptr* pointer for subsequent calls to `va_arg()` and `va_end()`.

The argument *variable_name* is the identifier of the rightmost named parameter in the parameter list (preceding `,` ...). Use the `va_start()` macro before the `va_arg()` macro. Corresponding `va_start()` and `va_end()` macro calls must be in the same function. If *variable_name* is declared as a register, with a function or an array type, or with a type that is not compatible with the type that results after application of the default argument promotions, then the behavior is undefined.

The `va_arg()` macro retrieves a value of the given *var_type* from the location given by *arg_ptr* and increases *arg_ptr* to point to the next argument in the list. The `va_arg()` macro can retrieve arguments from the list any number of times within the function.

The macros also provide fixed-point decimal support under z/OS XL C. The `sizeof(xx)` operator is used to determine the size and type casting that is used to generate the values. Therefore, a call, such as, `x = va_arg(ap, _Decimal(5,2));` is valid. The size of a fixed-point decimal number, however, cannot be made a variable. Therefore, a call, such as, `z = va_arg(ap, _Decimal(x,y))` where `x = 5` and `y = 2` is not valid.

The `va_end()` macro is needed by some systems to indicate the end of parameter scanning.

`va_start()` and `va_arg()` do not work with parameter lists of functions whose linkages were changed with the `#pragma linkage` directive.

`stdarg.h` and `varargs.h` are mutually exclusive. Whichever `#include` comes first, determines the form of macro that is visible.

The type definition for the `va_list` type in this implementation is `"char *va_list"`.

The `va_copy()` function creates a copy (*dest*) of a variable of type `va_list` (*src*). The copy appear as if it has gone through a `va_start()` and the exact set of sequences of `va_arg()` as that of *src*.

After `va_copy()` initializes *dest*, the `va_copy()` macro shall not be invoked to reinitialize *dest* without an intervening invocation of the `va_end()` macro for the same *dest*.

Returned Value

The `va_arg()` macro returns the current argument.

The `va_end()`, `va_copy()`, and `va_start()` macros return no values.

Related Information

- “`stdarg.h`” on page 46
- “`vsnprintf()` — Format and print data to fixed length buffer” on page 104

- “vsprintf() — Format and Print Data to Buffer”

vsnprintf() — Format and print data to fixed length buffer

Format

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

int vsnprintf(char *__restrict__ s, size_t n,
              const char *__restrict__ format, va_list arg);
```

General Description

The vsnprintf() function is equivalent to snprintf(), except that instead of being called with a variable number of arguments, it is called with an argument list as defined by stdarg.h. For a specification of the *format* string, see “sprintf() — Format and Write Data” on page 72.

Initialize the argument list by using the va_start macro before each call. These functions do not invoke the va_end macro, but instead invoke the va_arg macro causing the value of arg after the return to be unspecified.

Notes:

1. Use of vsnprintf() requires that an environment has been set up by using the __cinit() function. When the function is called, GPR 12 must contain the environment token created by the __cinit() call.
2. In contrast to some UNIX-based implementations of the C language, the z/OS XL C/C++ implementation of the vprintf() family increments the pointer to the variable arguments list. To control whether the pointer is incremented, call the va_end macro after each function call.

Returned Value

The vsnprintf() function returns the number of characters that would have been written had *n* been sufficiently large, not counting the terminating null character, or a negative value if an encoding error occurred. Thus, the null-terminated output has been completely written if and only if the returned value is nonnegative and less than *n*.

Related Information

- “stdarg.h” on page 46
- “stdio.h” on page 46
- “va_arg(), va_copy(), va_end(), va_start() — Access Function Arguments” on page 102

vsprintf() — Format and Print Data to Buffer

Format

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

int vsprintf(char * __restrict__ target-string,
             const char * __restrict__ format, va_list arg_ptr);
```

General Description

The `vsprintf()` function is equivalent to the `sprintf()` function, except that instead of being called with a variable number of arguments, it is called with an argument list as defined in `stdarg.h`. For a specification of the *format* string, see “`sprintf()` — Format and Write Data” on page 72.

Initialize the argument list by using the `va_start` macro before each call. These functions do not invoke the `va_end` macro, but instead invoke the `va_arg` macro causing the value of `arg` after the return to be unspecified.

Notes:

1. Use of `vsprintf()` requires that an environment has been set up by using the `__cinit()` function. When the function is called, GPR 12 must contain the environment token created by the `__cinit()` call.
2. In contrast to some UNIX-based implementations of the C language, the z/OS XL C/C++ implementation of the `vprintf()` family increments the pointer to the variable arguments list. To control whether the pointer to the argument is incremented, call the `va_end` macro after each call to `vsprintf()`.

Returned Value

If successful, `vsprintf()` returns the number of characters written *target-string*.

If unsuccessful, `vsprintf()` returns a negative value.

Related Information

- “`stdarg.h`” on page 46
- “`stdio.h`” on page 46
- “`va_arg()`, `va_copy()`, `va_end()`, `va_start()` — Access Function Arguments” on page 102

vsscanf() — Format Input of a STDARG Argument List

Format

```
#define _ISOC99_SOURCE
#include <stdarg.h>
#include <stdio.h>

int vsscanf(const char *__restrict__ s,
            const char *__restrict__ format, va_list arg);
```

General Description

The `vsscanf()` function is equivalent to the `sscanf()` function, except that instead of being called with a variable number of arguments, it is called with an argument list as defined in `stdarg.h`.

Initialize the argument list by using the **`va_start`** macro before each call. These functions do not invoke the **`va_end`** macro, but instead invoke the **`va_arg`** macro causing the value of *arg* after the return to be unspecified.

Notes:

1. Use of `vsscanf()` requires that an environment has been set up by using the `__cinit()` function. When the function is called, GPR 12 must contain the environment token created by the `__cinit()` call.

vsscanf

2. In contrast to some UNIX-based implementations of the C language, the z/OS XL C/C++ implementation of the `vscanf()` family increments the pointer to the variable arguments list. To control whether the pointer is incremented, call the **`va_end`** macro after each function call.

Returned Value

See “`sscanf()` — Read and Format Data” on page 78.

Related Information

- “`stdarg.h`” on page 46
- “`stdio.h`” on page 46
- “`sscanf()` — Read and Format Data” on page 78

Appendix A. Function stack requirements

Table 15 documents the stack frame requirements for each Metal C runtime function. All sizes are in bytes.

Table 15. Stack frame requirements for Metal C runtime functions

Function	AMODE 31 Stack Size	AMODE 64 Stack Size
abs	256	512
atoi	256	512
atol	256	512
atoll	1280	1536
calloc	1024	1536
__cinit	512	512
__cterm	1024	1024
div	256	512
free	512	1536
isalnum	256	512
isalpha	256	512
isblank	256	512
isctrl	256	512
isdigit	256	512
isgraph	256	512
islower	256	512
isprint	256	512
ispunct	256	512
isspace	256	512
isupper	256	512
isxdigit	256	512
labs	256	512
ldiv	256	512
llabs	512	512
lldiv	512	512
malloc	768	1024
__malloc31	768	1024
memcpy	512	512
memchr	512	512
memcmp	512	512
memcpy	512	512
memmove	512	512
memset	256	512
rand	256	512
rand_r	256	512

Table 15. Stack frame requirements for Metal C runtime functions (continued)

realloc	1024	2048
snprintf	3072	3584
snprintf when using e, E, f, F, g, G conversion specifiers	32000	32768
snprintf when using e, E, f, F, g, G conversion specifiers with the L conversion prefix	48896	49920
sprintf	3072	3584
sprintf when using e, E, f, F, g, G conversion specifiers	32000	32768
sprintf when using e, E, f, F, g, G conversion specifiers with the L conversion prefix	48896	49920
srand	256	512
sscanf	2304	2560
sscanf when using e, E, f, F, g, G conversion specifiers	4864	5632
sscanf when using e, E, f, F, g, G conversion specifiers	5888	6656
sscanf when using e, E, f, F, g, G conversion specifiers with the L conversion prefix	23040	23552
strcat	512	512
strchr	512	512
strcmp	512	512
strcpy	512	512
strcspn	768	768
strdup	1024	1536
strlen	512	512
strncat	512	512
strncmp	512	512
strncpy	512	512
strpbrk	768	768
strrchr	512	512
strspn	768	768
strstr	512	512
strtod	4096	4352
strtof	3072	3328
strtok	768	1024
strtok_r	1024	1536
strtol	1024	1024
strtold	21248	21248
strtoll	1024	1024
strtoul	1024	1024

Table 15. Stack frame requirements for Metal C runtime functions (continued)

strtoull	768	1024
tolower	256	512
toupper	256	512
vsprintf	3072	3584
vsprintf when using e, E, f, F, g, G conversion specifiers	32000	32768
vsprintf when using e, E, f, F, g, G conversion specifiers with the L conversion prefix	48896	49920
vsprintf	3072	3584
vsprintf when using e, E, f, F, g, G conversion specifiers	32000	32768
vsprintf when using e, E, f, F, g, G conversion specifiers with the L conversion prefix	48896	49920
vsscanf	2304	2560
vsscanf when using e, E, f, F, g, G conversion specifiers	4864	5632
vsscanf when using e, E, f, F, g, G conversion specifiers with the l conversion prefix	5888	6656
vsscanf when using e, E, f, F, g, G conversion specifiers with the L conversion prefix	23040	23552

Appendix B. Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Vol I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

z/OS information

z/OS information is accessible using screen readers with the BookServer/Library Server versions of z/OS books in the Internet library at:

<http://www.ibm.com/systems/z/os/zos/bkserv/>

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming 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. 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 IBM's application programming interfaces.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Policy for unsupported hardware

Various z/OS elements, such as DFSMS™, HCD, JES2, JES3, and MVS, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Programming Interface Information

This book documents intended Programming Interfaces that allow the customer to write programs to obtain the services of z/OS Metal C runtime library .

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at <http://www.ibm.com/legal/copytrade.shtml>.

IEEE is a trademark of the Institute of Electrical and Electronics Engineers, Inc. in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Adobe, Acrobat, and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Standards

The following standards are supported in combination with the z/OS Metal C runtime library:

- The C language is consistent with *Programming languages - C (ISO/IEC 9899:1999)*. This standard has officially replaced American National Standard for Information Systems-Programming Language C (X3.159–1989) and is technically equivalent to the ANSI C standard. The compiler supports the changes adopted into the C Standard by ISO/IEC 9899:1990/Amendment 1:1994. For more information on ISO, visit their web site at <http://www.iso.ch/>

Index

Special characters

- `__asm` operand lists
 - defining read-write `__asm` operands 18
- `__asm` operands
 - C expressions as `__asm` operands 14
 - defining
 - read-write `__asm` operands 18
 - multiple
 - defining 16
 - read-write 18
- `__asm` statement
 - inserting your own assembly instructions 13
 - using
 - code format string 13
- `__asm` statements
 - C expressions as `__asm` operands 14
 - code format string 14
 - constraints 14
 - examples
 - read-write `__asm` operands 18
 - specifiers 14
- `__cinit()` library function 55
- `__far` qualifier
 - far pointer 24
- `__malloc31()` library function 65
- `__term()` library function 58
- `_MI_BUILTIN` macro
 - data space allocation 28
- `_MI.BUILTIN` macro
 - AR-mode functions 27
 - far-pointer management 26
- `-mgoff` HLASM option
 - and Metal C programs 34
- `#pragma insert_asm`
 - inserting your own assembly statements 22
- `#pragma` directive
 - `MYEPILOG` 8
 - `MYPROLOG` 8
- `+` constraint
 - defining read-write `__asm` operands 19

A

- `abs()` library function 53
- absolute value 53
 - integer argument 53
- access registers
 - AR mode 26
 - management by compiler 26
 - restoring 28
 - saving 28
- accessibility 111
- ADATA debugging information
 - additional source-level information
 - output file format 38
 - CDAASMC procedure 37
 - CDAHLASM invocation 37

- addressing mode
 - and global SET symbols 9
 - and passing parameters 2
- attributes
 - `amode31` 23
 - `amode64` 23
 - recognition of 23
- switching 23
 - commands 35
 - example 23
- ALESERV HLASM macro
 - allocating alternative data spaces 26
- ALET
 - far pointer 24
 - implicit association 26
- ALIAS instructions
 - recognition of 4
- allocating
 - `realloc()` 70
- allocation
 - `_MI_BUILTIN` macro 28
 - of data space 28
- alphabetic character attribute 62
- alternative data spaces
 - accessing 26
 - allocating 26
- AMODE
 - and global SET symbols 9
 - and passing parameters 2
 - function save areas 3
 - return values 3
 - switching
 - commands 35
 - example 23
 - external function calls 23
 - internal function calls 23
- AR mode 24
 - linkage conventions 28
 - programming support 24
 - far-pointer management 26
- AR-mode functions
 - accessing alternative data spaces 26
 - ALET associations 26
 - built-in functions 27
 - C language constructs and far pointers 25
 - data space allocation 28, 32
 - default prolog and epilog code 28
 - far pointers
 - declaration 24
 - dereference 24
 - reference 24
 - memory references 26
- arguments
 - accessing 102
- ARMODE compiler option 24
- `armode` function attribute 24
- as command
 - building Metal C programs 34

- ASC mode
 - restoring 28
 - switching 28
- ASMLANGX debugging utility
 - debugging information format 37
- ASMLANGX utility
 - additional source-level information
 - ADATA debugging information 38
- assembly job step 37
- assembly language programs
 - debugging 38
 - load module size 38
 - source-level information 38
- assembly statements
 - embedding
 - code format string 14
 - example, simple 13
 - file-scope header 5
 - function header 6
 - making a C expression available to HLASM 15
 - making a C variable available to HLASM 14
 - making a C variable expression an `__asm` operand 14
 - operands 16
 - inserting
 - executable 13
 - non-executable 22
 - user-supplied 13
- `atoi()` library function 53
- `atol()` library function 54
- `atoll()` library function 54
- automatic variables
 - defining 6
 - mapping 6

B

- batch environment
 - binder invocation procedures 36
 - building Metal C programs 36
 - assembly step 37
 - bind step 37
 - compilation step 36
 - debugging assembly language programs 38
 - debugging information 37
 - extracting source-level information 37
 - IDF debugging information 38
- bind job step 37
- blank character attribute 62
- BookManager documents xvii
- buffers
 - format and print data 104
- building Metal C programs
 - assembly step
 - symbols longer than eight characters 34
- built-in functions
 - AR-mode functions 27
 - far-pointer management
 - AR-mode programming support 26
- builtins.h header file 41
 - data space allocation 28

- builtins.h header file (*continued*)
 - far versions of library functions 27
 - far-pointer management 26

C

- C expressions
 - used as `__asm` operands 14
- C language constructs
 - far pointers 25
- C memory functions
 - far versions 27
- C string functions
 - far versions 27
- C string pointer
 - copying to far pointer 31
- C symbols
 - name-encoding 4
- `calloc()` library function 55
- CDAASMC JCL procedure
 - binder invocation 36
 - extracting source-level information 37
 - invoking 37
- CDAHLASM
 - invocation 37
- CEE.SCEEPROC data set
 - binder invocation batch procedures 36
- characters
 - conversions
 - lowercase 102
 - uppercase 102
 - finding in a string 89
- classifying characters 60
- clobber list
 - example 17
- code base registers 4
 - clearing 6
- code format string
 - description 13
 - in an `__asm` statement 14
 - substitution specifiers 14
 - treatment of 14
- code format strings
 - data space allocation 28
 - examples
 - read-write `__asm` operands 18
- command
 - syntax diagrams xv
- comparing
 - `strcmp()` 85
 - `strcspn()` 86
 - strings 85, 86, 88
 - `strncmp()` 88
- compilation job step 36
- compiler options
 - AMODE
 - characteristics of compiler-generated assembly source code 4
 - ARMODEINOARMODE 24
 - EPILOG
 - versus `#pragma epilog` 8

- compiler options (*continued*)
 - LONGNAME
 - entry point definition 6
 - external symbols 5
 - LP64
 - characteristics of compiler-generated assembly source code 4
 - programming with Metal C 2
 - PROLOG
 - versus #pragma prolog 8
 - SERVICE
 - optional prefix data 6
- concatenating
 - strcat() 83
 - strings 83, 87
 - strncat() 87
- constants
 - defining 6
- conversions
 - character
 - to lowercase 102
 - to uppercase 102
 - specifier
 - argument in sscanf() 81
 - string to unsigned integer 99
- copying
 - strcpy() 85
 - strings 85, 89
 - strncpy() 89
- CSECT 4
- ctype.h header file 41

D

- data spaces
 - access 24
 - accessing 31
 - allocation 28
 - deallocation 28, 32
 - referencing 31
- data types
 - See type specifier
- debugging
 - assembly language programs 38
 - data formats 37
 - extracting source-level information 37
 - ADATA 38
 - ASMLANGX 38
 - IDF 38
 - in a batch environment 37
 - Interactive Debug Facility (IDF) 38
 - interactive utility 38
 - source-level information 38
- disability 111
- div_t structure 59
- div() library function 59
- division 59
- DSA
 - acquisition and release 3
 - address space 28
 - and global SET symbols 9

- DSA (*continued*)
 - default
 - AR-mode functions 28
 - function save areas 3
 - location 28
 - obtaining 8
 - obtaining and releasing 12
 - pointer 12
 - preallocation 8
- DSECT statement
 - and file-scope trailer 6
 - and function trailer 6
- DSPSERV HLASM macro
 - allocating alternative data spaces 26
- DWARF debugging information
 - CDAASMC procedure 37
 - CDAHLASM invocation 37
- Dynamic Link Libraries (DLLs)
 - See DLLs
- dynamic storage area
 - acquisition and release 3
 - function save areas 3
 - location 28
 - obtaining and releasing 12, 28
 - preallocation 8

E

- EBCDIC
 - codeset
 - IBM-1047 74, 82
- entry point
 - defining
 - under LONGNAME compiler option 6
- epilog code
 - AR-mode functions 8, 28
 - default 12
 - AR-mode functions 28
 - DSA pointer 12
 - NAB pointer 12
 - primary functions 8
 - sample 11
 - supplying your own 8
- EXIT_FAILURE macro 49
- EXIT_SUCCESS macro 49
- external symbols
 - and generated HLASM code 4
- external variables
 - defining 6
 - initializing 6

F

- F4SA save area format
 - and AMODE 3
 - and NAB 3
- F7SA save area format
 - and AMODE 3
 - and NAB 3
- far pointers
 - ALET associations 26

- far pointers (*continued*)
 - C language constructs 25
 - constructing 26
 - copied from C string pointers 31
 - declaration 24
 - dereference 24
 - dereferencing 32
 - passing and returning 28
 - reference 24
 - setting and getting
 - _MI.BUILTIN macro 26
 - built-in functions 26
- far_strcpy library function
 - data space allocation 31
- file-scope header
 - structure 5
- file-scope trailer
 - structure 6
- float.h header file 41
- fopen() library function 46
- formatted I/O 72
- free() library function 59
- function header
 - structure 6
- function prototypes
 - and AMODE 23
- function save area
 - chaining 8
- function save areas
 - AMODE 3
 - formats 3
 - setup 3
- function trailer
 - structure 6
- functions
 - AR-mode
 - prototypes 24
 - arguments 102
 - attributes
 - AR-mode 24
 - prototypes
 - AR-mode 24

G

- global SET symbols
 - and function header 6
- global variables
 - register specification 22
 - storage of 22
- GOFF HLASM option
 - and ALIAS instructions 4
 - when to specify 34
- GPRs
 - and global SET symbols 9

H

- header files
 - builtins.h 27
 - data space allocation 28

- header files (*continued*)
 - builtins.h (*continued*)
 - far-pointer management 26
 - stdint.h header file 47
 - string.h
 - data space allocation 28
 - strings.h
 - data space allocation 28
- hexadecimal 61
- HLASM
 - as utility
 - invoking 34
 - global SET symbols
 - values 6
 - ld utility
 - invoking 35
- HLASM options
 - GOFF
 - and ALIAS instructions 4
- HLASM options
 - with LONGNAME compiler option 34
- HLASM source program, compiler-generated 4
 - characteristics 4
 - structure 5

I

- IDF debugger
 - invocation 38
- initialization
 - strings 89
- insert_asm pragma
 - inserting your own assembly statements 22
- integer
 - pseudo-random 69
- Interactive Debug Facility (IDF)
 - generation of information 38
- inttypes.h header file 42
- isalnum() library function 60
- isalpha() library function 60, 62
- isblank() library function 60, 62
- iscntrl() library function 60
- isdigit() library function 60
- isgraph() library function 60
- islower() library function 60
- isprint() library function 60
- ispunct() library function 60
- isspace() library function 60
- isupper() library function 61
- isxdigit() library function 61

J

- JCL
 - assembly job step 37
 - bind job step 37
 - compilation job step 36
- JCL procedures
 - CEE.SCEEPROC data set 36
 - to build Metal C programs 36

K

keyboard 111

L

labs() library function 63

ld command
 building Metal C programs
 bind options 35

ldiv() library function 63

length function 87

library functions
 far versions 27

limits.h header file 44

linkage conventions
 AR-mode functions
 ASC mode 28
 MVS and Metal C 2

Linkage Editor
 TEST option and load module size 38

list form of a macro
 specifying and using 20

llabs() library function 64

lldiv() library function 64

locating storage 59

LONGNAME compiler option 4
 and HLASM options 34
 and Metal C programs 35

lowercase
 tolower() 102

LTORG statement
 and function trailer 6

M

mainframe
 education xviii

malloc() library function 65

matching failure 83

math.h header file 45

MB_CUR_MAX macro 49

memcpy() library function 66

memchr() library function 66

memcmp() library function 67

memcpy() library function 68

memmove() library function 68

memory
 allocation 70

memory references
 AR mode 26

memset() library function 69

Metal C
 feature and benefits 2

Metal C programs
 building 33
 assembly step 34
 compilation step 34
 xlc utility 34
 z/OS UNIX System Services 34
 JCL procedures to build 36

Metal C programs (*continued*)

 ld command 35

metal.h header file 46

MVS linkage conventions
 and Metal C 2

MYEPILOG #pragma directive
 using 8

MYPROLOG #pragma directive
 using 8

N

NAB linkage extension
 description 3

name encoding
 and C symbols 4

next available byte (NAB)
 pre-allocated stack space 3

noarmode function attribute 24

NOTEST assembler option
 and load module size 38

Notices 113

NULL macro 46

NULL pointer 46

NULL pointer constant 49

numbers 60

O

object code control
 address space control 24

 ASC mode 24

offsetof macro 46

P

parameter passing
 and AMODE 2

parameters
 and global SET symbols 9
 defining 6
 mapping 6

PDF documents xvii

pointers
 storing 15

precision argument, fprintf() family 75

prefix data
 example 6
 structure 6

printing
 sprintf() 72
 vsprintf() 104

prolog
 user-supplied
 global SET symbols 9

prolog code
 AR-mode functions 8, 28
 default 12
 AR-mode functions 28
 DSA pointer 12
 NAB pointer 12

prolog code (*continued*)
 primary functions 8
 sample 10
ptrdiff_t type in stddef header file 46

R

RAND_MAX macro 49
rand_r() library function 69
rand() library function 69
random
 number generator 69
 number initializer 78
 rand_r() 69
 rand() 69
 srand() 78
read-write operands, defining
 using the + constraint 19
reading
 formatted 78
 scanning 78
realloc() library function 70
reallocation of block size 70
reentrancy 4
register storage class specifier
 register specification 22
registers
 access 26
 clobbering 17
 controlling use of 17
 hardware access 24
 specified as __asm operands 14
 specifying 22
remainder 59
resource limits defined 44
return values
 AMODE 3
 formats 3
 setup 3

S

save area formats
 and AMODE 3
 and NAB 3
scanning
 sscanf() 78
SCCNLSAM data set
 epilog code sample 11
 prolog code sample 10
searching
 strchr() 84
 strings 84, 89
 strings for tokens 94, 95
 strspn() 90
seed for random numbers 78
SERVICE compiler option
 optional prefix data 6
SET symbols
 and AMODE 9
 and DSA 9

SET symbols (*continued*)
 and GPRs 9
 and number of fixed parameters 9
 and storage instructions 9
 compiler-defined 9
 for a user-supplied prolog 9
shortcut keys 111
size_t structure 46
snprintf() library function 71
source-level information
 extracting 37
 extracting in a batch environment
 CDAASMC 37
 for disassembly
 suppressing 38
 for IDF 38
space (white space)
 characters
 testing 60
sprintf() library function 72
srand() library function 78
sscanf() library function 78
stack
 allocating space 20
 pre-allocated stack space 3
standard save area format
 and AMODE 3
 and NAB 3
static variables
 defining 6
 mapping 6
stdarg.h header file 46
stddef.h header file 46
stdint.h header file 47
stdio.h header file 46
stdlib.h header file 48
storage
 allocation 70
storage instructions
 and global SET symbols 9
strcat() library function 83
strchr() library function 84
strcmp() library function 85
strcpy() library function 85
strcspn() library function 86
strdup() library function 86
streams
 formatted I/O 78
string.h header file 49
strings
 comparing 86, 88
 concatenating 83, 87
 conversions
 to unsigned integer 99
 copying 85, 89
 ignoring case 85, 86
 initializing 89
 length of 87
 searching 84, 89
 strspn() 90
 searching for tokens 94, 95

- strings (*continued*)
 - substring
 - locating 91
- strings.h header file
 - data space allocation 28
- strlen() library function 87
- strncat() library function 87
- strncmp() library function 88
- strncpy library function
 - data space allocation 28
- strncpy() library function 89
- strpbrk() library function 89
- strrchr() library function 90
- strspn() library function 90
- strstr() library function 91
- strtod() library function 91
- strtof() library function 92
- strtok_r() library function 95
- strtok() library function 94
- strtol() library function 95
- strtold() library function 97
- strtoll() library function 98
- strtoul() library function 99
- strtoull() library function 100
- syntax diagrams
 - how to read xv
- syntax of format for sprintf() 72

T

- TEST assembler option
 - and load module size 38
- testing 60, 62
 - characters
 - white space 60
 - numbers
 - hexadecimal 61
- tokens
 - strtok_r() 95
 - strtok() 94
- tolower() library function 102
- toupper() library function 102

U

- uppercase
 - toupper() 102

V

- va_arg() macro 102
- va_end() macro 102
- va_start() macro 102
- variables
 - making a C variable available to HLASM 14
- vsnprintf() library function 104
- vsprintf() library function 104
- vsscanf() library function 105

X

- xlc utility
 - and HLASM source file 34

Z

- z/OS Basic Skills information center xviii
- z/OS UNIX System Services
 - as utility 34
 - bind options 34
 - ld utility
 - bind options 35

Readers' Comments — We'd Like to Hear from You

z/OS
Metal C Programming Guide and Reference

Publication No. SA23-2225-02

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 your comments via e-mail to: mhvrfs@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



Cut or Fold
Along Line

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
MHVRCFS, Mail Station P181
2455 South Road
Poughkeepsie, NY
12601-5400



Fold and Tape

Please do not staple

Fold and Tape

Cut or Fold
Along Line



Program Number: 5694-A01

Printed in USA

SA23-2225-02

