z/OS

Metal C Programming Guide and Reference

SA23-2225-05

Note:
Before using this information and the product it supports, read the information in “Notices” on page 161.

This edition applies to version 1, release 13, modification 0 of IBM z/OS (product number 5694-A01) and to all subsequent releases and modifications until otherwise indicated in new editions.

This is a major revision of SA23-2225-04.

Contents

Figures

Tables

About this document

This document contains reference information that is intended to help you understand the IBM® z/OS® Metal C runtime library and use the header files and functions provided by the runtime to write applications that can be compiled using the METAL option of the z/OS XL C compiler.

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

Who should read this document

This document is intended for application programmers interested in writing Metal C applications using the z/OS Metal C runtime library.

Where to find more information

For an overview of the information associated with z/OS, see *z/OS Information Roadmap*.

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

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.

For users accessing the Information Center using a screen reader, syntax diagrams are provided in dotted decimal format.

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.

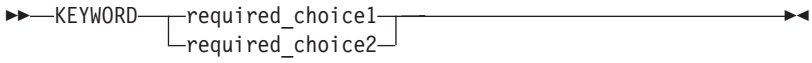
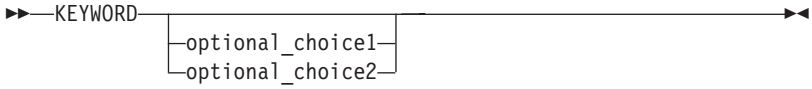
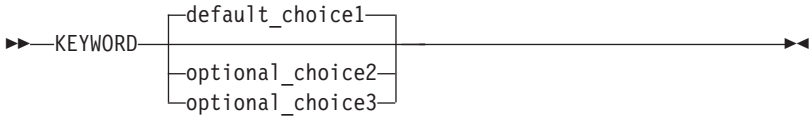
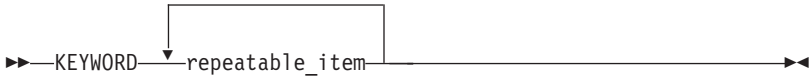
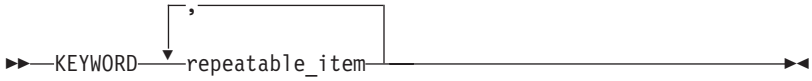
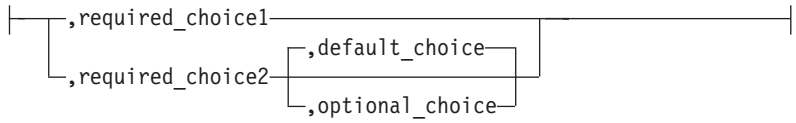
Keywords, variables, and operators may be displayed as required, optional, or default. Fragments, separators, and delimiters may be displayed as required or optional.

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.	
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 labelled 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.	 **fragment:** 

How to send your comments to IBM

We appreciate your input on this publication. Feel free to comment on the clarity, accuracy, and completeness of the information or give us any other feedback that you might have.

Use one of the following methods to send us your comments:

1. Send an email to mhvrcfs@us.ibm.com
2. Visit the Contact z/OS web page at http://www.ibm.com/systems/z/os/zos/webqs.html
3. Mail the comments to the following address:
   IBM Corporation
   Attention: MHVRCFS Reader Comments
   Department H6MA, Building 707
   2455 South Road
   Poughkeepsie, NY 12601-5400
   U.S.A.
4. Fax the comments to us as follows:
   From the United States and Canada: 1+845+432-9405
   From all other countries: Your international access code +1+845+432-9405

Include the following information:

- Your name and address
- Your email address
- Your telephone or fax number
- The publication title and order number:
  z/OS V1R13.0 Metal C Programming Guide and Reference
  SA23-2225-05
- The topic and page number related to your comment
- The text of your comment.

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 submit.

If you have a technical problem

Do not use the feedback methods listed above. Instead, do one of the following:

- Contact your IBM service representative
- Call IBM technical support
- Visit the IBM support portal at http://www.ibm.com/systems/z/support/

Summary of changes

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

Changes made in z/OS Version 1 Release 13

This document contains information that was previously presented in *z/OS Metal C Programming Guide and Reference*, SA23-2225-03, which supports z/OS Version 1 Release 12.

New information:

- "User-replaceable heap services" on page 65 has been added.
- "Building Metal C programs with IPA" on page 46 has been added.
- "Function entry point marker" on page 6, "Function property block" on page 6, and an explanation of the associated "Prefix data" on page 10 have been added to the explanation of "Structure of a compiler-generated HLASM source program" on page 5.
- Information about "Prefix data" on page 10 has been updated.
- "argc argv parsing support" on page 33 has been added.
- "User reserved DSA space" on page 13 has been added.
- "qsort() — Sort array" on page 83 has been added.

Changes made in z/OS Version 1 Release 12

This document contains information that was previously presented in *z/OS Metal C Programming Guide and Reference*, SA23-2225-02, which supports z/OS Version 1 Release 11.

New information:

- New compiler global SET symbols have been added. For more information, see "Compiler-generated global SET symbols" on page 13.
- The RENT option has been enabled under the METAL option to support constructed reentrancy for C programs with writable static and external variables. For more information, see "RENT mode support" on page 31.
- Appendix B, "CICS programming interface examples," on page 125 has been added.

Changed information:

- The "Readers' Comments - We'd Like to Hear from You" section at the back of this publication has been replaced with a new section "How to send your comments to IBM" on page xix. The hardcopy mail-in form has been replaced with a page that provides information appropriate for submitting comments to IBM.
- The "SCCNSAM(MYPROLOG) macro" on page 17 sample has been updated.

Changes made in z/OS Version 1 Release 11

This document contains information that was 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 53 has been added.
- “math.h” on page 57 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 63.
- “strtod — Convert Character String to Double” on page 105 has been added.
- “strtof — Convert Character String to Float” on page 106 has been added.
- “strtold — Convert Character String to Long Double” on page 111 has been added.

Changed information:

- “stdlib.h” on page 60 has been updated.
- “sprintf() — Format and Write Data” on page 86 has been updated.
- “sscanf() — Read and Format Data” on page 92 has been updated.
- Table 14 on page 121 has been updated.

Chapter 1. About IBM z/OS Metal C

The XL C METAL compiler option generates code that does not require access to the Language Environment® support at run time. 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 20.

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 63.

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

Metal C environment

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.

Metal C environments are intended to be used serially by a single dispatchable unit of work. If you need to share environments between multiple dispatchable units, you must make sure that the use of each environment is serialized.

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 33.
- Provides support for embedding your assembly statements into the compiler-generated code. See "Inserting HLASM instructions into the generated source code" on page 20.

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 the 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 the topic about 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.

The F4SA signature generation can be suppressed by setting the `&CCN_SASIG` global SET symbol to 0 in your prolog code. For information about the `&CCN_SASIG` global SET symbol, see Table 3 on page 15 User modifiable global SET symbols.

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 121.

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, you can use the IEABRCX DEFINE instruction 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. To disable the effect of IEABRCX, you can add the instruction IEABRCX DISABLE. For more information about the IEABRCX macro, see *z/OS MVS Programming: Authorized Assembler Services Reference EDT-IXG*.
- 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 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 43 on page 45.
- 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 function:
  - A function header
  - A function entry point marker
  - A function property block (FPB)
  - A function body
  - A function trailer
- File-scope trailer

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.

- IEABRCX DEFINE statement 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 10 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 entry point marker: A function entry point marker is generated immediately before the entry point of each function. The function entry point marker is an 8-byte field containing the signature `0x00C300C300D501nn`. Immediately following the marker is a 4-byte signed offset from the start of the entry point marker to the function property block belonging to the current function. Figure 1 shows what a function entry point marker looks like in generated assembler code.

```
        ENTRY @@CCN@2                                                  000005
@@CCN@2 AMODE 31                                                       000005
        DC XL8'00C300C300D50100'     Function entry point marker       000005
        DC A(@@FPB@1-*+8)            Signed offset to FPB              000005
        DC XL4'00000000'             Reserved                          000005
@@CCN@2 DS 0F
```

Figure 1. Function entry point marker in generated assembler code

Function property block: The function property block (FPB) is made up of a fixed part (20 bytes in size) followed by a contiguous optional part, with the presence of optional fields indicated by flag bits. Optional fields, if present, are stored immediately following the fixed part of the FPB aligned on fullword boundaries in the order specified below.

Figure 2 shows the FPB fixed area fields and the definitions.

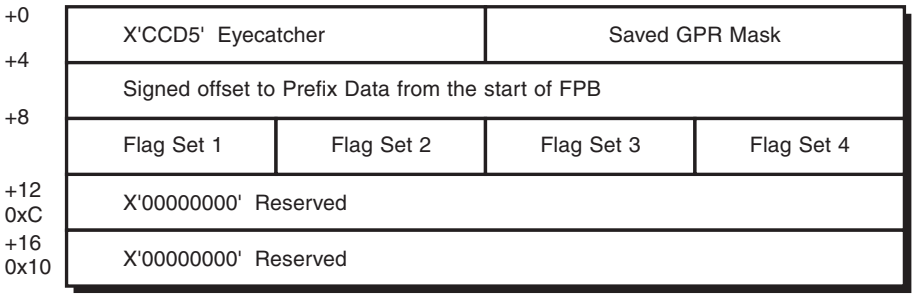


Figure 2. Function property block fixed area fields

Eyecatcher
A 16-bit field that is set to 0xCCD5.

Saved GPR Mask

A 16-bit mask, indicating which registers are saved and restored by the associated routine. Bit 0 indicates register 0, followed by bits for registers 1 to 15 in order.

Signed offset to Prefix Data from the start of FPB

The offset of the prefix data belonging to the compilation unit containing the function described by this FPB.

Flag Set 1

Flag definition	
'1.......'	Function is AMODE 64.
'0.......'	Function is AMODE 31.
'.1......'	Function is AR mode.
'.0......'	Function is primary mode.
'..00000.'	Reserved.
'.......1'	A vararg function.
'.......0'	Not a vararg function.

Flag Set 2

Flag definition	
'1.......'	External function.
'0.......'	Internal function.
'......00'	This function has the standard 72-byte save area.
'......01'	This function has the F4SA 144-byte save area.
'......10'	This function has the F7SA 216-byte save area.
'.00000..'	Reserved.

Flag Set 3

Flag definition	
'1.......'	Indicates the floating-point registers (FPR) are saved in the DSA and the FPR mask and offset to the FPR save area are present in the optional part of the FPB.
'0.......'	Indicates the floating-point registers (FPR) are not saved in the DSA.
'.1......'	Indicates the high-half of 64-bit general purpose registers (GPR) are saved in the DSA and the HGPR mask and offset to the HGPR save area are present in the optional part of the FPB.
'.0......'	Indicates the high-half of 64-bit general purpose registers (GPR) are not saved in the DSA.
'..000000'	Reserved.

Flag Set 4

Flag definition	
'0000000.'	Reserved.

'.......1' Indicates that the length of the function name and the function name field are present in the optional part of the FPB.

'.......0' Indicates that the function name field is not present in the FPB.

Note: When the COMPRESS compiler option is in effect, the function name field is not present in the FPB.

There are several optional FPB fields. The presence of each field is indicated by a flag bit in FPB flag set 3 or FPB flag set 4. When an optional field is less than 4 bytes in length, the entire word is present if any of the fields in that word are present. Unused parts of the word are filled with zeroes. The optional fields are fullword aligned and appear in the order listed below.

FPR Mask

A 16-bit mask indicating which of floating-point registers (FPR) are saved and restored by this function. Bit 0 indicates FPR0, followed by bits for FPR1 to FPR 15.

HGPR Mask

A 16-bit mask indicating which of 64-bit general purposes registers (GPR) whose high-words are saved and restored by this function. Bit 0 indicates GPR0, followed by bits for GPR1 to GPR 15.

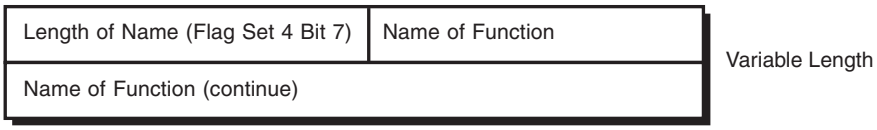
Note: If either bit 0 or bit 1 of flag set 3 is on, the fullword variable representing FPR mask and HGPR mask is present.

FPR Savearea Offset

A 32-bit field containing the offset from the start of the DSA to the FPR save area. The contents of the FPRs indicated by the FPR Mask are stored contiguously from the beginning of the FPR save area regardless of the register number. For example, if the FPR Mask contains 0x00A0 and the FPR save area offset contains 0x100, FPR8 is stored at R13+0x100 and FPR10 is stored at R13+0x108.

HGPR Savearea Offset

A 32-bit field containing the offset from the start of the DSA to the HGPR save area. The HGPR save area is 64-byte in size which holds 16 32-bit high-words. The order of the high-words stored in the save area is GPR14, GPR15, and GPR0 - GPR13. Only the slots which correspond to the bits in the HGPR Mask contain the saved high-word contents.



Name of Function

The optional function name fields start with a 2-byte length field which contains the actual length of the function name that follows.

Figure 3 shows what a function property block looks like in generated assembler code.

```
@@FPB@ LOCTR
@@FPB@1 DS 0F               Function Property Block                  000000
        DC XL2'CCD5'                Eyecatcher                       000000
        DC BL2'1111100000000011'    Saved GPR Mask                   000000
        DC A(@@PFD@@-@@FPB@1)       Signed Offset to Prefix Data     000000
        DC BL1'00000000'            Flag Set 1                       000000
        DC BL1'10000000'            Flag Set 2                       000000
        DC BL1'01000000'            Flag Set 3                       000000
        DC BL1'00000001'            Flag Set 4                       000000
        DC XL4'00000000'            Reserved                         000000
        DC XL4'00000000'            Reserved                         000000
        DC XL2'0000'                Saved FPR Mask                   000000
        DC BL2'1111000000000011'    Saved HGPR Mask                  000000
        DC XL4'00000058'            HGPR Save Area Offset            000000
        DC AL2(4)                                                    000000
        DC C'main'
```

Figure 3. Function property block in generated assembler code

In this example, the @@FPB@ LOCTR instruction tells the assembler to group all FPBs separate from the code and data generated for the functions.

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 fixed part (36 bytes in size) followed by a contiguous optional part, with the presence of optional fields indicated by flag bits in flag set 4. Optional fields, if present, are stored immediately following the fixed part of the prefix data aligned on halfword boundaries in the order specified by the left to right bits in flag set 4.

Figure 4 shows the prefix data fixed area fields and definitions.

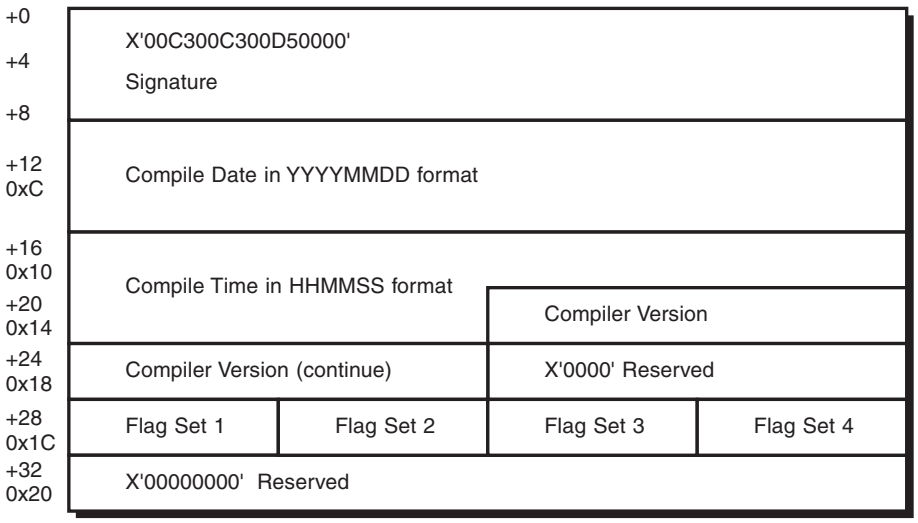


Figure 4. Prefix data fixed area fields

Signature
An 8-byte field that is set to `0x00C300C300D50000`. The last byte in the signature is the version number which can change in future releases.

Compile date
An 8-byte field that contains the date of the compile in YYYYMMDD format.

Compile time
A 6-byte field that contains the time of the compile in HHMMSS format.

Compiler version
A 4-byte field that contains the binary value of the compiler version and release.

Flag Set 1

Flag definition

`'.1......'`	Compiled with RENT option.
`'.0......'`	Compiled with NORENT option.
`'0.000000'`	Reserved.

Flag Set 2

Flag definition

'00000000' Reserved.

Flag Set 3

Flag definition

'00000000' Reserved.

Flag Set 4

Flag definition

'1.......' Indicates the presence of a user comment string.

'0.......' Indicates no optional user comment string.

'.1......' Indicates the presence of a service string.

'.0......' Indicates no service string.

'..000000' Reserved.

There are two optional prefix data fields, whose presence is indicated by a flag bit in flag set 4.

User Comment String: The user comment string comes from the string specified in both or one of `#pragma comment(copyright, "...")` and `#pragma comment(user, "...")`. If you have either or both `#pragma`, the flag bit is set to one, and the user comment string contains the concatenated strings from multiple `#pragma`.

Service String: The service string comes from the string specified in the SERVICE compiler option.

Figure 5 and Figure 6 on page 12 show how prefix data is generated from a sample program compiled with the RENT and SERVICE (“Service String”) options.

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

Figure 5. A sample program to generate prefix data

```
@@PFD@@ DC XL8'00C300C300D50000'      Prefix Data Marker          000008
        DC CL8'20101007'             Compiled Date YYYYMMDD      000008
        DC CL6'153745'               Compiled Time HHMMSS        000008
        DC XL4'410D0000'             Compiler Version            000008
        DC XL2'0000'                                             000008
        DC BL1'01000000'             Flag Set 1                  000008
        DC BL1'00000000'             Flag Set 2                  000008
        DC BL1'00000000'             Flag Set 3                  000008
        DC BL1'11000000'             Flag Set 4                  000008
        DC XL4'00000000'                                         000008
        DS 0H                                                    000008
        DC AL2(30)                                               000008
        DC C'copyright comment user comment'                     000008
        DS 0H                                                    000008
        DC AL2(14)                                               000008
        DC C'Service String'                                     000008
```

Figure 6. Prefix data generated

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 33 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 7 provides an example.

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

Figure 7. Specification of your own prolog and epilog code for a function

To apply the same prolog and epilog 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 epilog code is applied only to the functions that have external linkage. To apply your prolog and epilog 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 7 on page 12, 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 epilog code needs to ensure that:

- The primary functions of the epilog code have been performed.
- The content of GPR 15, on entry to your epilog code, is preserved.
- If a 64-bit integer value is returned from an AMODE 31 program, the low half of the return value contained in GPR 0 is preserved.

Your prolog and epilog 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.
- Preserve the registers used by the compiler generated code.
- Set up the NAB for the called functions.

User reserved DSA space: User reserved DSA space can be enabled by using the compiler option DSAUSER. When using this compiler option, a user field with the size of a pointer is reserved on the stack. This user field can be utilized by your prolog or epilog code. The user field can be located by the HLASM global set symbol &CCN_DSAUSER, which provides the offset to the user field. The compiler allocates the field on the stack only, without initializing it.

The following example shows how &CCN_DSAUSER is set by the compiler:

```
&CCN_DSAUSER SETC '#USER_2-@@AUTO@2'
```

The following example shows how &CCN_DSAUSER can be used in your prolog code:

```
STG 0,&CCN_DSAUSER.(,13)
```

For detailed information about the DSAUSER compiler option, see the topic about DSAUSER and NODSAUSER in *z/OS XL C/C++ User's Guide*.

Compiler-generated global SET symbols

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

Table 2 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: • 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_STATIC	Logical	Set to "1" if the function is static.
&CCN_MAIN	Logical	Set to "1" if this is function "main".
&CCN_RENT	Logical	Set to "1" if the RENT compiler option is specified.
&CCN_PRCN	Character	The symbol representing the function.
&CCN_CSECT	Character	The symbol representing the CSECT in effect.
&CCN_DSAUSER	Character	The assembly time computed offset to the user field on the function stack.
&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: • A=AR mode • P=Primary mode For information about AR mode, see “AR-mode programming support” on page 33.

&CCN_NAB_OFFSET	Character	The assembly time computed offset to the NAB pointer on the stack of the function.
		The following example shows how &CCN_NAB_OFFSET is set by the compiler:
		`&CCN_NAB_OFFSET SETC'#NAB_2-@@AUTO@2'`
		The following example shows how &CCN_NAB_OFFSET can be used in the prolog code:
		`STG 0,&CCN_NAB_OFFSET.(,13)`

Table 2. Compiler-generated global SET symbols (continued)

Global SET symbol	Type	Description

&CCN_IASM_MACRO	Character	The name of the in-stream macro that contains all the #pragma insert_asm supplied statements. The setting of &CCN_IASM_MACRO only happens in the presence of inserted assembler statements provided by #pragma insert_asm directives. In the presence of such directives, the compiler generates an in-stream HLASM MACRO like this:
		`MACRO`
		`@@IASM@`
		`*from #pragma insert_asm #1 000004`
		`*from #pragma insert_asm #2 000006`
		`*from #pragma insert_asm #3 000008`
		`MEND`
		The following example shows how &CCN_IASM_MACRO is set by the compiler:
		`&CCN_IASM_MACRO SETC '@@IASM@'`
		The MACRO name in &CCN_IASM_MACRO can be placed anywhere within the prolog/epilog code.

&CCN_PRCN_LONG	Character	The actual function name up to the 1024 character HLASM limit. The setting of &CCN_PRCN_LONG is subject to the HLASM limit of 1024 characters on a SETC instruction. When the function name is longer than 1024 characters, the character value set will be truncated to 1021 characters and appended with '...'.

Table 3 describes the global SET symbols that can be set by your prolog and epilog code to conditionally enable or disable code sequences generated by the compiler.

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.

&CCN_NAB_STORED	Logical	0	Set to "1" to indicate that NAB pointer storing code was done in the prolog code. The following example shows the code that is generated by the compiler to cause the NAB computing and storing code to be conditionally assembled based on the setting of &CCN_NAB_STORED:
			`AIF (&CCN_NAB_STORED).@@NONAB2`
			`LGHI 0,160`
			`ALGR 0,13`
			`STG 0,#NAB_2-@@AUTO@2(,13)`
			`.@@NONAB2 ANOP`

&CCN_IASM_FRONT	Logical	0	Set to "1" to indicate that &CCN_IASM_MACRO was already called. The following example shows the code that is generated by the compiler to cause the &CCN_IASM_MACRO to be conditionally assembled based on the setting of &CCN_IASM_STORE:
			`AIF (&CCN_IASM_FRONT).@@NOIASM1`
			`@@IASM@`
			`.@@NOIASM1 ANOP`

&CCN_WSA_INIT	Character	'CCNZWSAI' for 31-bit 'CCNZQWSI' for 64-bit	Function name for WSA initialization routine.

Table 3. User modifiable global SET symbols (continued)

Global SET symbol	Type	Default	Description
&CCN_WSA_TERM	Character	'CCNZWSAT' for 31-bit 'CCNZQWST' for 64-bit	Function name for WSA termination routine.
&CCN_APARSE	Logical	1	Set to "1" to trigger CCNZINIT call to parse `argc` and `argv`. Set to "0" to disable `argc` and `argv` parsing.

SCCNSAM(CCNZGBL) macro

Sample macros for prolog and epilog code are supplied in the SCCNSAM data set. The SCCNSAM(CCNZGBL) macro contains assembler instructions to declare all the Global Set Symbols to be referenced. You need to copy the CCNZGBL macro into your prolog and epilog code. Figure 8 shows the sample CCNZGBL macro.

Figure 8. SCCNSAM(CCNZGBL)

```
*********************************************************************
* *
* MACRO-NAME = CCNZGBL *
* DESCRIPTIVE-NAME = METAL C GLOBAL SET SYMBOLS *
* *
* USAGE = COPY CCNZGBL *
* *
*********************************************************************
         GBLA &CCN_DSASZ DSA size of the function
         GBLA &CCN_SASZ Save area size of this function
         GBLA &CCN_ARGS Number of fixed parameters
         GBLA &CCN_RLOW High GPR on STM/STMG
         GBLA &CCN_RHIGH Low GPR on STM/STMG
         GBLB &CCN_MAIN True if function is main
         GBLB &CCN_LP64 True if compiled with LP64
         GBLB &CCN_NAB True if NAB needed
         .* &CCN_NAB is to indicate if there are called functions that depend on
         .* stack space being pre-allocated. When &CCN_NAB is true you'll need
         .* to add a generous amount to the size set in &CCN_DSASZ when you
         .* otbain the stack space.
         GBLB &CCN_ALTGPR(16) Altered GPRs by the function
         GBLB &CCN_SASIG True to gen savearea signature
         GBLC &CCN_PRCN Entry symbol of the function
         GBLC &CCN_CSECT CSECT name of the file
         GBLC &CCN_LITN Symbol name for LTORG
         GBLC &CCN_BEGIN Symbol name for function body
         GBLC &CCN_ARCHLVL n in ARCH(n) option
         GBLC &CCN_ASCM A=AR mode P=Primary mode
         GBLC &CCN_IASM_MACRO MACRO name for all insert_asm
         GBLB &CCN_IASM_FRONT True if insert_asm at front
         GBLC &CCN_NAB_OFFSET Offset to NAB pointer in DSA
         GBLB &CCN_NAB_STORED True if NAB pointer stored
         GBLC &CCN_PRCN_LONG Full func name up to 1024 chars
         GBLB &CCN_STATIC True if function is static
         GBLB &CCN_RENT True if compiled with RENT
         GBLC &CCN_WSA_INIT WSA initialization function name
         GBLC &CCN_WSA_TERM WSA termination function name
         GBLB &CCN_APARSE True to parse OS PARM
         GBLC &CCN_DSAUSER Offset to user field in DSA
```

SCCNSAM(MYPROLOG) macro

Sample macros for prolog code are supplied in the SCCNSAM data set. Figure 9 shows the sample prolog code.

```
         MACRO
&NAME    MYPROLOG
         COPY  CCNZGBL
         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   (NOT &CCN_RENT).SKIP_R1
         AIF   (&CCN_LP64).LP64_11
         LR    2,0
         AGO   .SKIP_R1
.LP64_11 ANOP
         LGR   2,0
.SKIP_R1 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'
```

Figure 9. SCCNSAM(MYPROLOG) (Part 1 of 2)

```
.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   .CHECK_R
.LP64_5  ANOP
         LGR   15,1
         STG   15,136(,13)
         LG    1,32(,13)
         STG   13,128(,15)
         LGR   13,15
.CHECK_R ANOP
         AIF   (NOT &CCN_RENT).DROP
         AIF   (&CCN_LP64).LP64_12
         LR    0,2
         AGO   .DROP
.LP64_12 ANOP
         LGR   0,2
.DROP    ANOP
         DROP  15
         MEND
```

Figure 9. SCCNSAM(MYPROLOG) (Part 2 of 2)

SCCNSAM(MYEPILOG) macro

Sample macros for epilog code are supplied in the SCCNSAM data set. Figure 10 on page 19 shows the sample epilog code.

```
         MACRO
&NAME    MYEPILOG
         COPY  CCNZGBL
         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)
         DROP  15
         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
         MEND
```

Figure 10. 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 in Figure 9 on page 17 and Figure 10. 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 29.
- Invoking a macro in the list form. See “Specifying and using the list form of a macro” on page 27.
- Supplying your own function prologs and epilogs. See “Prolog and epilog code” on page 12.

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 11 on page 21 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 11. 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 12, 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 12. Code format string with two instructions

The example in Figure 12 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 12 and Figure 13 generate the same output.

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

Figure 13. Code format string with two instructions, formatted for readability

Notes:

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

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 14 on page 22 shows an `__asm` statement that substitutes a C variable into an output `__asm` operand. Figure 15 on page 22 shows those assembly instructions.

```
void foo() {
  int x;
                          1  2  3
  __asm ( " ST 12,%0\n" : "=m"(x) :: "r12" );
            4     5
}
```

Figure 14. Substitution of a C variable into an output __asm operand

Notes:

1. A colon that marks the beginning of the list of output __asm operands, it follows the code format string.
2. The output __asm operand is "=m"(x). The constraint "m" communicates the syntactic requirement to the 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.

From the __asm statement used in Figure 14, the compiler embeds the instructions shown in Figure 15 in the generated HLASM source code.

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

Figure 15. HLASM source code embedded by the compiler

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 14 (ST 12,%0).

Substitution of a C pointer into an __asm operand: The code format string in Figure 16 on page 23 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 16 on page 23. For an example that allows you to specify registers, see Figure 20 on page 25.

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

           1                    2       3
    __asm( " WTO  MF=(E,(%0)) " : : "r"(&wto_buff));
    return 0;
}
```

Figure 16. Substitution of a C pointer into an __asm operand

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 14 on page 22).
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.

Definition of multiple __asm operands: In Figure 17, 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) : "r12" );
                       1         2                 3
}
```

Figure 17. __asm operand lists

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 18 on page 24 shows the compiler-generated HLASM code from the __asm statement in Figure 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
```

Figure 18. Compiler-generated HLASM code from the __asm statement

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.

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 20 on page 25. Without the clobber list, the `__asm` statement embeds incorrect assembly statements, as shown in Figure 19.

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

Figure 19. Unsuccessful attempt to specify registers

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.

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 20 on page 25.

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 &pl  */
      " LR 1,%1\n"         /* load &dcb */
      " SVC 21"
                : :  "r"(&pl), "r"(&dcb) : "r0","r1");
                                           1    2
```

Figure 20. Register specification with clobbers

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.

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 21 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 21) and correctly (Figure 23 on page 26). The code in Figure 21 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
```

Figure 21. Incorrect __asm operand definition for both input and output

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

The compiler-generated HLASM source code in Figure 22 is the result of the incorrect definition in Figure 21.

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

Figure 22. Incorrect compiler-generated HLASM source code from the incorrect __asm operand definition for both input and output

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 21 specified variable `x` as an output operand only.

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 23.

Figure 23 shows the code format string that will embed the correct assembler statements (as shown in Figure 24).

```
__asm  ( " AR %0, %1" :  "=r"(x) : "r"(y), "0"(x) );
          1          2          3  4
```

Figure 23. Successful definition of an __asm operand for both input and output

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 24, before the code format string is embedded in the HLASM output.

The compiler-generated HLASM source code in Figure 24 is the result of the correct definition in Figure 23.

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

Figure 24. Correct compiler-generated HLASM source code from the correct __asm operand definition for both input and output

Note: The compiler inserted `L 4,@3x` at the beginning of the instruction sequence because the code format string in Figure 23 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.

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 25, the "+" constraint is used to define that the variable `x` is used both as input and output.

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

Note: This example is parsed as though the operand list in Figure 23 is given.

Figure 25. 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 26 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 26. 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 28 on page 28.
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 16 on page 23 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 28 on page 28, in which all __asm statements are connected through the C variable operands listmsg1 and buff.

Figure 27 on page 28 provides an example that uses the list form of a macro without considering reentrancy.

```
  1              2                  3
__asm(" WTO 'hello world',MF=L" : "DS"(listmsg1));

int main() {
       4                                        5
      __asm( " WTO  MF=(E,(%0)) " : : "r"(&listmsg1));
      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) to allocate 100 bytes). 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 27. 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 27 will be incorrect when support for reentrancy is required. The proper way to use the macro is shown in Figure 28.

```
__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 28. 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.

Note: 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\")")`.

Example: Using the #pragma insert_asm directive to map specific DSECT information: Figure 29 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 29. 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 30.

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*.

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

Notes:

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

Figure 30. 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 **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.
- The __ptr64 qualifier can be used to specify a 64-bit pointer on an AMODE 31 program; the pointer cannot be dereferenced at the AMODE 31 program.
- AMODE-switching is not supported at IPA link.

Example of an AMODE31 program that calls an AMODE64 program

In Figure 31, 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 45.

```
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 31. AMODE31 program that calls an AMODE64 program

RENT mode support

The RENT option supports constructed reentrancy for C programs with writable static and external variables. This makes Metal C programs with writable static and external variables to be reentrant so a program can be concurrently used by multiple users. The writable static area (WSA) can be managed by user provided routines. Using the RENT support, you can use Metal C as an alternative to assembler, to write programs to run in CICS® environment. For information about how the CICS API and the CICS XPI can be used in Metal C and for programming examples, see Appendix B, “CICS programming interface examples,” on page 125. The default of the RENT option is NORENT.

Note: The Metal C RENT support is independent of and different from the NOMETAL RENT support. They should not be mixed.

Example

```
xlc -qMETAL -qRENT -S a.c            1
as -mgoff a.s                        2
export _LD_SYSLIB="//'CBC.SCCNOBJ'"  3
ld a.o
```

Notes:

1. Request Metal C RENT support.
2. The HLASM GOFF option is required to assemble the compiler generated code for RENT.
3. It is necessary to add CBC.SCCNOBJ dataset to the binder SYSLIB for the resolution of CCNZINIT and CCNZTERM.

Linkage convention

General Purpose Register 0 (GPR0) is used to pass the WSA address. The prolog code you supplied needs to preserve the content of GPR0 on exit of the prolog code. Programs compiled with RENT and NORENT can be mixed as long as the NORENT programs do not call RENT programs.

Note: Global variables compiled with RENT and NORENT cannot have the same name.

Assembler code interface

The runtime RENT support is accomplished by additional calls generated for the function "main" between the prolog/epilog code and the function code. The RENT environment initialization and termination routines are called to establish and terminate the dynamically allocated WSA storage with the static initialization data applied. For the AMODE 31 "main" function, CCNZINIT and CCNZTERM are the names of these routines. While for the AMODE 64 "main" function, CCNZQINI and CCNZQTRM are the function names. For the simplicity of further references, only the names of the 31-bit version are used. The actual WSA storage management is done by user supplied plug-in routines called from CCNZINIT and CCNZTERM. Two user modifiable Global Set Symbols, &CCN_WSA_INIT and &CCN_WSA_TERM, can be used in the user supplied prolog code to set the user supplied WSA initialization and termination function names. The AMODE of the user supplied routines has to be the same as the AMODE of function "main".

Note: CCNZINIT, CCNZTERM, CCNZWSAI and CCNZWSAT require the stack space to be supplied by function "main" prolog code. Both CCNZINIT and CCNZTERM require NAB to be established by function "main". Also, CCNZINIT and CCNZTERM assume stack space to be available for the WSA initialization and termination functions. This arrangement is to ensure

that the stack space used by CCNZINIT and CCNZTERM as well as the WSA initialization and termination routines is consistent with the stack space used by function "main". Allocating 1K of extra stack space (in addition to the DSA size suggested by &CCN_DSASZ for "main") by function "main" should be sufficient for AMODE 31. For AMODE 64, the extra stack space is roughly doubled.

The following new Global Set Symbols are introduced for the RENT option.

- &CCN_MAIN
- &CCN_RENT
- &CCN_WSA_INIT
- &CCN_WSA_TERM

For detailed information about these new Global Set Symbols, see "Compiler-generated global SET symbols" on page 13.

You can provide your own WSA initialization and termination routines by setting these Global Set Symbols with the module names of your own routines. For example:

```
GBLC &CCN_WSA_INIT
GBLC &CCN_WSA_TERM
&CCN_WSA_INIT SETC 'MYWSAI'
&CCN_WSA_TERM SETC 'MYWSAT'
```

Your own WSA initialization and termination routines can be object modules, load modules, or program modules, and they need to be supplied to the binder's input.

The compiler generated code for "main" has the following kind of assembly statements in it:

- For AMODE 31:

  ```
  DC V(&CCN_WSA_INIT)
  DC V(&CCN_WSA_TERM)
  ```

- For AMODE 64:

  ```
  DC VD(&CCN_WSA_INIT)
  DC VD(&CCN_WSA_TERM)
  ```

WSA initialization routine

Given the size of the WSA for the whole application and the image of the WSA with initialization data applied, the WSA initialization routine you provided dynamically allocate the WSA storage for the application and copy the WSA image into it. The address of the allocated storage is returned which CCNZINIT saves it on the function main's stack to be propagated to the rest of the application. You are responsible for ensuring that the allocated storage is addressable to all parts of the application. This particularly means if there are AMODE 31 parts in the application, the WSA storage should not be allocated above the 2G bar if the AMODE 31 parts need to access it. Also, the WSA storage has to be allocated in the primary address space. WSA storage in data spaces is not supported.

The routine you provide is given an address of an area to store whatever extra information you want to keep and pass to the WSA termination routine you provided. The storage area size is the size of a pointer, that is, 4 bytes for AMODE 31, and 8 bytes for AMODE 64.

Function prototype:

```
typedef void * (init_func_t) (void * wsa_image_addr,
unsigned long wsa_size, void * *user_info_addr, unsigned int alignment);
```

Input parameters:

- wsa_image_addr - the address of the WSA image in the loaded program object
- wsa_size - the total size of the application's WSA
- user_info_addr - the address of the CCNZINIT provided save area for saving user information
- alignment - the minimum required alignment of the allocated WSA storage. For example, "alignment=8" means double-word alignment.

Return value:

The address of the allocated and initialized WSA storage. The default is the IBM supplied routine CCNZWSAI or CCNZQWSI, which allocates storage for both AMODE 31 and AMODE 64 with the following macro:

```
STORAGE OBTAIN,LENGTH=(n),BNDRY=PAGE
```

WSA termination routine

This routine is called from CCNZTERM for the WSA termination and cleanup process. It is passed in the address of the WSA storage allocated by the WSA initialization routine. It is also given the same WSA size that was originally passed to the WSA initialization routine.

Function prototype:

```
typedef void (term_func_t) (void * allocated_wsa_addr,
unsigned long wsa_size, void * user_info_addr);
```

Input parameters:

- allocated_wsa_addr - the address of the allocated WSA storage
- wsa_size - the total size of the application's WSA
- user_info_addr - the saved user information

Return value:

The default is the IBM supplied routine CCNZWSAT or CCNZQWST, which frees the storage with the following macro:

```
STORAGE RELEASE,LENGTH=(n),ADDR=(m)
```

argc argv parsing support

If your `main()` function uses the standard `argc` and `argv` arguments, the Metal C initialization routine is called to parse the raw parameter data received from the hosting environment and to convert the parameter to the standard `argc` and `argv` format. If your program is not invoked in the z/OS UNIX System Services (USS) environment, you can use the ARGPARSE or NOARGPARSE options to determine if the EXEC PARM needs to be further parsed into individual arguments; the EXEC PARM has to be in this format: a halfword length field followed by a maximum of 100 characters where the length field contains a binary count of the number of bytes in the PARM field. For more information about the ARGPARSE option, see *z/OS XL C/C++ User's Guide*.

If your `main()` function uses `argc` and `argv` arguments and you do not want the parsing to be performed, you can set the new Global Set Symbol &CCN_APARSE to 0 in your prolog code to conditionally bypass the argument parsing. For detailed information, see Table 3 on page 15.

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*.

The following sections describe 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 of a simple dereference of a far pointer, see Figure 38 on page 42.

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 32 on page 35.

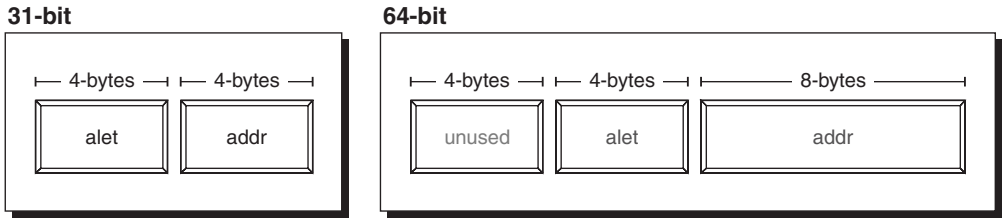


Figure 32. Far pointer sizes under different addressing modes

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.
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 &	The result is a normal pointer, except in the following cases: • 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 36 on page 39. 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.BUILTN to "1".
- Include the header file builtins.h.

Figure 33 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 33. Built-in functions for setting far-pointer components

Figure 34 on page 37 lists the extractors.

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

Figure 34. 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.BUILTN to "1".
- Include the header file `builtins.h`.

The semantics of these functions, listed in Figure 35, 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 37 on page 41.

Figure 35. 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 36 on page 39 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 36. 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 36. 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 36. Allocation and deallocation routines (Part 3 of 3)

Copying a string pointer to a far pointer

Figure 37 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 37. 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 38 on page 42 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 38. Example of a simple dereference of a far pointer

Building Metal C programs

Because the Metal option produces the final code in HLASM source code format, the build process needs to include an assembly step to produce the object files. The build process is demonstrated in Figure 39 on page 43. Note that the build process with IPA is more elaborated. For more information, see “Building Metal C programs with IPA” on page 46.

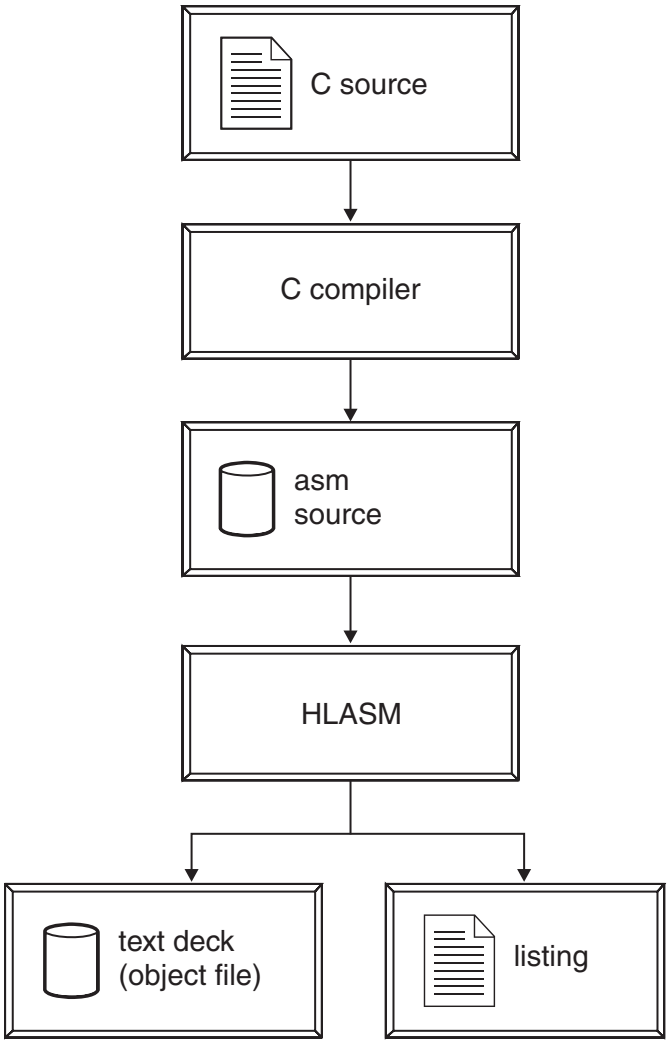


Figure 39. Metal C application build process

In summary, the C source file is sent to the C compiler, which generates the assembler source file. The assembler source file is sent to the HLASM assembler, which generates an object file and a listing.

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.
- Under MVS, if the C source file is a data set, the compiler uses the C source file name to form the name of the HLASM source file. The high-level qualifier is replaced with the userid under which the compiler is running, and .ASM is appended as the low-level qualifier.

C source file

Figure 40 shows a C source file mycode.c that can be used to generate an HLASM source file. The name of the generated HLASM 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 40. 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 HLASM source file.
2. Use the **as** command to generate the object file.
3. Use the **ld** command to generate the program.

Generating an HLASM source file using the xlc command: 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.

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 HLASM 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 HLASM source file name is mycode.s.

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

Figure 41. C compiler invocation to generate mycode.s

Generating an object file from the HLASM 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 42. The -o flag can be used to name the output file, where it can be a UNIX file or an MVS data set.

```
as mycode.s
```

Figure 42. Command that invokes HLASM to assemble mycode.s

A successful assemble will produce mycode.o.

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 43 on page 45.

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

Figure 43. Command that compiles an HLASM source file containing symbols longer than eight characters

A successful assemble will produce mycodelong.o.

Creating a program with the z/OS UNIX System Services ld command: Use the **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 should use **-b case=mixed** and the **-e** option must specify the entry point in its original case, as shown in Figure 44.

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

Figure 44. 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 45 shows the commands that compile and link the programs in Figure 31 on page 30.

```
xlc -S -qmetal a31.c                1
xlc -S -qmetal -q64 a64a.c          2
as -a=a31.lst -mgoff a31.s          3
as -a=a64a.lst -mgoff a64a.s        3
ld -o a.out a31.o a64a.o -e main    4
```

Figure 45. Commands that compile and link programs with different addressing modes

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 **ld** command links the object file produced by the **as** command into a program. The **-e** command specifies the entry point.

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.

After successful completion of the assembly step, 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 46. Job step that compiles HLQ.SOURCE.C(MYCODE)

Assembly of HLQ.SOURCE.ASM(MYCODE):

```
//-------------------------------------------------------------------
//* ASSEMBLY STEP:
//-------------------------------------------------------------------
//ASM     EXEC HLASMC
//SYSIN    DD  DSN=HLQ.SOURCE.ASM(MYCODE),DISP=SHR
//SYSLIN   DD  DSN=HLQ.OBJ(MYCODE),DISP=OLD
```

Figure 47. Assembly step of HLQ.SOURCE.ASM(MYCODE)

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

```
//*-------------------------------------------------------------------
//* BIND STEP:
//*-------------------------------------------------------------------
//BIND   EXEC PGM=IEWL,
//            PARM='AMODE=31,MAP,CASE=MIXED'
//SYSLMOD  DD  DSNAME=HLQ.LOAD(MYCODE),DISP=SHR  1
//SYSPRINT DD  SYSOUT=*
//OBJECT   DD  DSN=HLQ.OBJ,DISP=SHR
//SYSLIN   DD *
  INCLUDE OBJECT(MYCODE)
  ENTRY MYADD                                   2
/*
```

Figure 48. 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.

Building Metal C programs with IPA

Starting with z/OS V1R13 XL C compiler, the IPA option can be used with the METAL option. IPA is an optimization option that enables the compiler to find more optimization opportunities to improve your application performance. For more information about IPA, see the Using the IPA option section in *z/OS XL C/C++ Programming Guide* and the IPA considerations section in *z/OS XL C/C++ User's Guide*.

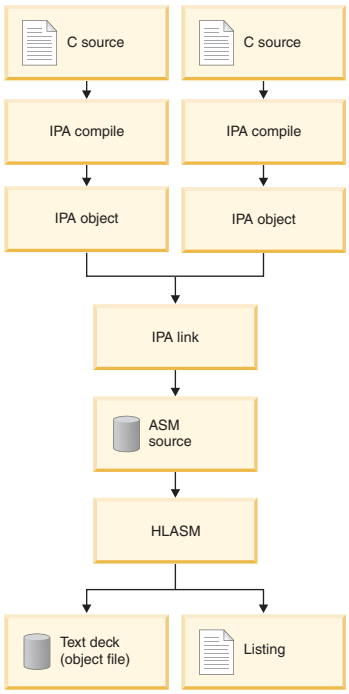


Figure 49. The process of building Metal C programs with IPA

You need to be aware of the following adjustments when invoking IPA for METAL.

- Switching addressing modes is not supported.
- The LONGNAME option is required.
- The IPA compile step only produces IPA object in the output file. Only IPA(NOOBJECT) is allowed, which instructs IPA to stop the compile process after the IPA object is produced. It does not produce HLASM source code, so the GENASM option cannot be used.
- The output file from IPA link step is one single HLASM source file for the whole program, and the GENASM option is required. There could be multiple structures in the HLASM source program, one for each partition. Under USS, the output HLASM source file resides in the directory where the IPA link took place. The default output file name for USS is a.s. In BATCH mode, the output HLASM source file goes in the dataset allocated to DD SYSLIN in the IPA link step.
- At the IPA link phase, all external references must be resolved. For Metal C, IPA does not attempt to convert external object modules or load modules into object code for the inclusion in the IPA produced program. You need to provide the same set of library data sets to both IPA link and the binder for symbol resolution.
- If you specify the PROLOG and EPILOG compiler options to supply your own prolog and epilog macros at compile time, the macros will only be applied to the functions defined in the source file.

- If you have #pragma insert_asm in your source file, IPA will assume the strong connection between the string provided by the pragma and the functions in the source file. IPA will not move functions defined in that source file to anywhere else.
- If you use global register variable or the RESERVE_REGS option during your compile, IPA link will merge the registers specified in the compile steps and apply the merged set of the originated compilation units to a partition.
- If you use the DSAUSER option in any of your compile steps, IPA link applies the option to the entire program.

The following compiler options are not supported by METAL with IPA:

- DEBUG
- REPORT

The following IPA sub-options are not supported with the METAL option:

- ATTRIBUTE
- GONUM
- PDF sub-options

The following IPA control file directives are not supported with the METAL option:

- EXPORT
- NOEXPORTS

Example: The following example shows how to compile a Metal C program with IPA.

IPA compile phase:

```
xlc -qmetal -qipa -c x.c
xlc -qmetal -qipa -c y.c
```

The above commands produce x.o and y.o.

Notes:

1. The -c option indicates compile.
2. No HLASM output is generated.
3. The objects are IPA objects, which can only be used for IPA link.
4. LONGNAME is implicitly turned on.

IPA link phase:

```
xlc -qmetal -qipa -S x.o y.o
```

This produces a.s.

Note: The structure of the compiler-generated HLASM source program is similar to that described in “Structure of a compiler-generated HLASM source program” on page 5, except that at IPA link there could be multiple structures in the HLASM source program, one for each partition.

The rest of the build process is similar to building Metal C programs without IPA. You need to add the assembly step to produce the object file from the IPA link generated HLASM source file. You also need to supply the object file produced by the assembler along with all other library data sets to the binder for producing the final executable program.

Assembly phase:

```
as -mgoff a.s
```

This produces a.o.

Note: The HLASM GOFF option must be specified because of the LONGNAME compiler option requirement with IPA.

Bind/Link phase:

```
ld -b case=mixed -e main a.o
```

This produces a.out.

Note: Because of the LONGNAME compiler option requirement with IPA, you should use the **-b case=mixed** ld utility option and the **-e** with the entry point in its original case.

Generation of debugging information

When the NOMETAL (the default) and DEBUG compiler options are in effect, the 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 and DEBUG compiler options are 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'          1
//SYSADATA DD DISP=SHR,DSN=hlq..SYSADATA  2
//ASMLANGX DD DISP=OLD,DSN=hlq..ASMLANGX  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 50. 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('hlq.ASMLANGX') SHR
   TSOLIB ACT DS('hlq.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 Linkage Conventions in *z/OS MVS Programming: Assembler Services Guide*, SA22-7605.
The Metal C Runtime Library.	Chapter 2, “Header files,” on page 53	For additional information about the Metal runtime library, see http://www.ibm.com/systems/z/zos/metalc/.

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

Information	Internal reference	External references
Using assembler statements within a C program.	• "Inserting HLASM instructions into the generated source code" on page 20 • "Inserting non-executable HLASM statements into the generated source code" on page 29	For detailed information about HLASM programming, see *HLASM MVS & VM Programmer's Guide*. For detailed information about inline assembly statements, see Inline assembly statements in *z/OS XL C/C++ Language Reference*. For more information about callable system services, see *z/OS MVS Programming: Callable Services for High-Level Languages*.
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 *z/OS XL C/C++ User's Guide*.
Making access registers available to the Metal C application.	"AR-mode programming support" on page 33	For detailed information about using access registers, see *z/OS MVS Programming: Extended Addressability Guide*.
Providing prolog and epilog code to customize the environment.	• "Compiler-generated global SET symbols" on page 13 • "Supplying your own prolog and epilog code" on page 12	*Not applicable.*
Building the application by using JCL procedures.	"Building Metal C programs using JCL" on page 45	*Not applicable.*
Building the application by using z/OS UNIX System Services.	"Building a Metal C program using z/OS UNIX System Services" on page 44	*Not applicable.*
Generating the appropriate debugging information.	"Generation of debugging information" on page 49	*Not applicable.*
Invoking the IDF debugger.	"IDF debugger invocation" on page 50	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 compiler, you must instruct the compiler to search this directory. There are a number of ways to do this.

Note: Some Metal C header files such as stdio.h have the same names as header files for the Language Environment C/C++ Run-Time Library. To avoid including these, or inadvertently including any other headers supported by the LE library and not by Metal C, remove the non-Metal libraries from the search order. Depending on how you specify the system library search path, you need to remove other libraries from the SYSLIB concatenation of the compiler, or specify the NOSEARCH compiler option before pointing to `/usr/include/metal/`.

If you are compiling in batch, you can use the SEARCH compiler option:

```
SEARCH(/usr/include/metal/)
```

If you are compiling using the NOSEARCH compiler option, you have the following options:

- Use the - I option of the xlc utility.

  ```
  -I /usr/include/metal/
  ```

- Use the cinc attribute in the xlc configuration file.

  ```
  cinc = -I /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 33.

ctype.h

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

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

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 on page 54 lists the definitions contained by float.h.

Table 7. Definitions in float.h

Constant	Description
FLT_RADIX	The radix for a z/OS XL C Metal C application. For FLOAT(IEEE), the value is 2.
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, q, such that any floating-point number with q decimal digits can be rounded into a floating-point number with p radix FLT_RADIX digits, and back again, without any change to the q decimal digits.
FLT_MIN_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 • 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 sprintf family for signed integers.

PRId8	PRId16	PRId32	PRId64
PRIdLEAST8	PRIdLEAST16	PRIdLEAST32	PRIdLEAST64
PRIdFAST8	PRIdFAST16	PRIdFAST32	PRIdFAST64
PRIdMAX			
PRIdPTR			
PRIi8	PRIi16	PRIi32	PRIi64
PRIiLEAST8	PRIiLEAST16	PRIiLEAST32	PRIiLEAST64
PRIiFAST8	PRIiFAST16	PRIiFAST32	PRIiFAST64
PRIiMAX			
PRIiPTR			

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

Macros for sprintf family for unsigned integers.

PRIo8	PRIo16	PRIo32	PRIo64
PRIoLEAST8	PRIoLEAST16	PRIoLEAST32	PRIoLEAST64
PRIoFAST8	PRIoFAST16	PRIoFAST32	PRIoFAST64
PRIoMAX			
PRIoPTR			
PRIu8	PRIu16	PRIu32	PRIu64
PRIuLEAST8	PRIuLEAST16	PRIuLEAST32	PRIuLEAST64
PRIuFAST8	PRIuFAST16	PRIuFAST32	PRIuFAST64
PRIuMAX			
PRIuPTR			
PRIx8	PRIx16	PRIx32	PRIx64
PRIxLEAST8	PRIxLEAST16	PRIxLEAST32	PRIxLEAST64
PRIxFAST8	PRIxFAST16	PRIxFAST32	PRIxFAST64
PRIxMAX			
PRIxPTR			
PRIX8	PRIX16	PRIX32	PRIX64
PRIXLEAST8	PRIXLEAST16	PRIXLEAST32	PRIXLEAST64
PRIXFAST8	PRIXFAST16	PRIXFAST32	PRIXFAST64
PRIXMAX			
PRIXPTR			

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`.

Symbolic name
Resource limit

CHAR_BIT
8

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: The definitions are as follows.

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 70.

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.

va_arg() va_copy() va_start() va_end()

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:

snprintf() sprintf() sscanf() vsnprintf() vsprintf() vsscanf()

The stdlio.h header file also contains definitions for the following macros, but you should not alter their value:

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.

- int*N*_t
- uint*N*_t

The following minimum-width integer types are defined.

- int_least*N*_t
- uint_least*N*_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_fast*N*_t
- uint_fast*N*_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.

- INT*N*_MAX
- INT*N*_MIN
- UINT*N*_MAX

Macros for limits of minimum width integer types.

- INT_LEAST*N*_MAX
- INT_LEAST*N*_MIN

- UINT_LEAST*N*_MAX

Macros for limits of fastest minimum width integer types.
- INT_FAST*N*_MAX
- INT_FAST*N*_MIN
- UINT_FAST*N*_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()	qsort()	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 58.

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:

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

string.h

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

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

[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 58.

Chapter 3. C functions available to Metal C programs

This topic 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 121 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.

User-replaceable heap services

The Metal C Runtime Library provides the ability to completely replace the underlying heap services at run time. You can use this function if you want the heap services to use a different storage management mechanism, for instance, one that is already in use elsewhere within an application.

A Metal C application replaces the underlying heap services by providing sets of function entry points in the *__csysenv_s* structure that is passed to the __cinit() function. To have the function entry point fields available and recognized by the __cinit() function, take the following steps:

- Define *__METAL_CSYSENV_VERSION 2* so that the *__csysenv_s* structure contains the heap service function entry point fields.
- In the *__csysenv_s* structure, set field *__cseversion* to *__CSE_VERSION_2*.
- In the *__csysenv_s* structure, provide addresses for heap functions that are to be replaced.
- Call the __cinit() function, providing the *__csysenv_s* structure that was initialized.

The Metal C application can provide at environment initialization time 8 bytes of data that can be accessed by the replacement heap services. To reserve the 8 bytes of data, take the following steps:

- Before calling the __cinit() function, set the user data of the application in field *__cseheapuserdata* in the *__csysenv_s* structure.
- Use the R12 environment token value as a pointer to the *__csysenvtoken_s* structure. In this structure, field *__csetheapuserdata* contains 8 bytes of data of the application.

During the __cinit() call, field *__csetheapuserdata* can only be set from *__cseheapuserdata* if heap services have been replaced; otherwise, field *__csetheapuserdata* will be set to binary zeroes.

Two sets of heap service function entry points are provided, one set for replacing heap services in the AMODE 31 version of the library, and the other set for replacing heap services in the AMODE 64 version of the library.

AMODE 31 heap services

To replace heap services in the AMODE 31 version of the library, consider the following *__csysenv_s* fields:

*void * (*__cseamode31malloc) (size_t)*
: When specified, MCRTL AMODE 31 malloc() calls this routine to obtain a piece of below-the-bar heap storage and returns its result to the caller of malloc(). *__cseamode31malloc* is treated as having the same function prototype as *malloc(): void * malloc (size_t);*

void (__cseamode31free) (void *)*
: When specified, MCRTL AMODE 31 free() calls this routine to free a piece of heap storage. *__cseamode31free* is treated as having the same function prototype as *free(): void free(void *);*

*void * (*__cseamode31realloc) (void *, size_t)*
: When specified, MCRTL AMODE 31 realloc() calls this routine to perform a realloc for a piece of heap storage and returns its result to the caller of

realloc(). __*cseamode31realloc* is treated as having the same function prototype as *realloc(): void * realloc (void *, size_t);*

Providing this routine is optional. If realloc() is called when a __*cseamode31malloc* routine has been provided but __*cseamode31realloc* has not, realloc() will return a zero.

Note: __*cseamode31malloc* and __*cseamode31free* must be provided together. __*cseamode31realloc* is optional, but when it is provided, the application must also include the other AMODE 31 heap services in this set.

AMODE 64 heap services

To replace heap services in the AMODE 64 version of the library, consider the following __*csysenv_s* fields:

*void * (* __cseamode64malloc) (size_t)*
When specified, MCRTL AMODE 64 malloc() calls this routine to obtain a piece of above-the-bar heap storage and returns to the caller of malloc(). __*cseamode64malloc* is treated as having the same function prototype as *malloc(): void * malloc (size_t);*

*void * (*__cseamode64malloc31) (size_t)*
When specified, MCRTL AMODE 64 __malloc31() calls this routine to obtain a piece of below-the-bar heap storage and returns its result to the caller of __malloc31(). __*cseamode64malloc31* is treated as having the same function prototype as *__malloc31(): void * __malloc31(size_t);*

void (__cseamode64free) (void *)*
When specified, MCRTL AMODE 64 free() calls this routine to free a piece of heap storage. __*cseamode64free* is treated as having the same function prototype as *free(): void free(void *);*

Note that MCRTL AMODE 64 free() accepts as input and processes heap storage that is allocated above or below the bar. The user-specified __*cseamode64free* routine must provide the same capability.

*void * (*__cseamode64realloc) (void *, size_t)*
When specified, MCRTL AMODE 64 realloc() calls this routine to perform a realloc for a piece of heap storage and returns its result to the caller of realloc(). __*cseamode64realloc* is treated as having the same function prototype as *realloc(): void * realloc (void *, size_t);*

Providing this routine is optional. If realloc() is called when a __*cseamode64malloc* routine has been provided but __*cseamode64realloc* has not, realloc() will return a zero.

Note that MCRTL AMODE 64 realloc() accepts as input and processes heap storage that is allocated above or below the bar. The user-specified __*cseamode64realloc* routine must provide the same capability.

Note: __*cseamode64malloc*, __*cseamode64malloc31*, and __*cseamode64free* must all be provided together. __*cseamode64realloc* is optional, but when it is provided, the application must also include the other AMODE 64 heap services in this set.

Usage notes

- Each heap service gets control in the AMODE of the calling service. The heap service must return to the calling service in that same AMODE.

- Each heap service is called using standard Metal C linkage conventions, including:
  - GPR 1 containing the address of the function parameter list (using C style parameter passing)
  - GPR 13 containing the address of a stack frame allocated on a contiguous Metal C stack

  GPR 12 contains the environment token representing the Metal C environment that is currently in use.
- It is not necessary to provide a replacement for the calloc() function. The calloc() function calls malloc() as part of its processing, so replacing malloc() indirectly alters the behavior of calloc() as well.
- When user-provided heap services are in use, the Metal C Runtime Library makes no attempt to keep track of any heap storage that has been allocated by the application. The application is entirely responsible for tracking its heap storage, and for freeing it after it calls __cterm() to terminate the Metal C environment.
- The heap allocation functions should return NULL when they are unable to obtain storage. The application is responsible for capturing its own diagnostic data when necessary.
- The Metal C Runtime Library expects the following alignment for the storage that is returned by the replacement heap services:
  - Storage returned from the below-the-bar heap (AMODE 64 __malloc31(), and AMODE 31 malloc()) is doubleword aligned.
  - Storage returned from the above-the-bar heap (AMODE 64 malloc()) is quadword aligned.

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 56
- “stdlib.h” on page 60
- “labs() — Calculate long absolute value” on page 77

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 60
- "atol() — Convert character string to long"
- "atoll() — Convert character string to signed long long" on page 69
- "strtol() — Convert Character String to Long" on page 109
- "strtoll() — Convert String to Signed Long Long" on page 112
- "strtoul() — Convert String to Unsigned Integer" on page 113
- "strtoull() — Convert String to Unsigned Long Long" on page 114

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 60
- "atoi() — Convert character string to integer"
- "atoll() — Convert character string to signed long long" on page 69
- "strtol() — Convert Character String to Long" on page 109
- "strtoll() — Convert String to Signed Long Long" on page 112
- "strtoul() — Convert String to Unsigned Integer" on page 113
- "strtoull() — Convert String to Unsigned Long Long" on page 114

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 about 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 60
- "atoi() — Convert character string to integer" on page 68
- "atol() — Convert character string to long" on page 68
- "strtol() — Convert Character String to Long" on page 109
- "strtoll() — Convert String to Signed Long Long" on page 112
- "strtoul() — Convert String to Unsigned Integer" on page 113
- "strtoull() — Convert String to Unsigned Long Long" on page 114

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 60
- "free() — Free a block of storage" on page 74
- "malloc() — Reserve storage block" on page 79
- "__malloc31() — Allocate 31–bit storage" on page 79
- "realloc() — Change reserved storage block size" on page 85

__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 8. 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       __csesubpool;      /* for 31 bit storage                 */
 __ptr31(void, __csetcbowner) /* owning TCB for resources           */
                              /* default: TCB mode - caller tcb,    */
                              /*   SRB. XMEM - CMRO TCB             */
 int       __csereserved;     /* Reserved field                     */
 char      __csettknowner[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
           __cseheap31incrsize; /* Minimum size, in bytes, to obtain
                                when expanding the AMODE 31 heap.
                                If 0, defaults to 32768 bytes        */
```

```
#ifdef __LL
 unsigned long long
           __cseheap64initsize; /* Minimum size, in MB, to obtain
                                for the initial AMODE 64 heap storage.
                                If 0, defaults to 1 MB               */
 unsigned long long
           __cseheap64incrsize; /* 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
           __cseheap64initsize_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
           __cseheap64incrsize_hh;
 unsigned int
           __cseheap64incrsize; /* 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 * myetkr12 __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.__csesubpool = 129;

  /*************************************************/
  /* Set heap initial and increment sizes.         */
  /*************************************************/
  mysysenv.__cseheap31initsize = 131072;
  mysysenv.__cseheap31incrsize = 8192;
  mysysenv.__cseheap64initsize = 20;
  mysysenv.__cseheap64incrsize = 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 9. csysenvtkn argument in __cterm()

Argument	Description
csysenvtkn	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 60
- “ldiv() — Compute quotient and remainder of integral division” on page 77
- “lldiv() — Compute quotient and remainder of integral division for long long type” on page 78

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.

Returned value

free() returns no value.

Related Information

- “stdlib.h” on page 60
- “calloc() — Reserve and initialize storage” on page 69
- “malloc() — Reserve storage block” on page 79
- “__malloc31() — Allocate 31–bit storage” on page 79
- “realloc() — Change reserved storage block size” on page 85

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.

isalnum() Test for an upper- or lowercase letter, or a decimal digit, as defined by code page IBM-1047.

isalpha() Test for an alphabetic character, as defined by code page IBM-1047.

isblank() Test for a blank character, as defined by code page IBM-1047.

iscntrl() Test for any control character, as defined by code page IBM-1047.

isdigit() Test for a decimal digit, as defined by code page IBM-1047.

isgraph() Test for a printable character excluding space, as defined by code page IBM-1047.

islower() Test for a lowercase character, as defined by code page IBM-1047.

isprint() Test for a printable character including space, as defined by code page IBM-1047.

ispunct() Test for any non-alphanumeric printable character, excluding space, as defined by code page IBM-1047.

isspace() Test for a white space character, as defined by code page IBM-1047.

isupper() Test for an uppercase character, as defined by code page IBM-1047.

isxdigit() 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 53
- “tolower(), toupper() — Convert Character Case” on page 116

isalpha() — Test for alphabetic character classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 74.

isblank() — Test for blank character classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 74.

iscntrl() — Test for control classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 74.

isdigit() — Test for decimal-digit classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 74.

isgraph() — Test for graphic classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 74.

islower() — Test for lowercase

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 74.

isprint() — Test for printable character classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 74.

ispunct() — Test for punctuation classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 74.

isspace() — Test for space character classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 74.

isupper() — Test for uppercase letter classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 74.

isxdigit() — Test for hexadecimal digit Classification

The information for this function is included in “isalnum() to isxdigit() — Test integer value” on page 74.

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 56
- “stdlib.h” on page 60
- “abs() — Calculate integer absolute value” on page 67
- “llabs() — Calculate absolute value of long long integer” on page 78

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 60
- “div() — Calculate quotient and remainder” on page 73
- “lldiv() — Compute quotient and remainder of integral division for long long type” on page 78

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 about 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 60
- "limits.h" on page 56
- "abs() — Calculate integer absolute value" on page 67
- "labs() — Calculate long absolute value" on page 77

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 about 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 60
- “div() — Calculate quotient and remainder” on page 73
- “ldiv() — Compute quotient and remainder of integral division” on page 77

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 60
- “calloc() — Reserve and initialize storage” on page 69
- “free() — Free a block of storage” on page 74
- “__malloc31() — Allocate 31–bit storage”
- “realloc() — Change reserved storage block size” on page 85

__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 60
- “calloc() — Reserve and initialize storage” on page 69
- “free() — Free a block of storage” on page 74
- “malloc() — Reserve storage block” on page 79
- “realloc() — Change reserved storage block size” on page 85

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 61
- “memchr() — Search buffer”
- “memcmp() — Compare bytes” on page 81
- “memcpy() — Copy buffer” on page 82
- “memmove() — Move buffer” on page 82
- “memset() — Set buffer to value” on page 83
- “strcpy() — Copy String” on page 99

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 61
- “memccpy() — Copy bytes in memory” on page 80
- “memcmp() — Compare bytes”
- “memcpy() — Copy buffer” on page 82
- “memmove() — Move buffer” on page 82
- “memset() — Set buffer to value” on page 83
- “strchr() — Search for Character” on page 98

---

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 *buf1* less than the contents of the buffer pointed to by *buf2*
= 0	The contents of the buffer pointed to by *buf1* identical to the contents of the buffer pointed to by *buf2*
> 0	The contents of the buffer pointed to by *buf1* greater than the contents of the buffer pointed to by *buf2*

Related Information

- “string.h” on page 61
- “memccpy() — Copy bytes in memory” on page 80
- “memchr() — Search buffer” on page 80
- “memcpy() — Copy buffer” on page 82
- “memmove() — Move buffer” on page 82
- “memset() — Set buffer to value” on page 83
- “strcmp() — Compare Strings” on page 99

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 61
- “memccpy() — Copy bytes in memory” on page 80
- “memchr() — Search buffer” on page 80
- “memmove() — Move buffer”
- “memset() — Set buffer to value” on page 83
- “strcpy() — Copy String” on page 99

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 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 61
- “memccpy() — Copy bytes in memory” on page 80
- “memchr() — Search buffer” on page 80
- “memcpy() — Copy buffer”
- “memset() — Set buffer to value” on page 83

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 61
- "memccpy() — Copy bytes in memory" on page 80
- "memchr() — Search buffer" on page 80
- "memcpy() — Copy buffer" on page 82
- "memmove() — Move buffer" on page 82

qsort() — Sort array

Format

```
#include <stdlib.h>

void qsort(void *base, size_t num, size_t width,
           int(*compare)(const void *element1, const void *element2));
```

General description

The qsort() function sorts an array of *num* elements, each of *width* bytes in size, where the first element of the array is pointed to by *base*.

The *compare* pointer points to a function, which you supply, that compares two array elements and returns an integer value specifying their relationship. The qsort() function calls the comparison function one or more times during the sort, passing pointers to two array elements on each call. The comparison function must compare the elements and return one of the following values:

Value	Meaning
< 0	*element1* less than *element2*
0	*element1* equal to *element2*
> 0	*element1* greater than *element2*

The sorted array elements are stored in increasing order, as returned by the comparison function. You can sort in reverse order by reversing the "greater than" and "less than" logic in the comparison function. If two elements are equal, their order in the sorted array is unspecified. The qsort() function overwrites the contents of the array with the sorted elements.

Returned value

The qsort() function returns no values.

Related information

- “stdlib.h” on page 60

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 60
- “rand_r() — Pseudo-random number generator”
- “srand() — Set Seed for rand() Function” on page 92

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 60
- "rand() — Generate random number" on page 84
- "srand() — Set Seed for rand() Function" on page 92

---

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 60
- "calloc() — Reserve and initialize storage" on page 69
- "free() — Free a block of storage" on page 74
- "malloc() — Reserve storage block" on page 79
- "__malloc31() — Allocate 31–bit storage" on page 79

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 58
- “sprintf() — Format and Write Data”
- “sscanf() — Read and Format Data” on page 92

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.

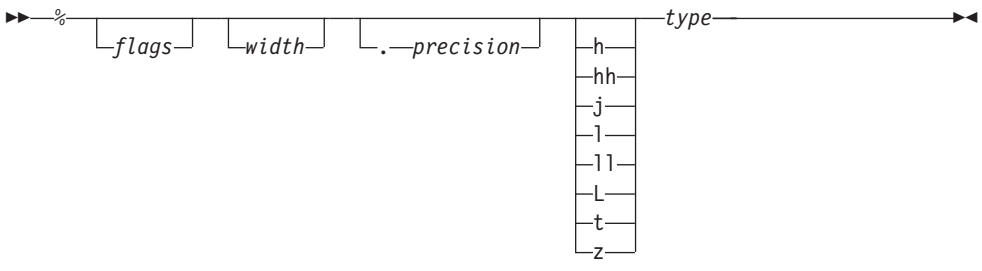
The *format-string* consists of ordinary characters, escape sequences, and conversion specifications. The ordinary characters are copied in order of their appearance. Conversion specifications, beginning with a percent sign (%) or the sequence (%n$) where n is a decimal integer in the range [1,NL_ARGMAX], 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.

Format Specification for sprintf()



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

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

The flag characters: The *flag* characters in Table 10 on page 88 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 10. Flag Characters for sprintf() Family

Flag	Meaning	Default
'	The integer portion of the result of a decimal conversion (%i,%d,%u, %f,%g or %G) will be formatted with the thousands' grouping characters.	No grouping.
–	Left-justify the result within the field width.	Right-justify.
+	Prefix the output value with a sign (+ or –) if the output value is of a signed type.	Sign appears only for negative signed values (–).
blank(' ')	Prefix the output value with a blank if the output value is signed and positive. The + flag overrides the *blank* flag if both appear, and a positive signed value will be output with a sign.	No blank.
#	When used with the `o`, `x`, or `X` formats, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively. 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: Definition of the *width* specification is as follows.

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: Definition of the *precision* specification is as follows.

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 11.

Table 11. Precision Argument in sprintf()

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

Table 11. Precision Argument in sprintf() (continued)

Type	Meaning	Default
g G	*Precision* 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 12 below shows the meaning of the type characters used in the precision argument.

Table 12. 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 *precision* is reached.
n	Pointer to integer	Number of characters successfully output so far to the *stream* or buffer; this value is stored in the integer whose address is given as the argument.

Table 12. Type Characters and their Meanings (continued)

Type	Argument	Output Format
p	Pointer	Pointer to void converted to a sequence of printable characters. See the individual system reference guides for the specific format.
f, F	Double	Signed value having the form [-]dddd.dddd, where dddd is one or more decimal digits. The number of digits before the decimal-point depends on the magnitude of the number. The number of digits after the decimal-point is equal to the requested precision. 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. 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. Convert a double argument representing a NaN in one of the styles: • [+/-]nan(n) for a signaling nan. • [+/-nanq(n)] for a quiet nan, where n is an integer and 1<= n<= INT_MAX-1. 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. The F conversion specifier produces INFe, NANS, or NANQ instead of infQ, nans or, nanq respectively.
e, E	Double	Signed value having the form [-]d.dddde[sign]ddd: • d is a single-decimal digit. • dddd is one or more decimal digits. • ddd is 2 or more decimal digits. • sign is + or -. If the *precision* 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. A double argument representing an infinity or NaN is converted in the style of an f or F conversion specifier.
g, G	Double	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 *precision*. Trailing zeros are truncated, and the decimal-point appears only if one or more digits follow it or a # flag is present. A double argument representing an infinity or NaN is converted in the style of an f or F conversion specifier.

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 58
- “snprintf() — Format and write data” on page 86
- “sscanf() — 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 60
- “rand() — Generate random number” on page 84
- “rand_r() — Pseudo-random number generator” on page 84

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*.

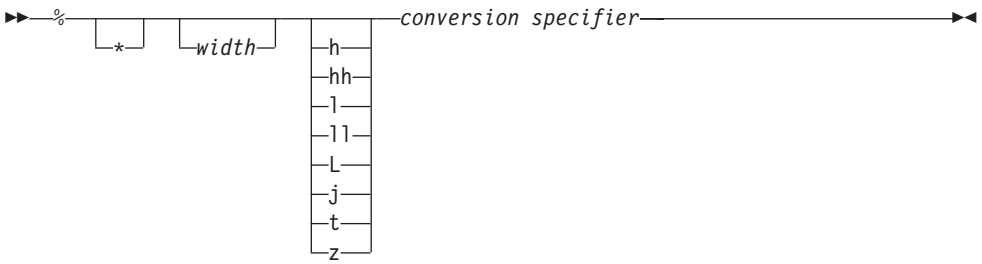
The *format-string* can contain either form of the conversion specification, that is, % or %n$ but the two forms cannot be mixed within a single *format-string* except that %% or %* can be mixed with the %n$ form.

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 13 on page 95 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.

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 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 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.

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 13.

Table 13. 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 *stream* or buffer.	Pointer to `int`, into which is stored the number of characters successfully read from the *stream* or buffer up to that point in the call to either fscanf() or to scanf().

Table 13. 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`.
[	A non-empty sequence of bytes to be matched against a set of expected bytes (the *scanset*), which form the conversion specification. White space characters that are ordinarily skipped are read when `%[` is specified. Consider the following situations: [^*bytes*]. In this case, the scanset contains all bytes that do not appear between the circumflex and the right square bracket. []abc] *or* [^]abc.] In both these cases the right square bracket is included in the scanset (in the first case:]abc and in the second case, *not*]abc) [a–z] In EBCDIC The – is in the scanset, the characters b through y are *not* in the scanset; in ASCII The – is *not* in the scanset, the characters b through y are. The code point for the square brackets ([and]) and the caret (^) vary among the EBCDIC encoded character sets. The default C locale expects these characters to use the code points for encoded character set Latin-1 / Open Systems 1047. Conversion proceeds one byte at a time: there is no conversion to wide characters.	Pointer to the initial byte of an array of char, signed char, or unsigned char large enough to accept the sequence and a terminating byte, which will be added automatically.
e E f F g G	Floating-point value consisting of an optional sign (+ or -), a series of one or more decimal digits possibly containing a decimal-point, and an optional exponent (`e` or `E`) followed by a possibly signed integer value.	Pointer to float

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 `%a`c, where *a* is a decimal integer. In this instance, the `c` conversion specifier means that the argument is a pointer to a character array. The next *a* characters are read from the input stream into the specified location, and no NULL character is added.

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

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 58
- “snprintf() — Format and write data” on page 86
- “sprintf() — Format and Write Data” on page 86
- “vsnprintf() — Format and print data to fixed length buffer” on page 118
- “vsscanf() — Format Input of a STDARG Argument List” on page 119

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

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 61
- “strchr() — Search for Character”
- “strcmp() — Compare Strings” on page 99
- “strcpy() — Copy String” on page 99
- “strcspn() — Compare Strings” on page 100
- “strncat() — Concatenate Strings” on page 101

---

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 61
- “memchr() — Search buffer” on page 80
- “strcat() — Concatenate Strings” on page 97
- “strcmp() — Compare Strings” on page 99
- “strcpy() — Copy String” on page 99
- “strcspn() — Compare Strings” on page 100
- “strncmp() — Compare Strings” on page 102
- “strpbrk() — Find Characters in String” on page 103
- “strrchr() — Find Last Occurrence of Character in String” on page 104
- “strspn() — Search String” on page 104

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 *string1* less than string pointed to by *string2*
= 0	String pointed to by *string1* equivalent to string pointed to by *string2*
> 0	String pointed to by *string1* greater than string pointed to by *string2*

Related Information

- "string.h" on page 61
- "memcmp() — Compare bytes" on page 81
- "strcspn() — Compare Strings" on page 100
- "strncmp() — Compare Strings" on page 102
- "strpbrk() — Find Characters in String" on page 103
- "strrchr() — Find Last Occurrence of Character in String" on page 104
- "strspn() — Search String" on page 104

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.

Returned Value

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

Related Information

- "string.h" on page 61
- "memcpy() — Copy buffer" on page 82
- "strcat() — Concatenate Strings" on page 97
- "strchr() — Search for Character" on page 98
- "strcmp() — Compare Strings" on page 99
- "strcspn() — Compare Strings"
- "strncpy() — Copy String" on page 103
- "strpbrk() — Find Characters in String" on page 103
- "strrchr() — Find Last Occurrence of Character in String" on page 104
- "strspn() — Search String" on page 104

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 61
- "strcat() — Concatenate Strings" on page 97
- "strchr() — Search for Character" on page 98
- "strcmp() — Compare Strings" on page 99
- "strcpy() — Copy String" on page 99
- "strncmp() — Compare Strings" on page 102
- "strpbrk() — Find Characters in String" on page 103
- "strrchr() — Find Last Occurrence of Character in String" on page 104
- "strspn() — Search String" on page 104

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 61
- "free() — Free a block of storage" on page 74
- "malloc() — Reserve storage block" on page 79

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 61
- "strncat() — Concatenate Strings"
- "strncmp() — Compare Strings" on page 102
- "strncpy() — Copy String" on page 103

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 61
- "strcat() — Concatenate Strings" on page 97
- "strncmp() — Compare Strings"
- "strncpy() — Copy String" on page 103
- "strpbrk() — Find Characters in String" on page 103
- "strrchr() — Find Last Occurrence of Character in String" on page 104
- "strspn() — Search String" on page 104

---

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 *substring1* less than string pointed to by *substring2*
= 0	String pointed to by *substring1* equivalent to string pointed to by *substring2*
> 0	String pointed to by *substring1* greater than string pointed to by *substring2*

Related Information

- "string.h" on page 61
- "memcmp() — Compare bytes" on page 81
- "strcmp() — Compare Strings" on page 99
- "strcspn() — Compare Strings" on page 100
- "strncat() — Concatenate Strings" on page 101
- "strncpy() — Copy String" on page 103
- "strpbrk() — Find Characters in String" on page 103
- "strrchr() — Find Last Occurrence of Character in String" on page 104
- "strspn() — Search String" on page 104

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 61
- “memcpy() — Copy buffer” on page 82
- “strcpy() — Copy String” on page 99
- “strncat() — Concatenate Strings” on page 101
- “strncmp() — Compare Strings” on page 102
- “strpbrk() — Find Characters in String”
- “strrchr() — Find Last Occurrence of Character in String” on page 104
- “strspn() — Search String” on page 104

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 61
- “strchr() — Search for Character” on page 98
- “strcspn() — Compare Strings” on page 100
- “strncmp() — Compare Strings” on page 102
- “strrchr() — Find Last Occurrence of Character in String” on page 104
- “strspn() — Search String” on page 104

strrchr() — Find Last Occurrence of Character in String

Format

```
#include <string.h>

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

General Description

The strrchr() 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, strrchr() returns a pointer to the last occurrence of *c* in *string*.

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

Related Information

- "string.h" on page 61
- "memchr() — Search buffer" on page 80
- "strchr() — Search for Character" on page 98
- "strcspn() — Compare Strings" on page 100
- "strncmp() — Compare Strings" on page 102
- "strpbrk() — Find Characters in String" on page 103
- "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 61
- "strcat() — Concatenate Strings" on page 97
- "strchr() — Search for Character" on page 98
- "strcmp() — Compare Strings" on page 99
- "strcpy() — Copy String" on page 99
- "strcspn() — Compare Strings" on page 100
- "strpbrk() — Find Characters in String" on page 103
- "strrchr() — 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 61
- "strchr() — Search for Character" on page 98
- "strcmp() — Compare Strings" on page 99
- "strcspn() — Compare Strings" on page 100
- "strncmp() — Compare Strings" on page 102
- "strpbrk() — Find Characters in String" on page 103
- "strrchr() — Find Last Occurrence of Character in String" on page 104
- "strspn() — Search String" on page 104

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:

- 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 92 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 60
- “atoi() — Convert character string to integer” on page 68
- “atol() — Convert character string to long” on page 68
- “sscanf() — Read and Format Data” on page 92
- “strtol() — Convert Character String to Long” on page 109
- “strtof — Convert Character String to Float”
- “strtold — Convert Character String to Long Double” on page 111
- “strtoul() — Convert String to Unsigned Integer” on page 113
- “vsscanf() — Format Input of a STDARG Argument List” on page 119

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 92 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 60
- “atoi() — Convert character string to integer” on page 68
- “atol() — Convert character string to long” on page 68
- “sscanf() — Read and Format Data” on page 92
- “strtod — Convert Character String to Double” on page 105
- “strtol() — Convert Character String to Long” on page 109
- “strtold — Convert Character String to Long Double” on page 111
- “strtoul() — Convert String to Unsigned Integer” on page 113
- “vsscanf() — Format Input of a STDARG Argument List” on page 119

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 61
- “strcat() — Concatenate Strings” on page 97
- “strchr() — Search for Character” on page 98
- “strcmp() — Compare Strings” on page 99
- “strcpy() — Copy String” on page 99
- “strcspn() — Compare Strings” on page 100
- “strspn() — Search String” on page 104

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 61
- “strcat() — Concatenate Strings” on page 97
- “strchr() — Search for Character” on page 98
- “strcmp() — Compare Strings” on page 99
- “strcpy() — Copy String” on page 99
- “strcspn() — Compare Strings” on page 100
- “strspn() — Search String” on page 104

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.

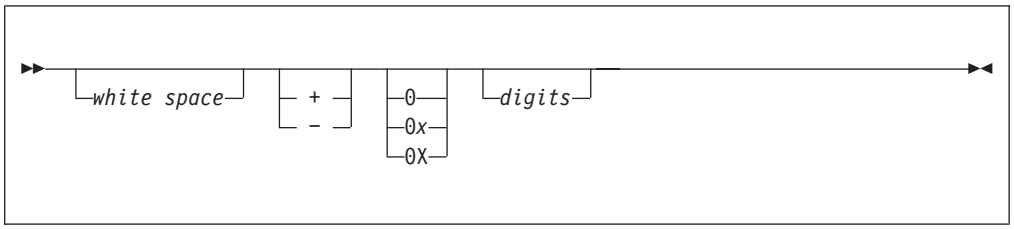
10 Sequence starts with nonzero decimal digit.

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 60
- “atoi() — Convert character string to integer” on page 68
- “atol() — Convert character string to long” on page 68
- “atoll() — Convert character string to signed long long” on page 69
- “sscanf() — Read and Format Data” on page 92
- “strtoul() — Convert String to Unsigned Integer” on page 113

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 92 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 60
- “atoi() — Convert character string to integer” on page 68
- “atol() — Convert character string to long” on page 68
- “sscanf() — Read and Format Data” on page 92
- “strtod — Convert Character String to Double” on page 105
- “strtof — Convert Character String to Float” on page 106
- “strtol() — Convert Character String to Long” on page 109
- “strtoul() — Convert String to Unsigned Integer” on page 113
- “vsscanf() — Format Input of a STDARG Argument List” on page 119

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 about 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.

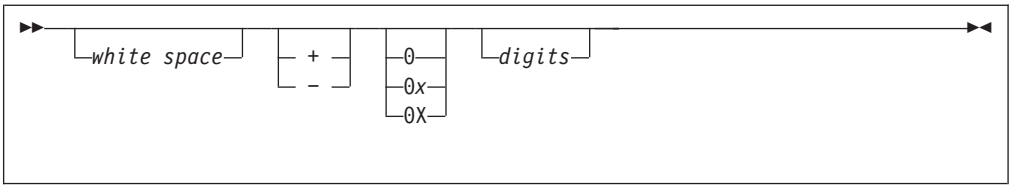
10 Sequence starts with nonzero decimal digit.

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, strtoll() returns the converted signed long long value, represented in the string.

If unsuccessful, strtoll() returns 0 if no conversion could be performed. If the correct value is outside the range of representable values, strtoll() returns `LLONG_MAX` (`LONGLONG_MAX`) or `LLONG_MIN` (`LONGLONG_MIN`), according to the sign of the value. If the value of base is not supported, strtoll() returns 0.

Related Information

- “stdlib.h” on page 60
- “atoi() — Convert character string to integer” on page 68
- “atol() — Convert character string to long” on page 68
- “atoll() — Convert character string to signed long long” on page 69
- “sscanf() — Read and Format Data” on page 92
- “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.

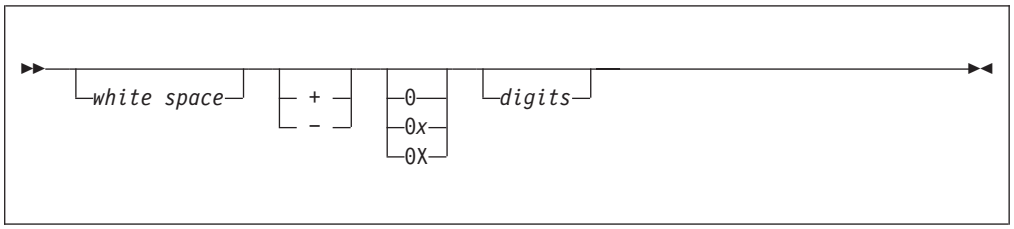
10 Sequence starts with nonzero decimal digit.

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 60
- “atoi() — Convert character string to integer” on page 68
- “atol() — Convert character string to long” on page 68
- “atoll() — Convert character string to signed long long” on page 69
- “sscanf() — Read and Format Data” on page 92
- “strtol() — Convert Character String to Long” on page 109

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 about how to make long long available.

General Description

The strtoull() 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.

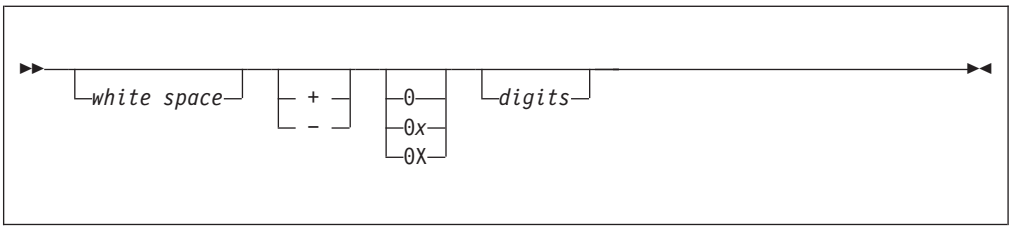
10 Sequence starts with nonzero decimal digit.

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 strtoull() 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 strtoull() 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, strtoull() returns the converted unsigned long long value, represented in the string.

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 (ULONGLONG_MAX). If the value of base is not supported, strtoull() returns 0.

Related Information

- "stdlib.h" on page 60
- "atoi() — Convert character string to integer" on page 68
- "atol() — Convert character string to long" on page 68
- "atoll() — Convert character string to signed long long" on page 69
- "sscanf() — Read and Format Data" on page 92
- "strtoul() — Convert String to Unsigned Integer" on page 113

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 53
- "isalnum() to isxdigit() — Test integer value" on page 74

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: See the sample code below.

```
#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 58
- “vsnprintf() — Format and print data to fixed length buffer”
- “vsprintf() — Format and Print Data to Buffer” on page 119

---

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 86.

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 58
- “stdio.h” on page 58
- “va_arg(), va_copy(), va_end(), va_start() — Access Function Arguments” on page 116

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 86.

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 58
- “stdio.h” on page 58
- “va_arg(), va_copy(), va_end(), va_start() — Access Function Arguments” on page 116

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.
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 92.

Related Information

- “stdarg.h” on page 58
- “stdio.h” on page 58
- “sscanf() — Read and Format Data” on page 92

Appendix A. Function stack requirements

Table 14 lists the stack frame requirements for each Metal C runtime function. All sizes are in bytes.

Table 14. 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
iscntrl	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
memccpy	512	512
memchr	512	512
memcmp	512	512
memcpy	512	512
memmove	512	512
memset	256	512
qsort	1280[1]	1792[1]
rand	256	512

Table 14. Stack frame requirements for Metal C runtime functions (continued)

Function	AMODE 31 stack size	AMODE 64 stack size
rand_r	256	512
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

Table 14. Stack frame requirements for Metal C runtime functions (continued)

Function	AMODE 31 stack size	AMODE 64 stack size
strtoll	1024	1024
strtoul	1024	1024
strtoull	768	1024
tolower	256	512
toupper	256	512
vsnprintf	3072	3584
vsnprintf when using e, E, f, F, g, G conversion specifiers	32000	32768
vsnprintf 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

Notes:

1. You must add the stack size of the comparison function you supply to the base value.

Appendix B. CICS programming interface examples

CICS Transaction Server for z/OS offers a number of programming interfaces. The application programming interfsace is widely used by CICS transactions running in a CICS environment. The exit programming interface can be used in a restricted environment that enables the customization of CICS to specific requirements, such as in global user exit programs.

The CICS application programming interface (CICS API) can be used from programs written in any of the high level programming languages supported by CICS, as well as in assembler programs. However, the exit programming interface (XPI) is provided to allow access to some of the CICS services from user exit programs, and these programs must be written in assembler. With the new support in this release, it is now possible for a C language program using the Metal C option to also use the CICS exit programming interface. This opens up the possibility of writing global user exit code using Metal C as a high level language alternative to assembler.

This topic contains some programming examples that demonstrate how the CICS XPI and the CICS API can be used in Metal C.

Runtime environment adapter

A Metal C “main” code in CICS requires the following capabilities.

- prolog code for environment initialization
- epilog code for environment termination
- writable static area (WSA) initialization plug-in
- writable static area (WSA) termination plug-in

In CICS, the prolog and epilog code are mandatory because the Metal C default prolog and epilog obtain storage using the MVS STORAGE macro. In CICS, storage should be obtained using CICS storage management API commands, and the execution environment should be set up by the DFHEIENT macro. The prolog code is mandatory, and the corresponding epilog code should also be provided.

WSA initialization and termination plug-in code should be provided. A C “main” program can be coded which does not use static data, but the CICS API injects static data into the code.

CICS programs should be reentrant, so a CICS “main” program must be compiled with the Metal C RENT option specified to meet this CICS requirement.

A Metal C subroutine requires the following capabilities.

- Optional prolog code for environment initialization
- Optional epilog code for environment termination

Subroutines do not cause writable static areas to be generated. When you write subroutines, natural reentrancy must be maintained. The CICS exit programming example provides an illustration of on how this is done.

Under CICS, the default Metal subroutine prolog and epilog code can be used if the space allocated for the execution stack does not run out.

CICS application programming interface example

The CICS application programming interface example consists of the following components.

MTLBOOT	Assembler bootstrap program
MTLHALO	Metal C “Hello World” using the CICS API
MTLENT	“main” prolog macro
MTLXIT	“main” epilog macro
MTLSENT	Subroutine prolog macro
MTLSXIT	Subroutine epilog macro

Data structures

The MTLBOOT assembler program defines the following data structures.

- PLISTINIT, which defines the WSA initialization parameters
- PLISTTRM, which defines the WSA termination parameters
- main_plist, which defines the “main” program entry parameters
- the application execution stack with the stack header defined by stack_hdr

Figure 51 illustrates how the API components and entry points used in the examples are related.

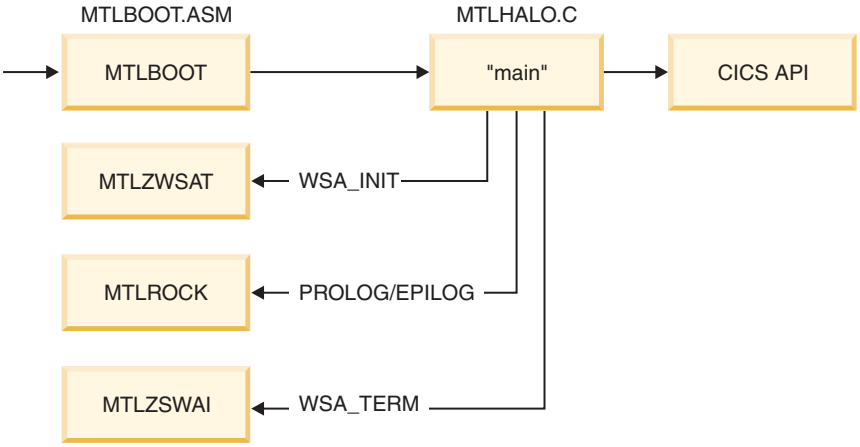


Figure 51. CICS API example flow

Example description

MTLBOOT

The MTLBOOT serves as the adapter program that sets up the proper CICS execution environment for assembler code. It provides the following services.

- sets up the CICS execution environment for assembler
- obtains the program stack storage for the Metal C program execution
- a callback service to obtain additional stack storage
- a callback service to free additional stack storage
- return to CICS after program execution
- the WSA initialization plug-in

- the WSA termination plug-in

The stack storage provided to the Metal C code has a header described by the stack_hdr dsect. The fields that are not standard are bootstg and mtlrock_ep. The bootstg field is a fullword containing a pointer to the CICS allocated dynamic save area associated with the bootstrap program. This is analogous to the “this” pointer in C++. The mtlrock_ep field is the entry point to the callback routine. An input parameter provided to the callback routine determines whether getmain or freemain is used to allocate storage.

MTLHALO
MTLHALO represents the application code. This is a simple example which demonstrates how the CICS API is used under Metal C. It also demonstrates the use of the prolog and epilog examples. This is the code that you would replace with your own application requirements.

MTLENT
MTLENT is mandatory for the “main” program. In this example set, the complexity is encapsulated within the bootstrap program so that the main prolog macro can be simple and reusable. This macro assumes that it is invoked with an entry environment described by the “main_plist”.

MTLXIT
MTLXIT is the “main” program exit macro.

MTLSENT
MTLSENT is an optional macro for subroutine entry. It provides an example of what can be done if additional stack storage needs to be obtained. This macro provides an example of how to invoke the callback routines in the bootstrap program.

MTLSXIT
MTLSXIT is an optional subroutine exit macro. If MTLSENT is used for entry, MTLSXIT should be used for exit.

Example code

Figure 52 on page 128 contains the bootstrap for using Metal C with CICS API programs.

```
*ASM     XOPTS(NOPROLOG NOEPILOG SP)
*****************************************************************
*                                                               *
* Module Name = MTLBOOT                                         *
*                                                               *
* Descriptive Name = CICS Bootstrap for metal C code example    *
*                                                               *
* @BANNER_START                          02                     *
*                       MTLBOOT                                 *
* Licensed Materials - Property of IBM                          *
*                                                               *
* "Restricted Materials of IBM"                                 *
*                                                               *
* 5655-S97                                                      *
*                                                               *
* (C) Copyright IBM Corp. 1989, 2009                            *
*                                                               *
* Description                                                   *
*   The bootstrap routine sets up stack storage and flows       *
*   control to the metal c program. It provides callbacks for   *
*   the WSA init/term plugin functionality. It also provides    *
*   the getmain/freemain entry points which are invoked by the  *
*   subroutine prolog/epilog routines.                          *
*                                                               *
*   This code also provides an implementation proof-of-concept  *
*   if the user wants to create a CICS API layer in assembler   *
*   and the business layer in metal c. This code can be extended*
*   with the code to implement the CICS API.                    *
*                                                               *
*****************************************************************
* The Metal C Bootstrap program.                                *
*****************************************************************
*
*****************************************************************
* Input parm list                                               *
*****************************************************************
PLISTTRM      DSECT       WSA termination plugin parameters
trm_addr      DS   F
trm_size      DS   F
trm_user_ptr  DS   F
*
PLISTINIT     DSECT       WSA initialization plugin parameters
init_addr     DS   F
init_size     DS   F
init_user_ptr DS   F
init_align    DS   F
*****************************************************************
* define the stack block control area                           *
*****************************************************************
stack_hdr     dsect
bootstg       ds   f       metal boot reg 12 contents
mtlrock_ep    ds   a       callback for getmain/freemain
blk_beg_addr  ds   a       begin address of block
blk_end_addr  ds   a       end address of block
lstack_hdr    equ  *-stack_hdr
*
metlget       equ  x'0001'
metlfre       equ  x'0002'
*
              copy dfhkebrc
```

Figure 52. CICS Bootstrap for metal C code example: MTLBOOT (Part 1 of 7)

```
*****************************************************************
* Program state                                                 *
*****************************************************************
         DFHEISTG
*
*  local working storage
*
booteye  ds     cl8
bootrsa  DS     18F
mtlrsa   ds     7F
*
AREA     DS     A
NAB      DS     A
RETCD    DS     F
*
main_plist  ds  0a
hdr_ptr     ds  a
stack_ptr   ds  a
org_r1      ds  a
*
dsapool  ds     10f        array of pointers to dsablocks
next_free ds    H          array index
*
         DFHEIEND
         DFHREGS ,
*
MTLBOOT  CSECT
MTLBOOT  DFHEIENT CODEREG=(R3),DATAREG=(R12),EIBREG=(R11)
MTLBOOT  AMODE  31
MTLBOOT  RMODE  31
*
* logic
* getmain 'main' working storage
* format R1 content
* dispatch
*
         LARL  R10,CONSTANTS
         USING CONSTANTS,R10
*
         mvc   booteye,=cl8'>MTLROCK'  set eyecatcher in anchor
         LA    R13,bootrsa    point R13 to local save
         st    r1,org_r1      save user plist register
         EXEC CICS ADDRESS EIB(DFHEIBR)
         EXEC CICS GETMAIN SET(R4) FLENGTH(DSA_SIZE) RESP(RETCD)
         clc  retcd,dfhresp(normal) check getmain error
         jne    main_abend
         using stack_hdr,r4
         st     r12,bootstg    work area for metal rock callback
         mvc    mtlrock_ep,callback_ep
         st     r4,blk_beg_addr
         lr     r5,r4          compute end address
         a      r5,dsa_size   *
         st     r5,blk_end_addr
*
```

Figure 52. CICS Bootstrap for metal C code example: MTLBOOT (Part 2 of 7)

```
         xr     r5,r5         store block address in pool ctrl
         la     r6,dsapool
         st     r4,0(r5,r6)   *
         la     r5,1(,r5)     increment index
         sth    r5,next_free  *
         drop   r4
*
         la     r5,lstack_hdr(,r4)  point to user area
         st     r4,hdr_ptr
         st     r5,stack_ptr
         la     r1,main_plist set plist register
*
* flow control to the metal c program
*
         L      R15,VMAIN
         BASR   R14,R15
* freemain whatever we've getmained
         j      main_return
*
main_abend  ds  0h
         EXEC  CICS ABEND ABCODE('CMTL')
main_return ds  0h
         DFHEIRET
         drop   r3
*
*****************************************************************
* The MTLROCK callback service                                  *
* The input environment is as follows;                          *
*  R2 - the function code                                       *
*  R1 - the boot token, the mtlboot eistg pointer. This is      *
*       important for getmain especially, because at the time   *
*       getmain for stack storage is invoked, there's no more   *
*       storage space to use.                                   *
*       C++ developers can think of this as the 'this' pointer  *
* On return from a getmain                                      *
*  R2 - block pointer                                           *
*  R1 - the user section                                        *
*                                                               *
* The getmained block pointer is stored in an array of 10       *
* elements. No check is make for overflow.                      *
*****************************************************************
         ENTRY MTLROCK
MTLROCK  DS    0H
         stm   r10,r0,mtlrsa-DFHEISTG(r1) save used regs
         lr    r12,r1               set token in r12
*
LETSROCK DS    0H
         larl  r10,constants
         EXEC CICS ADDRESS EIB(DFHEIBR)
*
         chi   r2,metlget           do we getmain
         je    getmain
```

Figure 52. CICS Bootstrap for metal C code example: MTLBOOT (Part 3 of 7)

```
*****************************************************************
* else it's a freemain. Since we are maintaining a stack,      *
* the last getmained block is the first to be freed            *
*****************************************************************
*
* freemain logic
         lh    r5,next_free  obtain block address to free
         bctr  r5,0          some validation needed here
         sll   r5,2          *
         la    r6,dsapool    *
         l     r4,0(r5,r6)   *
         EXEC CICS FREEMAIN DATAPOINTER(R4) RESP(RETCD)
         clc   retcd,dfhresp(normal)
         jne   freemain_abend *
         xr    r4,r4         clear the array location
         st    r4,0(r5,r6)   *
         srl   r5,2          decrement the array index
         sth   r5,next_free  *
         j     retrock
freemain_abend ds  0h
         EXEC  CICS ABEND ABCODE('FMTL')
         j     retrock
*
*****************************************************************
* getmain goes in here                                         *
*****************************************************************
getmain  ds    0h
         EXEC CICS GETMAIN SET(R4) FLENGTH(DSA_SIZE) RESP(RETCD)
         clc    retcd,dfhresp(normal) check getmain error
         jne    getmain_abend
         using stack_hdr,r4
         st     r12,bootstg    work area for metal rock callback
         mvc    mtlrock_ep,callback_ep
         st     r4,blk_beg_addr
         lr     r5,r4         compute end address
         a      r5,dsa_size   *
         st     r5,blk_end_addr
*
         lh     r5,next_free  store block address in pool ctrl
         sll    r5,2          *
         la     r6,dsapool
         st     r4,0(r5,r6)   *
         srl    r5,2          increment array index
         la     r5,1(,r5)     *
         sth    r5,next_free  *
         drop   r4
*
         lr     r2,r4         r2 has blk hdr ptr
         lr     r1,r4         r1 contains the user section
         la     r1,lstack_hdr(,r1)
         j      retrock
*
```

Figure 52. CICS Bootstrap for metal C code example: MTLBOOT (Part 4 of 7)

```
getmain_abend  ds  0h
         EXEC  CICS ABEND ABCODE('EMTL')
*
retrock  ds    0h
         LM    r10,r0,mtlrsa
         BR    r14
*
*****************************************************************
* METALC WSA Initialization plugin                              *
* The R13, dfheistg being used here is not the same as that     *
* being used in the main entry and in mtlrock. This EISTG is    *
* taken out of the initial DSA allocated for the metal c main   *
* entry point                                                   *
*****************************************************************
         ENTRY MTLZWSAT
MTLZWSAT DS    0F
         STM   14,11,12(13)
         LR    15,13
         L     13,8(,13)
         ST    15,4(,13)
* init common registers
         lr    12,13               eistg for this callback
         LR    R9,R1                set plist register
         LARL  R10,CONSTANTS
         USING PLISTTRM,R9
*
* logic
*
         EXEC CICS ADDRESS EIB(DFHEIBR)
         ICM  R4,B'1111',trm_addr
         jz    trm_retn
         EXEC CICS FREEMAIN DATAPOINTER(R4)
*
trm_retn ds    0h
         L     13,4(,13)
         L     14,12(,13)
         LM    1,11,24(13)
         BR    14
         DS    0F
         DROP  R9
*
*****************************************************************
* METALC WSA Initialization plugin                              *
* The R13, dfheistg being used here is not the same as that     *
* being used in the main entry and in mtlrock. This EISTG is    *
* taken out of the initial DSA allocated for the metal c main   *
* entry point                                                   *
*****************************************************************
```

Figure 52. CICS Bootstrap for metal C code example: MTLBOOT (Part 5 of 7)

```
         ENTRY MTLZWSAI
MTLZWSAI DS    0F
         STM   14,11,12(13)
         LR    15,13
         L     13,8(,13)
         ST    15,4(,13)
*
* init common registers
*
         lr    12,13               eistg for this callback
         LARL  R10,CONSTANTS
         LR    R9,R1         set plist register
         USING PLISTINIT,R9
         XC    AREA,AREA      init output data
*
* logic
*
         EXEC CICS ADDRESS EIB(DFHEIBR)
         ICM   R5,B'1111',INIT_SIZE
         JZ    RETURN         nothing to do
         icm   r6,b'1111',init_addr is there   something to copy
         jz    return         nothing to copy
*
         EXEC CICS GETMAIN SET(R4) FLENGTH(INIT_SIZE) RESP(RETCD)
         clc  retcd,dfhresp(normal) check getmain error
         je   wsacopy         proceed to copy
************************************************************
* controlled abend the transaction if getmain fails to allow*
* IPCS analysis                                             *
************************************************************
*        DC    H'0'
         EXEC  CICS ABEND ABCODE('DMTL')
         j     return
*
wsacopy  ds    0h
         ST    R4,AREA
         lr    r7,r5          propagate length
         mvcl  r4,r6          and copy the WSA area
*
RETURN   DS    0H
         L     15,AREA
         L     13,4(,13)
         L     14,12(,13)
         LM    1,11,24(13)
         BR    14
         DS    0F
*
CONSTANTS    DS   0D
VMAIN        DC    V(MAIN)
callback_ep  dc    a(mtlrock)
```

Figure 52. CICS Bootstrap for metal C code example: MTLBOOT (Part 6 of 7)

```
*************************************************************
* define the allocation size of a stack block here          *
*************************************************************
dsa_size     dc    a(32000)
*dsa_size    dc    a(4096)
         LTORG
*************************************************************
* prolog/epilog for a CICS API layer (proposed)             *
*************************************************************
         ENTRY MTL2CICS
MTL2CICS DS    0F
         STM   RE,RC,12(RD)       save registers
         lr    rc,rd              pick up the 'this' pointer
         ahi   rc,-4              *
         l     rc,0(,rc)          point to stack block header
         l     rc,0(,rc)          and pick up bootstg ptr
         st    rd,bootrsa+4       store prev save pointer
         la    rd,bootrsa         point to routine RSA
*
*  mtl2cics logic
*
mtl2cics_rtn   ds   0h
         l     rd,4(,rd)          recover previous save pointer
         lm    re,rc,12(rd)      restore regs
         br    re                 and return to caller
*
         ltorg                    contants/literal pool
         END   MTLBOOT
```

Figure 52. CICS Bootstrap for metal C code example: MTLBOOT (Part 7 of 7)

Figure 53 on page 135 contains example code that demonstrates the use of the prolog and epilog examples.

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

/* char w_text[81];    */
/* the generated code seems to locate globals in the             */
/* code section. Global variables make the program non-reentrant */
DFHEIBLK *eiptr;
/* The prolog and epilog will cause a getmain and freemain for   */
/* additional stack space to be driven if the stack block size   */
/* setting in MTLBOOT is 4096 bytes.                             */
#pragma prolog(bigrtn,"MTLSENT")
#pragma epilog(bigrtn,"MTLSXIT")
void bigrtn()
{
  char bigbuff[3800];
  /*char bigbuff[16]; */
  bigbuff[0] = '1';
}

#pragma prolog(sendmsg,"MTLSENT")
/* #pragma epilog(sendmsg,"MTLSXIT") */
void sendmsg(char *s)
{
  int out_len = 81;
  char w_text[81];

  bigrtn();
  memset(w_text,' ',out_len);
  strncpy(w_text,s,out_len);
  EXEC CICS SEND FROM(w_text) LENGTH(out_len) WAIT;
}


#pragma prolog(main,"MTLENT")
#pragma epilog(main,"MTLXIT")
main()
{
  char msg[]= "METALC Hello";
  int   msg_len = 12;

  /*int out_len = 81; */
  /*char  w_text[81]; */
  /*char  s[23] = "Hello CICS from METAL!\n";*/

  EXEC CICS ADDRESS EIB(dfheiptr);
  EXEC CICS WRITEQ TS QUEUE("NOEL0000") FROM(msg) LENGTH(msg_len);
  sendmsg("Hello CICS from METAL!\n");
}
```

Figure 53. CICS API used under Metal C example code: MTLHALO

Figure 54 on page 136 contains the main prolog for using Metal C with CICS API programs.

```
*****************************************************************
*                                                               *
* MODULE NAME = MTLENT                                          *
*                                                               *
* DESCRIPTIVE NAME = METAL C FOR CICS MAIN PROLOG               *
*                                                               *
* @BANNER_START                          02                   *
*                       MTLENT                                  *
* LICENSED MATERIALS - PROPERTY OF IBM                          *
*                                                               *
* "RESTRICTED MATERIALS OF IBM"                                 *
*                                                               *
* 5655-S97                                                      *
*                                                               *
* (C) COPYRIGHT IBM CORP. 1989, 2009                            *
*                                                               *
*****************************************************************
          MACRO
&NAME     MTLENT
          GBLC  &CCN_PRCN
          GBLC  &CCN_LITN
          GBLC  &CCN_BEGIN
          GBLC  &CCN_ARCHLVL
          GBLA  &CCN_DSASZ
          GBLA  &CCN_RLOW
          GBLA  &CCN_RHIGH
          GBLB  &CCN_NAM
          GBLB  &CCN_LP64
          GBLC  &CCN_WSA_INIT
          GBLC  &CCN_WSA_TERM
&CCN_WSA_INIT SETC 'MTLZWSAI'
&CCN_WSA_TERM SETC 'MTLZWSAT'
*********************************************************************
* THE BOOTSTRAP ROUTINE WILL PROVIDE US WITH THE INITIAL EXECUTION    *
* ENVIROMENT. THE INPUT DATA, POINTED BY R1 IS DEFINED IN MTLBOOT.    *
*                                                                     *
* PREPARE THE EXECUTION ENVIRONMENT.                                  *
*********************************************************************
          STM   14,12,12(13)
*
*********************************************************************
* CHAIN THE CURRENT STACK TO THE PREVIOUS                           *
*********************************************************************
          LR    14,13           SAVE PREVIOUS SAVEAREA PTR
          L     13,0(,1)        PICK UP THE BLOCK HEADER POINTER
          L     12,4(,1)        PICK UP THE USER AREA POINTER
          ST    13,0(,12)       SAVE AS CURRENT STACK PREFIX
          LA    13,4(,12)       POINT R13 TO METALC USER STACK AREA
*
          ST    13,8(,14)       CHAIN DSA TO CALLER'S DSA
          ST    14,4(,13)       POINT TO PREVIOUS STACK AREA
          L     1,8(,1)         RESTORE PLIST POINTER
*
          MEND
```

Figure 54. Metal C for CICS main prolog: MTLENT

Figure 55 on page 137 contains the main epilog for using Metal C with CICS API programs.

```
*********************************************************************
*                                                                   *
* MODULE NAME = MTLXIT                                              *
*                                                                   *
* DESCRIPTIVE NAME = METAL C FOR CICS MAIN EPILOG                   *
*                                                                   *
* @BANNER_START                          02                       *
*                       MTLXIT                                      *
* LICENSED MATERIALS - PROPERTY OF IBM                              *
*                                                                   *
* "RESTRICTED MATERIALS OF IBM"                                     *
*                                                                   *
* 5655-S97                                                          *
*                                                                   *
* (C) COPYRIGHT IBM CORP. 1989, 2009                                *
*                                                                   *
*********************************************************************
          MACRO
&NAME     MTLXIT
          GBLC  &CCN_PRCN
          GBLC  &CCN_LITN
          GBLC  &CCN_BEGIN
          GBLC  &CCN_ARCHLVL
          GBLA  &CCN_DSASZ
          GBLA  &CCN_RLOW
          GBLA  &CCN_RHIGH
          GBLB  &CCN_NAM
          GBLB  &CCN_LP64
*********************************************************************
* RETURN TO THE BOOTSTRAP PROGRAM, MTLBOOT, WHICH WILL CLEAN UP THE *
* EXECUTION ENVIRONMENT.                                            *
*********************************************************************
          L     13,4(,13)         POINT TO CALLER'S SAVE
          LM    14,12,12(13)      RESTORE CALLER'S REGS
          BR    14                AND RETURN
          MEND
```

Figure 55. Metal C for CICS main epilog: MTLXIT

Figure 56 on page 138 contains the subroutine prolog for using Metal C with CICS API programs.

```
*****************************************************************
*                                                               *
* MODULE NAME = MTLSENT                                         *
*                                                               *
* DESCRIPTIVE NAME = METAL C FOR CICS SUBROUTINE PROLOG         *
*                                                               *
* @BANNER_START                          02                   *
*                       MTLSENT                                 *
* LICENSED MATERIALS - PROPERTY OF IBM                          *
*                                                               *
* "RESTRICTED MATERIALS OF IBM"                                 *
*                                                               *
* 5655-S97                                                      *
*                                                               *
* (C) COPYRIGHT IBM CORP. 1989, 2009                            *
*                                                               *
*****************************************************************
          MACRO
&NAME     MTLSENT
          GBLC  &CCN_PRCN
          GBLC  &CCN_LITN
          GBLC  &CCN_BEGIN
          GBLC  &CCN_ARCHLVL
          GBLA  &CCN_DSASZ
          GBLA  &CCN_RLOW
          GBLA  &CCN_RHIGH
          GBLB  &CCN_NAM
          GBLB  &CCN_LP64
*****************************************************************
* WARNING R0 HOLDS THE WSA TOKEN. DO NOT USE IT                 *
*****************************************************************
          STM   14,12,12(13)
          LR    15,13
          AHI   15,-4             POINT TO STACK PREFIX
          L     14,0(,15)         PICK UP BLOCK HDR PTR
*****************************************************************
* CHECK IF CURRENT BLOCK STILL HAS SPACE                        *
* THE MAXIMUM STACK ANY ROUTINE CAN HAVE SHOULD BE DSA_SIZE     *
* MINUS 20, THE SYSTEM USAGE                                    *
*****************************************************************
          L     2,8(,13)          PICK UP THE NAB
          AHI   2,4               COUNT THE PREFIX SIZE
          AHI   2,&CCN_DSASZ      CCN_DSASZ SHOULD FIT IN A
*                                 HALFWORD
*
          C     2,12(,14)         WILL IT FIT IN CURRNT BLK?
          JH    GET_SPACE_&SYSNDX  SORRY, NO.
          L     2,8(,13)          PICK UP NAB AGAIN
          ST    14,0(,2)          SET HEADER POINTER AS NEW
*                                 STACK PREFIX
          LA    2,4(,2)           POINT TO USER STACK AREA
          ST    13,4(,2)          CHAIN TO PREVIOUS STACK
          LR    13,2              SET R13 TO CURRENT
          J     DONE_&SYSNDX
*
GET_SPACE_&SYSNDX DS 0H
```

Figure 56. Metal C for CICS subroutine prolog: MTLSENT (Part 1 of 2)

```
*************************************************************
* OBTAIN ADDITIONAL STACK STORAGE                           *
*************************************************************
         LHI   2,1              SET GETMAIN FUNCTION
         L     1,0(,14)         SET THE BOOT TOKEN
         L     15,4(,14)        SET THE MTLROCK SERVICE ADDRS
         BASR  14,15            AND CALL IT
*
*        DC    H'0'
         ST    2,0(,1)          SET STACK PREFIX WORD
         L     2,24(,13)        RECOVER PLIST REGISTER
         ST    1,8(,13)         UPDATE PREVIOUS NAB WHICH
*                               POINTS TO THE STACK PREFIX
         LA    1,4(,1)          POINT TO USER STACK AREA
         ST    13,4(,1)         CHAIN IT TO PREVIOUS
         LR    13,1             SET R13 TO CURRENT
         LR    1,2              SET PLIST POINTER
*
DONE_&SYSNDX   DS  0H
         MEND
```

Figure 56. Metal C for CICS subroutine prolog: MTLSENT (Part 2 of 2)

Figure 57 on page 140 contains the subroutine epilog for using Metal C with CICS API programs.

```
*****************************************************************
*                                                               *
* MODULE NAME = MTLSXIT                                         *
*                                                               *
* DESCRIPTIVE NAME = METAL C FOR CICS SUBROUTINE EPILOG         *
*                                                               *
* @BANNER_START                          02                     *
*                       MTLSXIT                                 *
* LICENSED MATERIALS - PROPERTY OF IBM                          *
*                                                               *
* "RESTRICTED MATERIALS OF IBM"                                 *
*                                                               *
* 5655-S97                                                      *
*                                                               *
* (C) COPYRIGHT IBM CORP. 1989, 2009                            *
*                                                               *
*****************************************************************
          MACRO
&NAME     MTLSXIT
          GBLC  &CCN_PRCN
          GBLC  &CCN_LITN
          GBLC  &CCN_BEGIN
          GBLC  &CCN_ARCHLVL
          GBLA  &CCN_DSASZ
          GBLA  &CCN_RLOW
          GBLA  &CCN_RHIGH
          GBLB  &CCN_NAM
          GBLB  &CCN_LP64
*
          LR    15,13
          AHI   15,-4             POINT TO PREFIX
          LR    2,15              COMPUTE THEORETICAL BLOCK CTL
          AHI   2,-16             AREA POINTER
          C     2,0(,15)          ARE WE AT THE TOP OF BLOCK ?
          JNE   NOFREE_&SYSNDX
*****************************************************************
* WE HAVE REACHED THE TOP OF THE BLOCK WHICH WE ARE GOING TO  *
* HAND BACK AS WE ARE DONE WITH IT.                           *
* THE CONTENTS OF R12 WILL BE DESTROYED IN THE PROCESS        *
*****************************************************************
          LR    14,2              COPY BLOCK HDR POINTER
          LHI   2,2               REQUEST FREEMAIN
          L     1,0(,14)          SET THE BOOT TOKEN
          L     15,4(,14)         SET EP ADDRESS
          BASR  14,15             INVOKE SERVICE
*
NOFREE_&SYSNDX  DS   0H
          L     13,4(,13)         POINT TO CALLER'S SAVE
          LM    14,12,12(13)
          BR    14
          MEND
```

Figure 57. Metal C for CICS subroutine epilog: MTLSXIT

CICS exit programming interface example

The CICS exit programming interface is described in the CICS Customization Guide. It is used for programs which run at global user exit points. The example in this section is for the XTSEREQ exit point. The example exit is driven when a temporary storage queue request is to be serviced by CICS. The example exit code however does not perform any business logic. All that is shown is a storage management getmain and freemain request using the exit programming interface.

The CICS exit programming interface example consists of the following components:

MTLBTXPI Assembler Language bootstrap program.

MTL2XPI Example using the CICS exit programming interface for Metal C.

Figure 58 illustrates how the XPI components and entry points used in the examples are related.

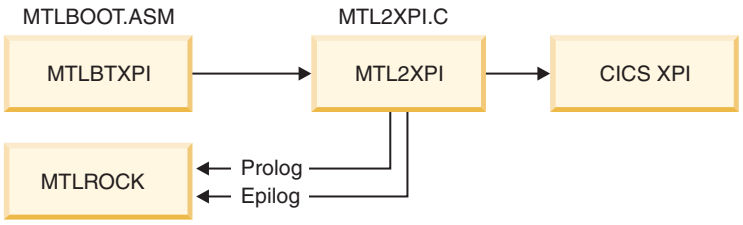


Figure 58. CICS XPI example flow

The prolog and epilog macros provided for the CICS API examples are also being used by MTL2XPI.

The MTLBTXPI program is similar to MTLBOOT. It performs the same function and sets up a similar execution environment for MTL2XPI. MTLBTXPI, unlike MTLBOOT, does not provide WSA initialization and termination. MTL2XPI is coded as a C subroutine but not a main program. This avoids the need of setting up a writable static area.

The CICS exit programming interface does not internally generate statically writable variables because it is an assembler only interface.

The data passed from MTLBTXPI to MTL2XPI is described by “main_plist” and the getmain allocated storage for use as the Metal C execution stack. In XPI execution, R1 points to the user exit parameter area. Sample code on how to properly pass this information for Metal C execution is shown. Like MTLBTXPI, the main_plist fields, hdr_ptr, and stack_ptr all point to areas in the Metal C stack storage.

MTL2XPI is the C code that shows an example of how to invoke a CICS XPI service in C. The services invoked are getmain and freemain, but this provides a demonstration that the CICS XPI which is assembler-only can be used by C code.

It should be pointed out that register 13 needs to be set to the contents of “uepstack” upon entry into the XPI. The contents must be copied into another register and not in the C execution stack storage, because register 13 is also used by Metal C as the execution stack pointer.

MTL2XPI uses MTLENT and MTLXIT as its prolog and epilog macros. This demonstrates reuse of these macros because the bootstrap programs, while different, provide the same interface data. MTLSENT and MTLSXIT can also be used in XPI using Metal C code if stack expansion is required.

Example code

Figure 59 on page 142 contains example code for the Metal C CICS bootstrap.

```
*ASM     XOPTS(NOPROLOG NOEPILOG SP)
*****************************************************************
*                                                               *
* Module Name = MTLBTXPI                                        *
*                                                               *
* Descriptive Name = CICS Bootstrap for metal C code example    *
*                                                               *
* @BANNER_START                           02                   *
*                       MTLBTXPI                                *
* Licensed Materials - Property of IBM                          *
*                                                               *
* "Restricted Materials of IBM"                                 *
*                                                               *
* 5655-S97                                                      *
*                                                               *
* (C) Copyright IBM Corp. 1989, 2009                            *
*                                                               *
* Description                                                   *
*   The bootstrap routine sets up stack storage and flows       *
*   control to the metal c program. The metalc program will     *
*   implement the business logic as a C subrooutine ie not      *
*   'main'. This means a WSA Init/Term callbacks don't need to  *
*   be provided.                                                *
*                                                               *
*   Like the metalc CICS application bootstrap, storage         *
*   management callbacks for the various subroutine             *
*   prolog/epilog code is still provided.                       *
*                                                               *
*   This bootstrap code is used for coding CICS Exit Programming*
*   Interface using programs. The XPI is used for assembler-only*
*   user exit programs.                                         *
*                                                               *
*   The z/OS C/C++ metalc option allows C programmers to use    *
*   the C language to write CICS XPI code in user exit programs.*
*                                                               *
*****************************************************************
* The Metal C Bootstrap program for the CICS XPI                *
*****************************************************************
*
*****************************************************************
* Define the stack block control area                           *
*****************************************************************
stack_hdr     dsect
bootstg       ds   f       bootstrap storage pointer
mtlrock_ep    ds   a       callback for getmain/freemain
blk_beg_addr  ds   a       begin address of block
blk_end_addr  ds   a       end address of block
lstack_hdr    equ  *-stack_hdr
*
metlget       equ  x'0001'
metlfre       equ  x'0002'
*
              copy dfhkebrc
```

Figure 59. CICS bootstrap for Metal C example program: MTLBTXPI (Part 1 of 7)

```
*****************************************************************
* Program state                                                 *
*****************************************************************
*
*  local working storage, pointed to by bootstg
*
workstg       dsect
bootrsa  DS     18F
booteye  ds     cl8
mtlrsa   ds     7F
saverd   DS     F
*
main_plist  ds  0a         metalc main entry argument
hdr_ptr     ds  a          ptr to dsa stack header
stack_ptr   ds  a          ptr to dsa stack user area
org_r1_ptr  ds  a          r1 contents on entry
*
org_r1      ds  a          r1 contents on entry
dsapool  ds     10f        array of pointers to dsablocks
next_free ds    H          array index
lworkstg    equ *-workstg
*****************************************************************
* The XTSEREQ exit is driven before CICS processes a temporary  *
* storage API request. To use this example for other global user*
* exit points, include the relevant definition of the user exit *
* parameter list.                                               *
*****************************************************************
         DFHUEXIT TYPE=EP,ID=XTSEREQ GLUE parameter list definition
         DFHUEXIT TYPE=XPIENV      Setup XPI environment
         COPY DFHSMMCY             Setup XPI plist for SM
*
MTLBTXPI CSECT
MTLBTXPI AMODE  31
MTLBTXPI RMODE  31
*
* logic
* getmain 'main' working storage
* format R1 content
* dispatch
*
         stm   R14,R12,12(R13)     Save caller's registers
* Get access to GLUE parameter list
         lr    R9,R1               Save r1 contents
*
         larl  R10,CONSTANTS
         using CONSTANTS,R10
*
*  get bootstrap work area storage
*
         l     r2,work_size
         lr    r3,r1               set dfhuepar ptr register
         brasl r11,getstor
         lr    r4,r1
         using workstg,r4          set addr'ability
         mvc   booteye,=cl8'>MTLBXPI'  set eyecatcher in anchor
         st    r9,org_r1      save user plist register
         la    r9,org_r1      normalize to C parm convention
         st    r9,org_r1_ptr  *
```

Figure 59. CICS bootstrap for Metal C example program: MTLBTXPI (Part 2 of 7)

```
*
*   get initial dynamic storage area
*
         l      r2,dsa_size
         l      r3,org_r1          set dfhuepar ptr register
         brasl  r11,getstor
         lr     r5,r1
         using stack_hdr,r5
         st     r4,bootstg    work area for metal rock callback
         mvc    mtlrock_ep,callback_ep
         st     r5,blk_beg_addr
         lr     r6,r5         compute end address
         a      r6,dsa_size   *
         st     r6,blk_end_addr
*
         xr     r6,r6         store block address in pool ctrl
         la     r7,dsapool
         st     r5,0(r6,r7)   *
         la     r6,1(,r6)     increment index
         sth    r6,next_free  *
         drop   r5
*
         la     r6,lstack_hdr(,r5)  point to user area
         st     r5,hdr_ptr
         st     r6,stack_ptr
         la     r1,main_plist set plist register
*
* flow control to the metal c program
*
         la     R13,bootrsa   point R13 to bootstrap save
         l      R15,metal_ep
         basr   R14,R15
*
* free the allocated metalc stack storage
*
         lh    r6,next_free  obtain block address to free
         bctr  r6,0          some validation needed here
         sll   r6,2          *
         la    r7,dsapool    *
         l     r2,0(r6,r7)   set free storage parameter
         l     r3,org_r1           pick up dfhuepar ptr
         brasl r11,freestor
*
main_return ds  0h
         l     r3,org_r1      load dfhuepar ptr
         lr    r2,r4          address to free
         brasl  r11,freestor  and free up our work area
         l     r13,UEPEPSA-DFHUEPAR(,r3) point to caller's save
         lm    r14,r12,12(r13)     restore caller's register
         br    r14                 return to caller
*
```

Figure 59. CICS bootstrap for Metal C example program: MTLBTXPI (Part 3 of 7)

```
*****************************************************************
* The MTLROCK callback service                                  *
* The input environment is as follows;                          *
*  R2 - the function code                                       *
*  R1 - the boot token, the mtlbtxpi workstorage ptr. This is   *
*       important for getmain especially, because at the time   *
*       getmain for stack storage is invoked, there's no more   *
*       storage space to use.                                   *
*       C++ developers can think of this as the 'this' pointer  *
* On return from a getmain                                      *
*  R2 - block pointer                                           *
*  R1 - the user section                                        *
*                                                               *
* The getmained block pointer is stored in an array of 10       *
* elements. No check is make for overflow.                      *
*****************************************************************
         ENTRY MTLROCK
MTLROCK  DS    0H
         stm   r10,r0,mtlrsa-workstg(r1) save used regs
         lr    r4,r1            set token in r4
*
LETSROCK DS    0H
         larl  r10,constants
*
         chi   r2,metlget            do we getmain
         je    getmain
*****************************************************************
* else it's a freemain. Since we are maintaining a stack,      *
* the last getmained block is the first to be freed            *
*****************************************************************
*
* freemain logic
         lh    r5,next_free  obtain block address to free
         bctr  r5,0          some validation needed here
         sll   r5,2          *
         la    r6,dsapool    *
         l     r2,0(r5,r6)   set free storage parameter
         l     r3,org_r1           pick up dfhuepar ptr
         brasl r11,freestor
*
         xr    r3,r3         clear the array location
         st    r3,0(r5,r6)   *
         srl   r5,2          decrement the array index
         sth   r5,next_free  *
         j     retrock
*
```

Figure 59. CICS bootstrap for Metal C example program: MTLBTXPI (Part 4 of 7)

```
*************************************************************
* getmain goes in here                                      *
*************************************************************
getmain  ds    0h
         l      r2,dsa_size        set size to get
         l      r3,org_r1          set dfhuepar ptr register
         brasl  r11,getstor
         lr     r5,r1
         using stack_hdr,r5
         st     r4,bootstg    work area for metal rock callback
         mvc    mtlrock_ep,callback_ep
         st     r5,blk_beg_addr
         lr     r6,r5         compute end address
         a      r6,dsa_size   *
         st     r6,blk_end_addr
*
         lh     r6,next_free  store block address in pool ctrl
         sll    r6,2          *
         la     r7,dsapool
         st     r5,0(r6,r7)   *
         srl    r6,2          increment array index
         la     r6,1(,r6)     *
         sth    r6,next_free  *
         drop   r5
*
         lr     r2,r5         r2 has blk hdr ptr
         lr     r1,r5         r1 contains the user section
         la     r1,lstack_hdr(,r1)
         j      retrock
*
retrock  ds    0h
         lm    r10,r0,mtlrsa
         br    r14
*
*
CONSTANTS    ds   0D
metal_ep     dc    V(mtl2xpi)
callback_ep  dc    a(mtlrock)
*************************************************************
* define the allocation size of a stack block here. Change  *
* these as needed.                                          *
*************************************************************
dsa_size     dc    a(32000)
work_size    dc    a(lworkstg)
*dsa_size    dc    a(4096)
         LTORG
```

Figure 59. CICS bootstrap for Metal C example program: MTLBTXPI (Part 5 of 7)

```
*************************************************************
* Get dynamic storage area.                                 *
* The CICS storage management XPI function, DHFSMMCX, is used *
* to get the storage.                                       *
* R2 contains the storage length                            *
* R1 contains the storage address on exit or zero if an error *
* occurred.                                                 *
* Work regs;                                                *
* R3,R6,R7                                                  *
*************************************************************
getstor  ds    0h
         using dfhuepar,r3        set by caller
         l     R6,UEPXSTOR          set base to XPI plist
         using DFHSMMC_ARG,R6      .. tell ASM
         st    r13,saverd          save R13
         l     R13,UEPSTACK        Set R13 to the Kernel stack
         DFHSMMCX CALL,                                                @
               CLEAR,                                                  @
               IN,                                                     @
               FUNCTION(GETMAIN),                                      @
               GET_LENGTH((R2)),                                       @
               SUSPEND(YES),                                           @
               STORAGE_CLASS(USER),                                    @
               OUT,                                                    @
               ADDRESS((R7)),                                          @
               RESPONSE(*),                                            @
               REASON(*)
         CLI   SMMC_RESPONSE,SMMC_OK Response OK?
         BE    GETOK               .. yes
*
         xr    r7,r7               clear output register
         WTO   'MTLBTXPI - GETMAIN failure '
         LA    R15,UERCBYP         Get return code
         L     R1,UEPEPSA          Get address of caller's RSA
         ST    R15,16(R1)          Store RC in caller's R15
*
GETOK    DS    0H                  R2 is base register for program data
         L     R13,saverd          restore r13 contents
         lr    r1,r7               set output reg
         br    r11                 and return to caller
         drop  r3,r6
```

Figure 59. CICS bootstrap for Metal C example program: MTLBTXPI (Part 6 of 7)

```
************************************************************
* Free allocated storage                                   *
* R2 contains the address of the storage to be freed       *
* Work Regs;                                               *
* R3,R6                                                    *
************************************************************
freestor DS    0H
         using dfhuepar,r3         r3 set by caller
         L     R6,UEPXSTOR          set base to XPI plist
         USING DFHSMMC_ARG,R6      .. tell ASM
         st    r13,saverd
         L     R13,UEPSTACK        Set R13 to the Kernel stack
         DFHSMMCX CALL,                                                @
               CLEAR,                                                  @
               IN,                                                     @
               FUNCTION(FREEMAIN),                                     @
               ADDRESS((R2)),                                          @
               STORAGE_CLASS(USER),                                    @
               OUT,                                                    @
               RESPONSE(*),                                            @
               REASON(*)
*
* return code checking is left as an exercise for the reader
*
         l     r13,saverd
         br    r11
         drop  r3,r6
         LTORG
         END   MTLBTXPI
```

Figure 59. CICS bootstrap for Metal C example program: MTLBTXPI (Part 7 of 7)

Figure 60 on page 149 contains example code to use the CICS exit programming API in C.

```
/*****************************************************************/
/*                                                               */
/* Program Name : MTL2XPI                                        */
/* Description  : Sample code to use the CICS Exit Programming API*/
/*                in C                                           */
/* Author       : Noel C. Sales                                  */
/* Date         : 21 Jan 2010                                    */
/*                                                               */
/*****************************************************************/


/*---------------------------------------------------------------*/
/* a C mapping for the dfhuepar dsect                            */
/*---------------------------------------------------------------*/
typedef struct {
  void  *uepexn;      /* Address of exit number                  */
  void  *uepgaa;      /* Address of global work area             */
  void  *uepgal;      /* Address of work area length             */
  void  *uepcrca;     /* Address of current return code          */
  void  *ueptca;      /* reserved                                */
  void  *uepcsa;      /* reserved                                */
  void  *uepepsa;     /* Address of exit prog save area          */
  void  *uephmsa;     /* Address of host module"s RSA            */
  void  *uepgind;     /* Address of task data key and data       */
                      /* location flags                          */
  void  *uepstack;    /* Address of kernel stack entry           */
  void  *uepxstor;    /* Address of storage for XPI              */
                      /* parameter list                          */
       /* standard parameters above completed by User Exit Handler*/
  void  *ueptrace;    /* Address of Trace flag                   */
  void  *uepparms;    /* Start of variable parameters            */
  void  *ueppcds;     /* Address of program control exits        */
                      /*       DSECT                             */
  void  *ueptacb;     /* Address of TACB                         */
} dfhuepar_t;

/*
  ********************************************************************
   Notes:
     Metal C uses register 13 to point to the DSA by default. However,
     we are entered with register 13 pointing to the LIFO stack. To
     resolve this the bootstrap program should tuck register 13 in a
     safe place in the DSA it allocates.

     Before invoking the XPI API, we save the current register 13 in a
     variable, replace it with the register 13 tucked away in the
     'safe place' then restore R13 when we're through.

     To ensure that we still have addressability to the variable,
     We have the option to use WSA or a register.
     We cannot use the local stack storage because
     register 13 points to the local stack and we just overlayed the
     register.
 ********************************************************************
*/

typedef struct
{
   void       *stack_hdr_p;
   void       *stack_user_area_p;
   dfhuepar_t *ueparm_p;
} mtl_parm_t;
```

Figure 60. CICS exit programming API example program: MTL2XPI (Part 1 of 3)

```
static void get_storage(dfhuepar_t *plist,void **storage)
{
  void       *pstg;
  short      getlen;
  void       *sm_arg;
  void       *kern_stack;
  register   saverd;
  register   rx;

  getlen = 128;
  sm_arg = plist->uepxstor;
  kern_stack = plist->uepstack;

************************************************************
* Call the XPI function.                                   *
* The syntax for DFHSMMCX GETMAIN is                       *
* DFHSMMCX [CALL,]                                         *
*          [CLEAR,]                                        *
*          [IN,                                            *
*          FUNCTION(GETMAIN),                              *
*          GET_LENGTH(name4 | (Rn) | expression),          *
*          STORAGE_CLASS(CICS|CICS24|SHARED_CICS|          *
*             SHARED_CICS24|SHARED_USER|SHARED_USER24|USER|*
*             USER24|TERMINAL),                            *
*          SUSPEND(NO|YES),                                *
*          [INITIAL_IMAGE(name1 | literalconst),]          *
*          [TCTTE_ADDRESS(name4 | (Ra)),]]                 *
*          [OUT,                                           *
*          ADDRESS(name4 | (Rn) | *),                      *
*          RESPONSE(name1 | *),                            *
*          REASON(name1 | *)]                              *
* In this example, we use the GET_LENGTH, STORAGE_CLASS and *
* SUSPEND input parameters, and output the ADDRESS, and the *
* response and reason.                                     *
************************************************************
  __asm("  L %0,%4 \n"
        "  LR %2,13\n"
        "  L  13,%5\n"
        "  USING DFHSMMC_ARG,%0 \n"
        "  DFHSMMCX CALL,CLEAR,"
        "IN,FUNCTION(GETMAIN),"
          "GET_LENGTH(%3),"
          "SUSPEND(YES),"
          "STORAGE_CLASS(USER),"
        "OUT,"
          "ADDRESS(%1),"
          "RESPONSE(*),"
          "REASON(*)\n"
        " DROP %0\n"
        " LR 13,%2"
         : "=r"(rx),"=m"(pstg),"=r"(saverd)
         : "m"(getlen),"m"(sm_arg),"m"(kern_stack)
       );

  *storage = pstg;
}
```

Figure 60. CICS exit programming API example program: MTL2XPI (Part 2 of 3)

```
static void business_logic(dfhuepar_t *plist, void * storage)
{
}

static void free_storage(dfhuepar_t *plist, void *storage)
{

    register rx;
    register saverd;
    void     *sm_arg;
    void     *kern_stack;
    void     *uepstack;
    void     *address;

    sm_arg = plist->uepxstor;
    kern_stack = plist->uepstack;
    address = storage;

 __asm("  L   %0,%3\n"
          "  LR  %1,13\n"
          "  L   13,%4\n"
          "  USING DFHSMMC_ARG,%0\n"
          "  DFHSMMCX CALL,CLEAR,"
          "IN,"
            "FUNCTION(FREEMAIN),"
            "ADDRESS(%2),"
            "STORAGE_CLASS(USER),"
          "OUT,"
            "RESPONSE(*),"
            "REASON(*)\n"
          " LR 13,%1\n"
          " DROP %0"
           : "=r"(rx),"=r"(saverd)
           : "m"(address),"m"(sm_arg),"m"(kern_stack)
       );
}

#pragma prolog(mtl2xpi,"MTLENT")
#pragma epilog(mtl2xpi,"MTLXIT")

void mtl2xpi(dfhuepar_t *plist)
{

  __asm(" DFHUEXIT TYPE=XPIENV\n"
        " COPY DFHSMMCY\n"
        "&CCN_CSECT CSECT");

  void *storage;
  get_storage(plist,&storage);
  business_logic(plist,storage);
  free_storage(plist,storage);

}
```

Figure 60. CICS exit programming API example program: MTL2XPI (Part 3 of 3)

CICS definitions

The CICS API example program is, for all intents and purposes, an assembler program. It requires the normal CICS definitions, using for example CEDA or the CICS Explorer, to define it as a program and the definition to map the program to a CICS transaction.

The exit is enabled using the **ENABLE PROGRAM** command, for example via the CECI transaction:

```
CECI ENABLE PROGRAM(MTLBTXPI) EXIT('xtsereq') start
```

The exit is triggered each time a temporary storage request is to be serviced by CICS. The following command issued using the CECI transaction is an example of a request to temporary storage.

```
CECI WRITEQ TS QU('NOELNOEL') FROM('HELLO')
```

Figure 61 describes the CEDA definition for the API example program.

```
CEDA View PROGram (METALH)
  PROGram        : METALH
  Group          : NCSMETAL
  DEScription    : FIRST METAL PROGRAM
  Language       : Assembler          CObol | Assembler | Le370 | C | Pli
  RELoad         : No                 No | Yes
  RESident       : No                 No | Yes
  USAge          : Normal             Normal | Transient
  USElpacopy     : No                 No | Yes
  Status         : Enabled            Enabled | Disabled
  RSl            : 00                 0-24 | Public
  CEdf           : Yes                Yes | No
  DAtalocation   : Any                Below | Any
  EXECKey        : User               User | Cics
  COncurrency    : Quasirent          Quasirent | Threadsafe
  Api            : Cicsapi            Cicsapi | Openapi
```

Figure 61. CICS CEDA definition for the API example program

Figure 62 describes the CICS transaction definition.

Figure 62. CICS transaction definition

```
CEDA View TRANSaction (MET0)
  TRANSaction    : MET0
  Group          : NCSMETAL
  DEScription    :
  PROGram        : METALH
  TWasize        : 00000              0-32767
  PROFile        : DFHCICST
  PArtitionset   :
  STAtus         : Enabled            Enabled | Disabled
  PRIMedsize     : 00000              0-65520
  TASKDATALoc    : Any                Below | Any
  TASKDATAKey    : User               User | Cics
  STOrageclear   : No                 No | Yes
  RUnaway        : System             System | 0 | 500-2700000
  SHutdown       : Disabled           Disabled | Enabled
  ISolate        : Yes                Yes | No
```

Figure 63 on page 153 describes the CICS XPI example as defined in the CEDA.

```
CEDA View PROGram (MTLBTXPI)
  PROGram        : MTLBTXPI
  Group          : NCSMETAL
  DEScription    :
  Language       : Assembler          CObol | Assembler | Le370 | C | Pli
  RELoad         : No                 No | Yes
  RESident       : No                 No | Yes
  USAge          : Normal             Normal | Transient
  USElpacopy     : No                 No | Yes
  Status         : Enabled            Enabled | Disabled
  RSl            : 00                 0-24 | Public
  CEdf           : Yes                Yes | No
  DAtalocation   : Any                Below | Any
  EXECKey        : Cics               User | Cics
  COncurrency    : Quasirent          Quasirent | Threadsafe
  Api            : Cicsapi            Cicsapi | Openapi
```

Figure 63. Defining the CICS XPI example in the CEDA

JCL example

The following example JCL shows you how to build the example code. You need to provide appropriate libraries in place of those shown in the example JCL, such as MTLUSR.XPLINK.LOAD and MTLUSR.METAL.OBJ.

```
//MTLUSR00 JOB (999,POK),'METAL',CLASS=A,MSGCLASS=H,NOTIFY=&SYSUID
//*
//* BINDER USING THE METAL XPI SAMPLE PROGRAM
//* //LKED     EXEC  PGM=IEWL,REGION=256K,
//    PARM='LIST,LET,XREF,MAP,AC(0),RENT,REUS,AMODE(31)'
//SYSPRINT DD  SYSOUT=*
//SYSUT1   DD  SPACE=(CYL,(10,10)),UNIT=SYSDA
//SYSLMOD  DD  DSN=MTLUSR.XPLINK.LOAD,DISP=SHR
//SYSLIB   DD  DSN=CICSTS41.CICS.SDFHLOAD,DISP=SHR
//         DD  DISP=SHR,DSN=MTLUSR.METAL.OBJ
//         DD  DISP=SHR,DSN=MTLUSR.METALC.SCCNOBJ
//USROBJ   DD  DSN=MTLUSR.METAL.OBJ,DISP=SHR
//SYSLIN    DD *
INCLUDE USROBJ(MTLBTXPI)
INCLUDE USROBJ(MTL2XPI)
ENTRY MTLBTXPI
NAME  MTLBTXPI(R)
/*
```

Figure 64. CICS LNKXPI JCL example

```
//MTLUSR0  JOB (999,POK),'CICSASM',CLASS=A,MSGCLASS=H,NOTIFY=&SYSUID
//XPIASM   PROC DSN=,MEM=
//*****************************************************************
//* run ASM
//*****************************************************************
//STEPASM EXEC PGM=ASMA90,PARM=OBJECT,REGION=0M
//SYSUT1  DD   UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSPRINT DD  SYSOUT=*
//SYSIN    DD DSN=&DSN(&MEM),DISP=SHR
//SYSLIN  DD  DISP=SHR,DSN=MTLUSR.METAL.OBJ(&MEM)
//SYSLIB DD DISP=SHR,DSN=SYS1.MACLIB
//           DD DISP=SHR,DSN=CICSTS41.CICS.SDFHMAC
//           DD DISP=SHR,DSN=CEE.SCEEMAC
// PEND
//*****************************************************************
//* START OF COMPILES:
//*****************************************************************
//COMP     EXEC XPIASM,DSN='DEV.METALC.SAMPCODE',MEM='MTLBTXPI'
```

Figure 65. CICS ASMXPI JCL example

```
//MTLUSRPC JOB (999,POK),'CCOMP',NOTIFY=&SYSUID,
//   CLASS=A,MSGCLASS=H
//*-------------------------------------------------------------
//* CICS Metal JCL
//*
//*******************************************************************
//* Compile the code
//*******************************************************************
//CCAM     PROC IDSN=,ADSN=,ODSN=,MEM=
//CC       EXEC PGM=CCNDRVR,REGION=0M,
// PARM=('OPTFILE(DD:OPTIONS)')
//STEPLIB  DD DISP=SHR,DSN=MTLCICS.METALC.SCCNCMP
//         DD DSN=CICSTS41.CICS.SDFHLOAD,DISP=SHR
//         DD DISP=SHR,DSN=CEE.SCEERUN
//         DD DISP=SHR,DSN=CEE.SCEERUN2
//OPTIONS  DD DISP=SHR,DSN=MTLCICS.METALC.SAMPJCL(OPTXPI)
//SYSLIB   DD PATH='/usr/include/metal',PATHOPTS=ORDONLY
//         DD DSN=CICSTS41.CICS.SDFHC370,DISP=SHR
//         DD DSN=CICSTS41.CICS.SDFHMAC,DISP=SHR
//         DD DSN=MTLCICS.METALC.SAMPMAC,DISP=SHR
//************************************************
//SYSUT1   DD  UNIT=SYSDA,SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT4   DD  UNIT=SYSDA,SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//SYSUT5   DD  UNIT=SYSDA,SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT6   DD  UNIT=SYSDA,SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT7   DD  UNIT=SYSDA,SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT8   DD  UNIT=SYSDA,SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT9   DD  UNIT=SYSDA,SPACE=(32000,(30,30)),
//             DCB=(RECFM=VB,LRECL=137,BLKSIZE=882)
//SYSUT10  DD  SYSOUT=*
//SYSUT14  DD  UNIT=SYSDA,SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=3200,BLKSIZE=12800)
//SYSUT15  DD  SYSOUT=*
//************************************************
//SYSPRINT DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//SYSCPRT  DD SYSOUT=*
//*SYSLIN   DD DSN=&&SYSLIN,DISP=(NEW,PASS),SPACE=(TRK,(10,100)),
//*         UNIT=SYSDA,BLKSIZE=3200,LRECL=80,RECFM=FB,DSORG=PS
//SYSLIN   DD DISP=SHR,DSN=&ADSN(&MEM)
//SYSIN    DD DISP=SHR,DSN=&IDSN(&MEM)
```

Figure 66. CICS CCXPI JCL example (Part 1 of 2)

```
//*******************************************************************
//* Assemble the code
//*******************************************************************
//ASM     EXEC PGM=ASMA90,REGION=0M,PARM='GOFF'
//SYSLIB   DD  DSN=SYS1.MACLIB,DISP=SHR
//         DD DISP=SHR,DSN=CICSTS41.CICS.SDFHMAC
//         DD DISP=SHR,DSN=CEE.SCEEMAC
//         DD DSN=MTLCICS.METALC.SAMPMAC,DISP=SHR
//SYSUT1   DD  UNIT=(SYSDA,SEP=SYSLIB),SPACE=(CYL,(10,5)),DSN=&SYSUT1
//SYSPRINT DD  SYSOUT=*
//SYSLIN   DD  DISP=SHR,DSN=&ODSN(&MEM)
//SYSIN    DD  DISP=SHR,DSN=&ADSN(&MEM)
// PEND
//*
//COMP     EXEC CCAM,IDSN='MTLCICS.METALC.SAMPCODE',
//         ADSN='MTLUSR.METAL.GENASM',
//         ODSN='MTLUSR.METAL.OBJ',MEM=MTL2XPI
//*
```

Figure 66. CICS CCXPI JCL example (Part 2 of 2)

```
METAL GENASM
OPT(0) PHASEID LANGLVL(EXTENDED)
SO LIST CSECT
float(ieee)
DEF(MVS,CM_MVS,_TCP31_PROTOS)
nose se(/usr/include/metal, DD:SYSLIB)
SSCOM
AGG
RENT
```

Figure 67. CICS OPTXPI JCL example

Appendix C. Accessibility

Publications for this product are offered in Adobe Portable Document Format (PDF) and should be compliant with accessibility standards. If you experience difficulties when using PDF files, you may view the information through the z/OS Internet Library website or the z/OS Information Center. If you continue to experience problems, send an email to mhvrcfs@us.ibm.com or write to:

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
USA

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 or Library Server versions of z/OS books in the Internet library at:

`http://www.ibm.com/systems/z/os/zos/bkserv/`

One exception is command syntax that is published in railroad track format, which is accessible using screen readers with the Information Center, as described in “Dotted decimal syntax diagrams.”

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users accessing the Information Center using a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, you know that your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol giving information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, this indicates a reference that is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you should refer to separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

- ? means an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.
- ! means a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicate that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword.

In this example, if you include the FILE keyword but do not specify an option, default option KEEP will be applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

- * means a syntax element that can be repeated 0 or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one data area, more than one data area, or no data area. If you hear the lines 3*, 3 HOST, and 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

  Notes:

  1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
  2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you could write HOST STATE, but you could not write HOST HOST.
  3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.

- + means a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times; that is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can only repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

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 information documents intended programming interfaces that allow the customer to write programs to obtain the services of z/OS Metal C runtime library.

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/.

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 If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at &sdq;Copyright and trademark information&sdq; at www.ibm.com/legal/copytrade.shtml.

Adobe, Acrobat, and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

IEEE is a trademark of the Institute of Electrical and Electronics Engineers, Inc. in the United States and other countries.

Other company, product, and service names might be trademarks or service marks of others.

Index

Special characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

T

U

V

X

Z

Product Number: 5694-A01

Printed in USA

SA23-2225-05