z/OS

IBM

XL C/C++ User's Guide

SC09-4767-08

z/OS

XL C/C++ User's Guide

SC09-4767-08

Note!

Before using this information and the product it supports, be sure to read the information in “Notices” on page 677.

Ninth Edition (September 2009)

This edition applies to IBM z/OS XL C/C++ compiler in IBM z/OS operating system Version 1 Release 11 (5694-A01) and to all subsequent releases until otherwise indicated in new editions. This edition replaces SC09-4767-07. Make sure that you use the correct edition for the level of the program listed above. Also, ensure that you apply all necessary PTFs for the program.

IBM welcomes your comments. You can send your comments to the following Internet address: compinfo@ca.ibm.com. Be sure to include your e-mail address if you want a reply.

Include the title and order number of this book, and the page number or topic related to your comment. When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1998, 2009.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

© Copyright IBM Corp. 1998, 2009

About this document

This edition of *z/OS XL C/C++ User's Guide* is intended for users of the IBM® z/OS® XL C/C++ compiler with the IBM Language Environment® element provided with z/OS. It provides you with information about implementing (compiling, linking, and running) programs that are written in C and C++. It contains guidelines for preparing C and C++ programs to run on the z/OS operating system.

Systems programmers who are interested in customizing compiler option defaults, should review Appendix F, "Customizing default options for z/OS XL C/C++ compiler," on page 673. Information on customizing cataloged procedures, IBM REXX™ EXECs, and EXECs is available in "Tailoring cataloged procedures, REXX EXECs, and EXECs" on page 467.

Note: As of z/OS V1R7, the z/OS C/C++ compiler has been rebranded to z/OS XL C/C++.

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

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

Typographical conventions

The following table explains the typographical conventions used in this document.

Table 1. Typographical conventions

Typeface	Indicates	Example
bold	Commands, executable names, compiler options and pragma directives that contain lower-case letters.	The xlc utility provides two basic compiler invocation commands, **xlc** and **xlC** (**xlc++**), along with several other compiler invocation commands to support various C/C++ language levels and compilation environments.
italics	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	Make sure that you update the *size* parameter if you return more than the *size* requested.
`monospace`	Programming keywords and library functions, compiler built-in functions, file and directory names, examples of program code, command strings, or user-defined names.	If one or two cases of a `switch` statement are typically executed much more frequently than other cases, break out those cases by handling them separately before the `switch` statement.

How to read syntax diagrams

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

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

Symbols

The following symbols may be displayed in syntax diagrams:

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

Syntax items

Syntax diagrams contain many different items. Syntax items include:

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

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

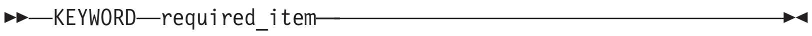
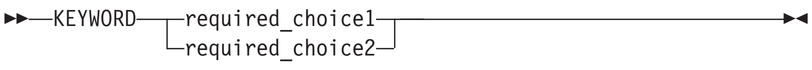
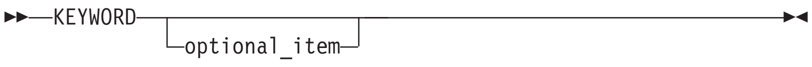
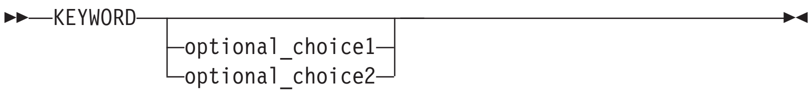
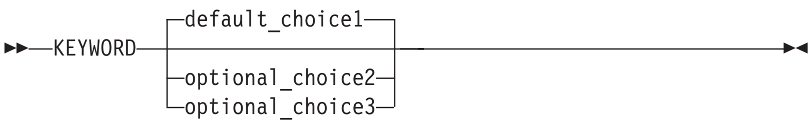
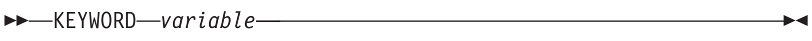
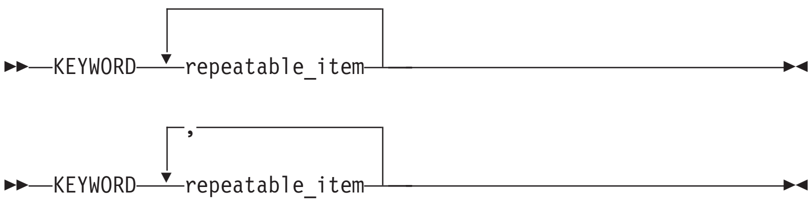
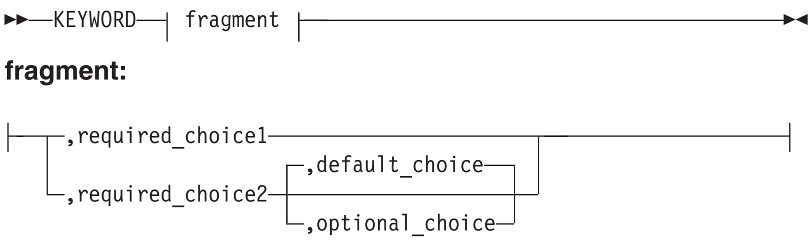
Keywords, variables, and operators may be displayed as required, optional, or default. Fragments, separators, and delimiters may be displayed as required or optional.

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

Syntax examples

The following table provides syntax examples.

Table 2. 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.	 KEYWORD fragment fragment: ,required_choice1 ,required_choice2 ,default_choice ,optional_choice

z/OS XL C/C++ and related documents

This topic summarizes the content of the z/OS XL C/C++ documents and shows where to find related information in other documents.

Table 3. z/OS XL C/C++ and related documents

Document Title and Number	Key Sections/Chapters in the Document
z/OS XL C/C++ Programming Guide, SC09-4765	Guidance information for: • XL C/C++ input and output • Debugging z/OS XL C programs that use input/output • Using linkage specifications in C++ • Combining C and assembler • Creating and using DLLs • Using threads in z/OS UNIX® System Services applications • Reentrancy • Handling exceptions, error conditions, and signals • Performance optimization • Network communications under z/OS UNIX System Services • Interprocess communications using z/OS UNIX System Services • Structuring a program that uses C++ templates • Using environment variables • Using System Programming C facilities • Library functions for the System Programming C facilities • Using run-time user exits • Using the z/OS XL C multitasking facility • Using other IBM products with z/OS XL C/C++ (IBM CICS® Transaction Server for z/OS, CSP, DWS, IBM DB2®, IBM GDDM®, IBM IMS™, ISPF, IBM QMF™) • Internationalization: locales and character sets, code set conversion utilities, mapping variant characters • POSIX character set • Code point mappings • Locales supplied with z/OS XL C/C++ • Charmap files supplied with z/OS XL C/C++ • Examples of charmap and locale definition source files • Converting code from coded character set IBM-1047 • Using built-in functions • Programming considerations for z/OS UNIX System Services C/C++
z/OS XL C/C++ User's Guide, SC09-4767	Guidance information for: • z/OS XL C/C++ examples • Compiler options • Binder options and control statements • Specifying Language Environment run-time options • Compiling, IPA Linking, binding, and running z/OS XL C/C++ programs • Utilities (Object Library, CXXFILT, DSECT Conversion, Code Set and Locale, ar and make, BPXBATCH, c89, xlc) • Diagnosing problems • Cataloged procedures and IBM REXX EXECs • Customizing default options for the z/OS XL C/C++ compiler

Table 3. z/OS XL C/C++ and related documents (continued)

Document Title and Number	**Key Sections/Chapters in the Document**
z/OS XL C/C++ Language Reference, SC09-4815	Reference information for: • The C and C++ languages • Lexical elements of z/OS XL C and C++ • Declarations, expressions, and operators • Implicit type conversions • Functions and statements • Preprocessor directives • C++ classes, class members, and friends • C++ overloading, special member functions, and inheritance • C++ templates and exception handling • z/OS XL C and C++ compatibility
z/OS XL C/C++ Messages, GC09-4819	Provides error messages and return codes for the compiler, and its related application interface libraries and utilities. For the XL C/C++ run-time library messages, refer to *z/OS Language Environment Run-Time Messages*, SA22-7566. For the c89 and xlc utility messages, refer to *z/OS UNIX System Services Messages and Codes*, SA22-7807.
z/OS XL C/C++ Run-Time Library Reference, SA22-7821	Reference information for: • header files • library functions
z/OS C Curses, SA22-7820	Reference information for: • Curses concepts • Key data types • General rules for characters, renditions, and window properties • General rules of operations and operating modes • Use of macros • Restrictions on block-mode terminals • Curses functional interface • Contents of headers • The terminfo database
z/OS XL C/C++ Compiler and Run-Time Migration Guide for the Application Programmer, GC09-4913	Guidance and reference information for: • Common migration questions • Application executable program compatibility • Source program compatibility • Input and output operations compatibility • Class library migration considerations • Changes between releases of z/OS • Pre-z/OS C and C++ compilers to current compiler migration • Other migration considerations
z/OS Metal C Programming Guide and Reference, SA23-2225	Guidance and reference information for: • Metal C run time • Metal C programming • AR mode
Standard C++ Library Reference, SC09-4949	The documentation describes how to use the following three main components of the Standard C++ Library to write portable C/C++ code that complies with the ISO standards: • ISO Standard C Library • ISO Standard C++ Library • Standard Template Library (C++) The ISO Standard C++ library consists of 51 required headers. These 51 C++ library headers (along with the additional 18 Standard C headers) constitute a hosted implementation of the C++ library. Of these 51 headers, 13 constitute the Standard Template Library, or STL.

Table 3. z/OS XL C/C++ and related documents (continued)

Document Title and Number	Key Sections/Chapters in the Document
z/OS Common Debug Architecture User's Guide, SC09-7653	This documentation is the user's guide for IBM's `libddpi` library. It includes: • Overview of the architecture • Information on the order and purpose of API calls for model user applications and for accessing DWARF information • Information on using the Common Debug Architecture with C/C++ source This user's guide is part of the Run-Time Library Extensions documentation.
z/OS Common Debug Architecture Library Reference, SC09-7654	This documentation is the reference for IBM's `libddpi` library. It includes: • General discussion of Common Debug Architecture • Description of APIs and data types related to stacks, processes, operating systems, machine state, storage, and formatting This reference is part of the Run-Time Library Extensions documentation.
DWARF/ELF Extensions Library Reference, SC09-7655	This documentation is the reference for IBM's extensions to the `libdwarf` and `libelf` libraries. It includes information on: • Consumer APIs • Producer APIs This reference is part of the Run-Time Library Extensions documentation.
Debug Tool documentation, available on the Debug Tool for z/OS library page on the World Wide Web	The documentation, which is available at www.ibm.com/software/awdtools/debugtool/library/, provides guidance and reference information for debugging programs, using Debug Tool in different environments, and language-specific information.
README file (Shipped with Program materials)	Partitioned data set CBC.SCCNDOC on the product tape contains the README member, which provides additional information for using the z/OS XL C/C++ licensed program, including late changes to z/OS XL C/C++ publications. To access any README files that were published after the ship date, go to www.ibm.com/support/docview.wss?uid=swg27007531.

Note: For complete and detailed information on linking and running with Language Environment services and using the Language Environment run-time options, refer to *z/OS Language Environment Programming Guide*, SA22-7561. For complete and detailed information on using interlanguage calls, refer to *z/OS Language Environment Writing Interlanguage Communication Applications*, SA22-7563.

The following table lists the z/OS XL C/C++ and related documents. The table groups the documents according to the tasks they describe.

Table 4. Documents by task

Tasks	Documents
Planning, preparing, and migrating to z/OS XL C/C++	• *z/OS XL C/C++ Compiler and Run-Time Migration Guide for the Application Programmer*, GC09-4913 • *z/OS Language Environment Customization*, SA22-7564 • *z/OS Language Environment Run-Time Application Migration Guide*, GA22-7565 • *z/OS UNIX System Services Planning*, GA22-7800 • *z/OS Planning for Installation*, GA22-7504
Installing	• *z/OS Program Directory* • *z/OS Planning for Installation*, GA22-7504 • *z/OS Language Environment Customization*, SA22-7564
Option customization	• *z/OS XL C/C++ User's Guide*, SC09-4767

Table 4. Documents by task (continued)

Tasks	Documents
Coding programs	• *z/OS XL C/C++ Run-Time Library Reference*, SA22-7821 • *z/OS XL C/C++ Language Reference*, SC09-4815 • *z/OS XL C/C++ Programming Guide*, SC09-4765 • *z/OS Metal C Programming Guide and Reference*, SA23-2225 • *z/OS Language Environment Concepts Guide*, SA22-7567 • *z/OS Language Environment Programming Guide*, SA22-7561 • *z/OS Language Environment Programming Reference*, SA22-7562
Coding and binding programs with interlanguage calls	• *z/OS XL C/C++ Programming Guide*, SC09-4765 • *z/OS XL C/C++ Language Reference*, SC09-4815 • *z/OS Language Environment Programming Guide*, SA22-7561 • *z/OS Language Environment Writing Interlanguage Communication Applications*, SA22-7563 • *z/OS MVS Program Management: User's Guide and Reference*, SA22-7643 • *z/OS MVS Program Management: Advanced Facilities*, SA22-7644
Compiling, binding, and running programs	• *z/OS XL C/C++ User's Guide*, SC09-4767 • *z/OS Language Environment Programming Guide*, SA22-7561 • *z/OS Language Environment Debugging Guide*, GA22-7560 • *z/OS MVS Program Management: User's Guide and Reference*, SA22-7643 • *z/OS MVS Program Management: Advanced Facilities*, SA22-7644
Compiling and binding applications in the z/OS UNIX System Services (z/OS UNIX) environment	• *z/OS XL C/C++ User's Guide*, SC09-4767 • *z/OS UNIX System Services User's Guide*, SA22-7801 • *z/OS UNIX System Services Command Reference*, SA22-7802 • *z/OS MVS Program Management: User's Guide and Reference*, SA22-7643 • *z/OS MVS Program Management: Advanced Facilities*, SA22-7644
Debugging programs	• README file • *z/OS XL C/C++ User's Guide*, SC09-4767 • *z/OS XL C/C++ Messages*, GC09-4819 • *z/OS XL C/C++ Programming Guide*, SC09-4765 • *z/OS Language Environment Programming Guide*, SA22-7561 • *z/OS Language Environment Debugging Guide*, GA22-7560 • *z/OS Language Environment Run-Time Messages*, SA22-7566 • *z/OS UNIX System Services Messages and Codes*, SA22-7807 • *z/OS UNIX System Services User's Guide*, SA22-7801 • *z/OS UNIX System Services Command Reference*, SA22-7802 • *z/OS UNIX System Services Programming Tools*, SA22-7805 • Debug Tool documentation, available on the Debug Tool Library page on the World Wide Web (www.ibm.com/software/awdtools/debugtool/library/) • z/OS messages database, available on the z/OS Library page at www.ibm.com/systems/z/os/zos/bkserv/ through the LookAt Internet message search utility.
Developing debuggers and profilers	• *z/OS Common Debug Architecture User's Guide*, SC09-7653 • *z/OS Common Debug Architecture Library Reference*, SC09-7654 • *DWARF/ELF Extensions Library Reference*, SC09-7655
Packaging XL C/C++ applications	• *z/OS XL C/C++ Programming Guide*, SC09-4765 • *z/OS XL C/C++ User's Guide*, SC09-4767
Using shells and utilities in the z/OS UNIX System Services environment	• *z/OS XL C/C++ User's Guide*, SC09-4767 • *z/OS UNIX System Services Command Reference*, SA22-7802 • *z/OS UNIX System Services Messages and Codes*, SA22-7807
Using sockets library functions in the z/OS UNIX System Services environment	• *z/OS XL C/C++ Run-Time Library Reference*, SA22-7821

Table 4. Documents by task (continued)

Tasks	**Documents**
Using the ISO Standard C++ Library to write portable C/C++ code that complies with ISO standards	• *Standard C++ Library Reference*, SC09-4949
Porting a z/OS UNIX System Services application to z/OS	• *z/OS UNIX System Services Porting Guide* This guide contains useful information about supported header files and C functions, sockets in z/OS UNIX System Services, process management, compiler optimization tips, and suggestions for improving the application's performance after it has been ported. The *Porting Guide* is available as a PDF file which you can download, or as web pages which you can browse, at the following web address: www.ibm.com/servers/eserver/zseries/zos/unix/bpxa1por.html
Working in the z/OS UNIX System Services Parallel Environment	• *z/OS UNIX System Services Parallel Environment: Operation and Use*, SA22-7810 • *z/OS UNIX System Services Parallel Environment: MPI Programming and Subroutine Reference*, SA22-7812
Performing diagnosis and submitting an Authorized Program Analysis Report (APAR)	• *z/OS XL C/C++ User's Guide*, SC09-4767 • CBC.SCCNDOC(APAR) on z/OS XL C/C++ product tape
Note: For information on using the prelinker, see Appendix A, "Prelinking and linking z/OS XL C/C++ programs," on page 601.	

Softcopy documents

The z/OS XL C/C++ publications are supplied in PDF and IBM BookMaster® formats on the following CD: *z/OS Collection*, SK3T-4269. They are also available at www.ibm.com/software/awdtools/czos/library/.

To read a PDF file, use the Adobe® Reader. If you do not have the Adobe Reader, you can download it (subject to Adobe license terms) from the Adobe web site at www.adobe.com.

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

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

Softcopy examples

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

- *z/OS XL C/C++ Language Reference*, SC09-4815
- *z/OS XL C/C++ User's Guide*, SC09-4767
- *z/OS XL C/C++ Programming Guide*, SC09-4765

In the following documents, a label on an example indicates that the example is distributed as a softcopy file:

- *z/OS XL C/C++ Language Reference*, SC09-4815
- *z/OS XL C/C++ Programming Guide*, SC09-4765
- *z/OS XL C/C++ User's Guide*, SC09-4767

The label is the name of a member in the CBC.SCCNSAM data set. The labels begin with the form CCN or CLB. Examples labelled as CLB appear only in the *z/OS XL C/C++ User's Guide*, while examples labelled as CCN appear in all three documents, and are further distinguished by *x* following CCN, where *x* represents one of the following:

- R and X refer to *z/OS XL C/C++ Language Reference*, SC09-4815
- G refers to *z/OS XL C/C++ Programming Guide*, SC09-4765
- U refers to *z/OS XL C/C++ User's Guide*, SC09-4767

z/OS XL C/C++ on the World Wide Web

Additional information on z/OS XL C/C++ is available on the World Wide Web on the z/OS XL C/C++ home page at: www.ibm.com/software/awdtools/czos/

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

Where to find more information

Please see *z/OS Information Roadmap* for an overview of the documentation associated with z/OS, including the documentation available for z/OS Language Environment.

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.

Technical support

Additional technical support is available from the z/OS XL C/C++ Support page. This page provides a portal with search capabilities to a large selection of technical support FAQs and other support documents. You can find the z/OS XL C/C++ Support page on the Web at:

www.ibm.com/software/awdtools/czos/support

If you cannot find what you need, you can e-mail:

compinfo@ca.ibm.com

For the latest information about z/OS XL C/C++, visit the product information site at:

www.ibm.com/software/awdtools/czos/

For information about boosting performance, productivity and portability, visit the C/C++ Cafe at:

http://www-949.ibm.com/software/rational/cafe/community/ccpp

How to send your comments

Your feedback is important in helping to provide accurate and high-quality information. If you have any comments about this document or any other z/OS XL C/C++ documentation, send your comments by e-mail to:

compinfo@ca.ibm.com

Be sure to include the name of the document, the part number of the document, the version of, and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).

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

Chapter 1. About IBM z/OS XL C/C++

The XL C/C++ feature of the IBM z/OS licensed program provides support for C and C++ application development on the z/OS platform.

z/OS XL C/C++ includes:
- A C compiler (referred to as the z/OS XL C compiler)
- A C++ compiler (referred to as the z/OS XL C++ compiler)
- Performance Analyzer host component, which supports the IBM C/C++ Productivity Tools for IBM OS/390® product
- A set of utilities for C/C++ application development

Notes:
1. The Run-Time Library Extensions base element was introduced in z/OS V1R5. It includes the Common Debug Architecture (CDA) Libraries, the c89 utility, and, as of z/OS V1R6, the xlc utility. The Common Debug Architecture provides a consistent and common format for debugging information across the various languages and operating systems that are supported on the IBM System z® platform. Run-Time Library Extensions also includes legacy libraries to support existing programs. These are the UNIX System Laboratories (USL) I/O Stream Library and USL Complex Mathematics Library. The IBM Open Class® Library is not supported.
2. The Standard C++ Library is included with Language Environment libraries.
3. The z/OS XL C/C++ compiler works with the mainframe interactive Debug Tool product and IBM WebSphere® Developer for System z, integrated with IBM Debug Tool for z/OS and IBM Debug Tool Utilities and Advanced Functions for z/OS.

IBM offers the C and C++ compilers on other platforms, such as the IBM AIX®, Linux®, IBM OS/400®, and IBM z/VM® operating systems. The C compiler is also available on the IBM VSE/ESA™ platform.

What's new for z/OS V1R11 XL C/C++

z/OS XL C/C++ has made the following performance and usability enhancements for the z/OS V1R11 release:

dbgld — Create a module map for debugging
The compiler creates a debug side file for each compilation unit if the DEBUG compiler option is specified. The dbgld utility opens all of the debug side files associated with the module and stores all of the functions, global variables, external types, and source files in a module map. z/OS XL C/C++ V1R11 introduces the **dbgld -c** feature to add captured source file to the module map, which consists of all files that contain executable lines of code. For more information about the new **dbgld -c** feature, see Chapter 22, "dbgld — Create a module map for debugging," on page 573.

CDADBGLD — Create a debug side file for the module map
The CDADBGLD utility is the MVS batch equivalent of the dbgld command and is useful for the users who do not have z/OS UNIX System Services. The compiler creates a debug side file for each compilation unit if the DEBUG compiler option is specified. The path names or data set names of all the debug side files are then stored in the module, which is an executable file or a DLL. The

© Copyright IBM Corp. 1998, 2009

CDADBGLD utility opens all of the debug side files associated with the module and stores all of the functions, global variables, external types, and source files in a module map. The performance of the debugger, especially the start time, will be significantly improved if the CDADBGLD utility is executed before the execution of a debugger. For more information, see Chapter 23, "CDADBGLD — Create a debug side file for the module map," on page 575.

makedepend utility improvements
The stand-alone makedepend utility is no longer being enhanced in V1R11. Instead, this utility is superseded in favour of the xlc option **-qmakedep**. For more information about the new **-qmakedep** compiler option and its flag options, see Chapter 4, "Compiler Options," on page 41 and Chapter 24, "xlc — Compiler invocation using a customizable configuration file," on page 577.

Macro definitions listing
The SHOWMACROS option is used to display macros definitions to preprocessed output. This option can help to determine the available functionality in the XL C/C++ compiler. The macro listing may prove useful in debugging complex macro expansions. See "SHOWMACROS | NOSHOWMACROS" on page 224 for more information about the SHOWMACROS option.

Compare-and-trap instructions generation
z/OS V1R11 XL C/C++ introduces the RTCHECK option to generate compare-and-trap instructions which perform certain types of runtime checking. The messages can be used to debug the C and C++ programs. For more information about the RTCHECK option, see "RTCHECK | NORTCHECK" on page 217.

IPA Pseudo-C listing enhancement
When the LIST option is specified, the z/OS XL C/C++ compiler generates a listing file for each source file named on the command line. When used with the REPORT option, pseudo-C code listing files are produced to show how sections of code have been optimized in both compile and link phases. You can use this information to understand your application code and to tune your code for better performance. For more information, see the REPORT compiler option description in "REPORT | NOREPORT" on page 210.

Suppress skipped source
When a listing file is generated using the SOURCE option, the SKIPSRC option can be used to determine whether the source statements skipped by the compiler are shown in the source section of the listing file. This feature improves readability of the listing by displaying or skipping the false blocks of a preprocessing directive. For more information about the SKIPSRC option, see "SKIPSRC" on page 225.

Prefetch instructions generation
z/OS XL C/C++ V1R11 introduces the PREFETCH option to insert prefetch instructions automatically where there are opportunities to improve code performance. See "PREFETCH | NOPREFETCH" on page 206 for more information about the PREFETCH compiler option

PROLOG and EPILOG static function support

Starting from z/OS XL C/C++ V1R11, the PROLOG and EPILOG statements add support for all extern (or file scope) functions in the same compilation unit, or all extern and static functions in the same compilation unit, depending on the suboption used. See "EPILOG (C only)" on page 106 and "PROLOG (C only)" on page 206 for more information about the static function support feature.

C++ unicode literal support

z/OS XL C/C++ V1R11 adds support for the unicode literals and character types. This feature allows writing of programs that handle the UTF-16 and UTF-32 encodings. Two predefined macros: __IBM_UTF_LITERAL and __IBMCPP_UTF_LITERAL__ are supported. For further information about the macros, see "LANGLVL" on page 147.

Zero extent array members support

z/OS XL C/C++ V1R11 adds support for Zero extent array members, which are GNU C/C++ extensions. The Zero extent array members are useful as the last element of a structure, which is really a header for a variable-length object. For more information about the Zero extent array members feature, see *z/OS XL C/C++ Language Reference*.

GCC statement expression support

z/OS XL C/C++ V1R11 adds support for GCC statement expression. This feature enables a compound statement enclosed in parentheses to appear as an expression and allows you to use loops, switches, and local variables within an expression. For more information about the GCC statement expression feature, see *z/OS XL C/C++ Language Reference*.

New compiler suboption

z/OS XL C/C++ V1R11 introduces the following new compiler suboptions:

- EPILOG(EXTERN | ALL)
- LANGLVL(EXTENDED0X | EXTENDEDFRIEND | EXTERNTEMPLATE | DEPENDENTBASELOOKUP)
- NAMEMANGLING(zOSV1R11_ANSI | zOSV1R10_ANSI | zOSV1R9_ANSI | zOSV1R8_ANSI | zOSV1R7_ANSI | OSV2R10)
- PROLOG(EXTERN | ALL)
- TARGET(zOSV1R11)

For more information, see "EPILOG (C only)" on page 106, "PROLOG (C only)" on page 206, "TARGET" on page 240, "LANGLVL" on page 147, and "NAMEMANGLING (C++ only)" on page 185.

New macros

z/OS XL C/C++ V1R11 introduces the following new macros:

- __IBM_UTF_LITERAL
- __IBMCPP_UTF_LITERAL__ (C++ only)
- __IBMCPP_EXTENDED_FRIEND (C++ only)
- __IBMCPP_EXTERN_TEMPLATE (C++ only)

For further information about the predefined macros related to language levels, see "LANGLVL" on page 147.

New pragma directives

z/OS XL C/C++ V1R11 introduces following new suboptions for **#pragma namemangling**:

- zosv1r11_ansi
- zosv1r10_ansi
- zosv1r9_ansi
- zosv1r8_ansi
- zosv1r7_ansi
- zosv2r10

z/OS XL C/C++ V1R11 also introduces `main` as a new suboption for the **#pragma prolog** and **#pragma epilog** directives.

For more information about these #pragma directives, see *z/OS XL C/C++ Language Reference*.

Performance improvements

As an ongoing effort, improvements are provided in the generated code to take advantage of new instructions in the hardware architecture. You can benefit from these improvements by recompiling your existing source without code changes.

Compiling with AFP(VOLATILE) is no longer necessary with the latest CICS® in release V1R11, which has Extended MVS Linkage Convention support and has this implemented natively. For a summary of whether to recompile the source codes under various situations using AFP(VOLATILE) and CICS, see Table 22 on page 116 in “FLOAT” on page 114.

The z/OS XL C/C++ compiler also provide additional built-in functions to help your programs use selected hardware instructions directly. For information about these built-in functions, see the built-in functions described in *z/OS XL C/C++ Programming Guide*.

For information about the changes that the Language Environment element has made for z/OS V1R11, see "What's New in Language Environment for z/OS" in *z/OS Language Environment Concepts Guide*.

The XL C/C++ compilers

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

The C language

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

The C++ language

The C++ language is based on the C language and includes all of the advantages of C listed above. In addition, C++ also supports object-oriented concepts, generic types or templates, and an extensive library. For a detailed description of the differences between z/OS XL C++ and z/OS XL C, refer to *z/OS XL C/C++ Language Reference*.

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

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

Common features of the z/OS XL C and XL C++ compilers

The XL C and XL C++ compilers, when used with the Language Environment element, offer many features to increase your productivity and improve program execution times:

- Optimization support:
  - Extra Performance Linkage (XPLINK) function calling convention, which has the potential for a significant performance increase when used in an environment of frequent calls between small functions. XPLINK makes subroutine calls more efficient by removing non-essential instructions from the main path.
  - Algorithms to take advantage of the IBM zSeries® architecture to achieve improved optimization and memory usage through the OPTIMIZE and IPA compiler options.
  - The OPTIMIZE compiler option, which instructs the compiler to optimize the machine instructions it generates to try to produce faster-running object code and improve application performance at run time.
  - Interprocedural Analysis (IPA), to perform optimizations across procedural and compilation unit boundaries, thereby optimizing application performance at run time.
  - Additional optimization capabilities are available with the INLINE compiler option.
- DLLs (dynamic link libraries) to share parts among applications or parts of applications, and dynamically link to exported variables and functions at run time.

  DLLs allow a function reference or a variable reference in one executable to use a definition located in another executable at run time.

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

  With reentrancy, many users can simultaneously run a program. A reentrant program uses less storage if it is stored in the Link Pack Area (LPA) or the Extended Link Pack Area (ELPA) and simultaneously run by multiple users. It also reduces processor I/O when the program starts up, and improves program performance by reducing the transfer of data to auxiliary storage. z/OS XL C programmers can design programs that are naturally reentrant. For those programs that are not naturally reentrant, z/OS XL C programmers can use constructed reentrancy. To do this, compile programs with the RENT option and use the program management binder supplied with z/OS or the Language Environment prelinker and program management binder. The z/OS XL C++ compiler always uses the constructed reentrancy algorithms.

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

  z/OS UNIX System Services is the IBM implementation of the open operating system environment, as defined in the XPG4 and POSIX standards.
- Support features in the following standards at the system level:
  - *ISO/IEC 9899:1999*
  - *ISO/IEC 9945-1:1990 (POSIX-1)/IEEE POSIX 1003.1-1990*
  - The core features of *IEEE POSIX 1003.1a, Draft 6, July 1991*
  - *IEEE Portable Operating System Interface (POSIX) Part 2, P1003.2*
  - The core features of *IEEE POSIX 1003.4a, Draft 6, February 1992* (the IEEE POSIX committee has renumbered POSIX.4a to POSIX.1c)
  - *X/Open CAE Specification, System Interfaces and Headers, Issue 4 Version 2*
  - The core features of *IEEE 754-1985 (R1990) IEEE Standard for Binary Floating-Point Arithmetic (ANSI)*, as applicable to the IBM System z environment.
  - *X/Open CAE Specification, Networking Services, Issue 4*
  - A subset of *IEEE Std. 1003.1-2001 (Single UNIX Specification, Version 3)*.
- Support for the Euro currency

z/OS XL C compiler-specific features

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

- The ability to write portable code that supports the following standards:
  - *ISO/IEC 9899:1999*
  - *ANSI/ISO 9899:1990[1992]* (formerly *ANSI X3.159-1989 C*)
  - *X/Open Specification Programming Languages, Issue 3, Common Usage C*
  - *FIPS-160*
- System programming capabilities, which allow you to use z/OS XL C in place of assembler
- Extensions of the standard definitions of the C language to provide you with support for the z/OS environment, such as fixed-point (packed) decimal data support

z/OS XL C++ compiler-specific features

In addition to the features common to z/OS XL C and XL C++, the z/OS XL C++ compiler supports the *Programming languages - C++ (ISO/IEC 14882:1998)* standard. Also, it further conforms to the *Programming languages - C++ (ISO/IEC 14882:2003(E))* standard, which incorporates the latest Technical Corrigendum 1.

Class libraries

z/OS V1R10 XL C/C++ uses the following thread-safe class libraries:

- Standard C++ Library, including the Standard Template Library (STL), and other library features of *Programming languages - C++ (ISO/IEC 14882:1998)* and *Programming languages - C++ (ISO/IEC 14882:2003(E))*
- UNIX System Laboratories (USL) C++ Language System Release I/O Stream and Complex Mathematics Class Libraries

Note: As of z/OS V1R5, all application development using the C/C++ IBM Open Class Library (Application Support Class and Collection Class Libraries) is not supported. As of z/OS V1R9, the execution of applications using the C/C++ IBM Open Class Library is not supported.

For new code and enhancements to existing applications, the Standard C++ Library should be used. The Standard C++ Library includes the following:

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

Utilities

The z/OS XL C/C++ compilers provide the following utilities:

- The xlc utility to invoke the compiler using a customizable configuration file.
- The c89 utility to invoke the compiler using host environment variables.
- The CXXFILT utility to map z/OS XL C++ mangled names to their original function names.
- The DSECT conversion utility to convert descriptive assembler DSECTs into z/OS XL C/C++ data structures.
- The makedepend utility to derive all dependencies in the source code and write these into the makefile. The **make** command will determine which source files to recompile, whenever a dependency has changed. This frees the user from manually monitoring such changes in the source code.
- The CDAHLASM utility, which produces debug information in DWARF (for Metal C applications) and ADATA formats. This utility uses the HLASM assembler to compile the source files produced by compiling Metal C code.

Language Environment utilities include:

- The object library utility (C370LIB; also known as EDCALIAS) to update partitioned data set (PDS and PDSE) libraries of object modules. The object library utility supports XPLINK, IPA, and LP64 compiled objects.

  Note: In this document, references to *Partitioned Data Set* (PDS) include both the PDS and *Partitioned Data Set Extended* (PDSE) physical formats, unless stated otherwise.
- The prelinker which combines object modules that comprise a z/OS XL C/C++ application to produce a single object module. The prelinker supports only object and extended object format input files, and does not support GOFF.

Note: IBM has stabilized the prelinker. Further enhancements will not be made to the prelinker utility. IBM recommends that you use the binder instead of the prelinker and linker.

dbx

You can use the **dbx** shell command to debug programs, as described in *z/OS UNIX System Services Programming Tools* and *z/OS UNIX System Services Command Reference*.

Refer to www.ibm.com/servers/eserver/zseries/zos/unix/bpxa1dbx.html for further information about `dbx`.

Language Environment element

z/OS XL C/C++ exploits the C/C++ run-time environment and library of run-time services available with the Language Environment element provided with z/OS. For an introduction to the Language Environment element, see "Overview" in *z/OS Language Environment Concepts Guide*.

Note: The Language Environment run-time option TRAP(ON) should be set when using z/OS XL C/C++. Refer to *z/OS Language Environment Programming Reference* for details on the Language Environment run-time options.

Language Environment downward compatibility

Language Environment downward compatibility support is provided. Assuming that you have met the required programming guidelines and restrictions, described in "Downward Compatibility Considerations" in *z/OS Language Environment Programming Guide*, this support enables you to develop applications on higher release levels of z/OS for use on platforms that are running lower release levels of z/OS. In XL C and XL C++, downward compatibility support is provided through the XL C/C++ TARGET compiler option. See "TARGET" on page 240 for details on this compiler option.

Note: As of z/OS V1R3, the executables produced with the binder's COMPAT=CURRENT setting will not run on lower levels of z/OS. You will have to explicitly override to a particular program object level, or use the COMPAT=MIN setting introduced in z/OS V1R3.

About prelinking, linking, and binding

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

Normally, the program management binder is used to perform the bind step. However, in many cases the prelink and link steps can be used in place of the bind step. When they cannot be substituted, and the program management binder alone must be used, it will be stated.

The terms *bind* and *link* have multiple meanings.

- With respect to building an application:

  In both instances, the program management binder is performing the actual processing of converting the object file(s) into the application executable module.

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

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

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

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

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

- With respect to executing code in an application:

  The *linkage definition* refers to the program call linkage between program functions and methods. This includes the passing of control and parameters. Refer to "Program Linkage" in *z/OS XL C/C++ Language Reference* for more information about linkage specification.

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

Notes on the prelinking process

You cannot use the prelinker if you are using the XPLINK, GOFF, or LP64 compiler options. IBM recommends using the binder instead of the prelinker whenever possible.

The prelinker was designed to process long names and support constructed reentrancy in earlier versions of the C compiler on the IBM MVS™ and OS/390 operating systems. The Language Environment prelinker provides output that is compatible with the linkage editor, which is shipped with the binder.

The *binder* is designed to include the functions of the prelinker, the linkage editor, the loader, and a number of APIs to manipulate the program object. Thus, the binder is a superset of the linkage editor. Its functionality provides a high level of compatibility with the prelinker and linkage editor, but provides additional functionality in some areas. Generally, the terms *binding* and *linking* are interchangeable. In particular, the binder supports:

- Inputs from the object module
- XOBJ, GOFF, load module and program object
- Auto call resolutions from z/OS UNIX archives and C370LIB object directories
- Long external names
- All prelinker control statements

Notes:

1. You need to use the binder for XPLINK objects.
2. As of z/OS V1R7, the Hierarchical File System (HFS) functionality has been stabilized and zSeries File System (zFS) is the strategic file system for z/OS UNIX System Services. The term *z/OS UNIX file system* includes both HFS and zFS.

For more information about the compatibility between the binder, and the linker and prelinker, see *z/OS MVS Program Management: User's Guide and Reference*.

Updates to the prelinking, linkage-editing, and loading functions that are performed by the binder are delivered through the binder. If you use the Language Environment prelinker and the linkage editor (supplied through the binder), you have to apply the latest maintenance for the Language Environment element as well as the binder.

File format considerations

You can use the binder in place of the prelinker and linkage editor but there are exceptions involving file format considerations. For further information, on when you cannot use the binder, see Chapter 9, "Binding z/OS XL C/C++ programs," on page 401.

The program management binder

The binder provided with z/OS combines the object modules, load modules, and program objects comprising an application. It produces a single z/OS output program object or load module that you can load for execution. The binder supports all C and C++ code, provided that you store the output program in a PDSE member or a z/OS UNIX System Services file.

If you cannot use a PDSE member or z/OS UNIX file, and your program contains C++ code, or C code that is compiled with any of the RENT, LONGNAME, DLL or IPA compiler options, you must use the prelinker. C and C++ code compiled with the GOFF or XPLINK compiler options cannot be processed by the prelinker.

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

- Faster rebinds when recompiling and rebinding a few of your source files
- Rebinding at the single compile unit level of granularity (except when you use the IPA compile-time option)
- Input of object modules, load modules, and program objects
- Improved long name support:
  - Long names do not get converted into prelinker generated names
  - Long names appear in the binder maps, enabling full cross-referencing
  - Variables do not disappear after prelink
  - Fewer steps in the process of producing your executable program

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

z/OS UNIX System Services

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

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

z/OS UNIX provides support for both existing z/OS applications and new z/OS UNIX applications through the following:

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

ar	Creates and maintains library archives
as	Invokes HLASM to create assembler applications
BPXBATCH	Allows you to submit batch jobs that run shell commands, scripts, or z/OS XL C/C++ executable files in z/OS UNIX files from a shell session
c89	Uses host environment variables to compile, assemble, and bind z/OS UNIX, C/C++ and assembler applications
dbx	Provides an environment to debug and run programs
gencat	Merges the message text source files (usually *.msg) into a formatted message catalog file (usually *.cat)
iconv	Converts characters from one code set to another
ld	Combines object files and archive files into an output executable file, resolving external references
lex	Automatically writes large parts of a lexical analyzer based on a description that is supplied by the programmer
localedef	Creates a compiled locale object
make	Helps you manage projects containing a set of interdependent files, such as a program with many z/OS source and object files, keeping all such files up to date with one another
xlc	Allows you to invoke the compiler using a customizable configuration file
yacc	Allows you to write compilers and other programs that parse input according to strict grammar rules

  - Support for other utilities such as:

dspcat	Displays all or part of a message catalog
dspmsg	Displays a selected message from a message catalog
mkcatdefs	Preprocesses a message source file for input to the gencat utility

runcat	Invokes mkcatdefs and pipes the message catalog source data (the output from mkcatdefs) to gencat

- Access to the Hierarchical File System (HFS), with support for the POSIX.1 and XPG4 standards
- Access to the zSeries File System (zFS), which provides performance improvements over HFS, and also supports the POSIX.1 and XPG4 standards
- z/OS XL C/C++ I/O routines, which support using z/OS UNIX files, standard z/OS data sets, or a mixture of both
- Application threads (with support for a subset of POSIX.4a)
- Support for z/OS XL C/C++ DLLs

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

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

z/OS XL C/C++ applications with z/OS UNIX System Services C functions

All z/OS UNIX System Services C functions are available at all times. In some situations, you must specify the POSIX(ON) run-time option. This is required for the POSIX.4a threading functions, the POSIX `system()` function, and signal handling functions where the behavior is different between POSIX/XPG4 and ISO. Refer to *z/OS XL C/C++ Run-Time Library Reference* for more information about requirements for each function.

You can invoke a z/OS XL C/C++ program that uses z/OS UNIX C functions using the following methods:

- Directly from a shell.
- From another program, or from a shell, using one of the `exec` family of functions, or the BPXBATCH utility from TSO or MVS batch.
- Using the POSIX `system()` call.
- Directly through TSO or MVS batch without the use of the intermediate BPXBATCH utility. In some cases, you may require the POSIX(ON) run-time option.

Input and output

The z/OS XL C/C++ run-time library that supports the z/OS XL C/C++ compiler supports different input and output (I/O) interfaces, file types, and access methods. The Standard C++ Library provides additional support.

I/O interfaces

The z/OS XL C/C++ run-time library supports the following I/O interfaces:

C Stream I/O
This is the default and the ISO-defined I/O method. This method processes all input and output on a per-character basis.

Record I/O

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

TCP/IP Sockets I/O

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

The z/OS UNIX socket application program interface (API) provides support for both UNIX domain sockets and Internet domain sockets. UNIX domain sockets, or *local sockets*, allow interprocess communication within z/OS, independent of TCP/IP. Local sockets behave like traditional UNIX sockets and allow processes to communicate with one another on a single system. With Internet sockets, application programs can communicate with each other in the network using TCP/IP.

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

File types

In addition to conventional files, such as sequential files and partitioned data sets, the z/OS XL C/C++ run-time library supports the following file types:

Virtual Storage Access Method (VSAM) data sets

z/OS XL C/C++ has native support for the following VSAM data sets:

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

For more information on how to perform I/O operations on these VSAM file types, see "Performing VSAM I/O operations" in *z/OS XL C/C++ Programming Guide*.

Hierarchical File System files

z/OS XL C/C++ recognizes Hierarchical File System (HFS) file names. The name specified on the `fopen()` or `freopen()` call has to conform to certain rules. See "Opening Files" in *z/OS XL C/C++ Programming Guide* for the details of these rules. You can create regular HFS files, special character HFS files, or FIFO HFS files. You can also create links or directories.

Note: As of z/OS V1R7, the Hierarchical File System (HFS) functionality has been stabilized and zSeries File System (zFS) is the strategic UNIX System Services file system for z/OS.

Memory files

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

IBM Hiperspace™ expanded storage

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

zSeries File System

zSeries File System (zFS) is a z/OS UNIX file system that can be used in addition to the Hierarchical File System (HFS). zFS may provide performance gains in accessing files that are frequently accessed and updated. The I/O functions in the z/OS XL C/C++ run-time library support zFS.

Additional I/O features

z/OS XL C/C++ provides additional I/O support through the following features:

- Large file support, which enables I/O to and from z/OS UNIX System Services files that are larger than 2 GB (see "large file support" in *z/OS XL C/C++ Language Reference*)
- User error handling for serious I/O failures (SIGIOERR)
- Improved sequential data access performance through enablement of the IBM DFSMS™ software support for 31-bit sequential data buffers and sequential data striping on extended format data sets
- Full support of PDSEs on z/OS (including support for multiple members opened for write)
- Overlapped I/O support under z/OS (NCP, BUFNO)
- Multibyte character I/O functions
- Fixed-point (packed) decimal data type support in formatted I/O functions
- Support for multiple volume data sets that span more than one volume of DASD or tape
- Support for Generation Data Group I/O

The System Programming C facility

The System Programming C (SPC) facility allows you to build applications that do not require dynamic loading of Language Environment libraries. It also allows you to tailor your application for better utilization of the low-level services available on your operating system. SPC offers a number of advantages:

- You can develop applications that can be executed in a customized environment rather than with Language Environment services. Note that if you do not use Language Environment services, only some built-in functions and a limited set of z/OS XL C/C++ run-time library functions are available to you.
- You can substitute the z/OS XL C language in place of assembly language when writing system exit routines by using the interfaces that are provided by SPC.

- SPC lets you develop applications featuring a user-controlled environment in which a z/OS XL C environment is created once and used repeatedly for C function execution from other languages.
- You can utilize co-routines by using a two-stack model to write application service routines. In this model, the application calls on the service routine to perform services independent of the user. The application is then suspended when control is returned to the user application.

Interaction with other IBM products

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

- CICS Transaction Server for z/OS

  You can use the CICS Command-Level Interface to write C/C++ application programs. The CICS Command-Level Interface provides data, job, and task management facilities that are normally provided by the operating system.
- IBM DB2® Universal Database™ for z/OS

  DB2 programs manage data that is stored in relational databases. You can access the data by using a structured set of queries that are written in Structured Query Language (SQL).

  A DB2 program uses SQL statements that are embedded in the application program. The SQL translator (DB2 preprocessor) translates the embedded SQL into host language statements, which are then compiled by the z/OS XL C/C++ compilers. Alternatively, use the SQL compiler option to compile a DB2 program with embedded SQL without using the DB2 preprocessor. The DB2 program processes requests, then returns control to the application program.
- Debug Tool

  z/OS XL C/C++ supports program development by using the Debug Tool. This tool allows you to debug applications in their native host environment, such as CICS Transaction Server for z/OS, IMS, and DB2. Debug Tool provides the following support and function:
  - Step mode
  - Breakpoints
  - Monitor
  - Frequency analysis
  - Dynamic patching

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

  You can specify either data sets or z/OS UNIX System Services files as source files.

  For further information, see www.ibm.com/software/awdtools/debugtool/.
- WebSphere Developer for System z

  z/OS V1R7 XL C/C++ and later releases enable you to use WebSphere Developer for System z V7.0 to improve the efficiency of application development. For information about WebSphere Developer for System z, see: www.ibm.com/software/awdtools/devzseries/.
- IBM C/C++ Productivity Tools for OS/390

  Note: Starting with z/OS V1R5, both the C/C++ compiler optional feature and the Debug Tool product will need to be installed if you wish to use IBM

C/C++ Productivity Tools for OS/390. For more information about Debug Tool, refer to www.ibm.com/software/awdtools/debugtool/.

With the IBM C/C++ Productivity Tools for OS/390 product, you can expand your z/OS application development environment out to the workstation, while remaining close to your familiar host environment. IBM C/C++ Productivity Tools for OS/390 includes the following workstation-based tools to increase your productivity and code quality:

– Performance Analyzer to help you analyze, understand, and tune your C and C++ applications for improved performance
– Distributed Debugger that allows you to debug C or C++ programs from the convenience of the workstation
– Workstation-based editor to improve the productivity of your C and C++ source entry
– Advanced online help, with full text search and hypertext topics as well as printable, viewable, and searchable Portable Document Format (PDF) documents

Note: References to *Performance Analyzer* in this document refer to the IBM OS/390 Performance Analyzer included in the IBM C/C++ Productivity Tools for OS/390 product.

In addition, IBM C/C++ Productivity Tools for OS/390 includes the following host components:

– Debug Tool
– Host Performance Analyzer

Use the Performance Analyzer on your workstation to graphically display and analyze a profile of the execution of your host z/OS XL C or C++ application. Use this information to time and tune your code so that you can increase the performance of your application.

Use the Distributed Debugger to debug your z/OS XL C or C++ application remotely from your workstation. Set a breakpoint with the simple click of the mouse. Use the windowing capabilities of your workstation to view multiple segments of your source and your storage, while monitoring a variable at the same time.

Use the workstation-based editor to quickly develop C and C++ application code that runs on z/OS. Context-sensitive help information is available to you when you need it.

- IBM Fault Analyzer for z/OS

  The IBM Fault Analyzer helps developers analyze and fix application and system failures. It gathers information about an application and the surrounding environment at the time of the abend, providing the developer with valuable information needed for developing and testing new and existing applications. For more information, refer to: www.ibm.com/software/awdtools/faultanalyzer/.

- Application Performance Analyzer for z/OS

  The Application Performance Analyzer for z/OS is an application program performance analysis tool that helps you to:

  – Optimize the performance of your existing application
  – Improve the response time of your online transactions and batch turnaround times
  – Isolate performance problems in applications

  For more information, refer to: www.ibm.com/software/awdtools/apa/.

- ISPF Software Configuration and Library Manager facility (SCLM)

The ISPF Software Configuration and Library Manager facility (SCLM) maintains information about the source code, objects and load modules. It also keeps track of other relationships in your application, such as test cases, JCL, and publications. The SCLM Build function translates input to output, managing not only compilation and linking, but all associating processes required to build an application. This facility helps to ensure that your production load modules match the source in your production source libraries. For more information, refer to: www.ibm.com/software/awdtools/ispf/features/sclm-ov.html.

- Graphical Data Display Manager (GDDM)

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

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

  For more information, refer to: www.ibm.com/software/applications/gddm/.

- Query Management Facility (QMF)

  z/OS XL C supports the Query Management Facility (QMF), a query and report writing facility, which allows you to write applications through a callable interface. You can create applications to perform a variety of tasks, such as data entry, query building, administration aids, and report analysis. For more information, refer to: www.ibm.com/software/data/qmf/.

- z/OS Java support

  The Java language supports the Java Native Interface (JNI) for making calls to and from C/C++. These calls do not use ILC support but rather the Java-defined JNI, which is supported by both compiled and interpreted Java code. Calls to C or C++ do not distinguish between these two.

Additional features of z/OS XL C/C++

Feature	Description
`long long` Data Type	z/OS XL C/C++ supports `long long` as a native data type when the compiler option LANGLVL(LONGLONG) is turned on. This option is turned on by default by the compiler option LANGLVL(EXTENDED). As of z/OS V1R7, the XL C compiler supports long long as a native data type (according to the ISO/IEC 9899:1999 standard), when the LANGLVL(STDC99) option or LANGLVL(EXTC99) option is in effect.
Multibyte Character Support	z/OS XL C/C++ supports multibyte characters for those national languages such as Japanese whose characters cannot be represented by a single byte.
Wide Character Support	Multibyte characters can be normalized by z/OS XL C library functions and encoded in units of one length. These normalized characters are called wide characters. Conversions between multibyte and wide characters can be performed by string conversion functions such as `wcstombs()`, `mbstowcs()`, `wcsrtombs()`, and `mbsrtowcs()`, as well as the family of wide-character I/O functions. Wide-character data can be represented by the `wchar_t` data type.

Feature	Description
Extended Precision Floating-Point Numbers	z/OS XL C/C++ provides three IBM z/Architecture® floating-point number data types: single precision (32 bits), declared as `float`; double precision (64 bits), declared as `double`; and extended precision (128 bits), declared as `long double`. Extended precision floating-point numbers give greater accuracy to mathematical calculations. z/OS XL C/C++ also supports IEEE 754 floating-point representation (base-2 or binary floating-point formats). By default, `float`, `double`, and `long double` values are represented in z/Architecture floating-point formats (base-16 floating-point formats). However, the IEEE 754 floating-point representation is used if you specify the FLOAT(IEEE) compiler option. For details on this support, see "FLOAT" on page 114. As of z/OS V1R9, XL C/C++ also supports IEEE 754 decimal floating-point representation (base-10 floating-point formats), with the types `_Decimal32`, `_Decimal64`, and `_Decimal128`, if you specify the DFP compiler option. For details on this support, see "DFP | NODFP" on page 100.
Command Line Redirection	You can redirect the standard streams `stdin`, `stderr`, and `stdout` from the command line or when calling programs using the `system()` function.
National Language Support	z/OS XL C/C++ provides message text in either American English or Japanese. You can dynamically switch between these two languages.
Coded Character Set (Code Page) Support	The z/OS XL C/C++ compiler can compile C/C++ source written in different EBCDIC code pages. In addition, the iconv utility converts data or source from one code page to another.
Selected Built-in Library Functions	For selected library functions, the compiler generates an instruction sequence directly into the object code during optimization to improve execution performance. String and character functions are examples of these built-in functions. No actual calls to the library are generated when built-in functions are used.
Multi-threading	Threads are efficient in applications that allow them to take advantage of any underlying parallelism available in the host environment. This underlying parallelism in the host can be exploited either by forking a process and creating a new address space, or by using multiple threads within a single process. For more information, refer to "Using Threads in z/OS UNIX Applications" in *z/OS XL C/C++ Programming Guide*.
Packed Structures and Unions	z/OS XL C provides support for packed structures and unions. Structures and unions may be packed to reduce the storage requirements of a z/OS XL C program or to define structures that are laid out according to COBOL or PL/I structure alignment rules.
Fixed-point (Packed) Decimal Data	z/OS XL C supports fixed-point (packed) decimal as a native data type for use in business applications. The packed data type is similar to the COBOL data type `COMP-3` or the PL/I data type `FIXED DEC`, with up to 31 digits of precision.
Long Name Support	For portability, external names can be mixed case and up to 32 K - 1 characters in length. For C++, the limit applies to the mangled version of the name.
System Calls	You can call commands or executable modules using the `system()` function under z/OS, z/OS UNIX System Services, and TSO. You can also use the `system()` function to call EXECs on z/OS and TSO, or shell scripts using z/OS UNIX System Services.
Exploitation of Hardware	Use the ARCHITECTURE compiler option to select the minimum level of machine architecture on which your program will run. Note that certain features provided by the compiler require a minimum architecture level. For more information, refer to "ARCHITECTURE" on page 69. Use the TUNE compiler option to optimize your application for a specific machine architecture within the constraints imposed by the ARCHITECTURE option. The TUNE level must not be lower than the setting in the ARCHITECTURE option. For more information, refer to "TUNE" on page 254.

Feature	Description
Built-in Functions for Floating-Point and Other Hardware Instructions	Use built-in functions for floating-point and other hardware instructions that are otherwise inaccessible to XL C/C++ programs. For more information, see the built-in functions described in *z/OS XL C/C++ Programming Guide*.

Chapter 2. z/OS XL C example

This information describes the basic steps to compile, bind, and run a C example program under z/OS batch, TSO, or the z/OS shell.

If you have not yet compiled a C program, some concepts in this information may be unfamiliar. Refer to Chapter 7, “Compiling,” on page 335, Chapter 9, “Binding z/OS XL C/C++ programs,” on page 401, and Chapter 11, “Running a C or C++ application,” on page 455 for a detailed description on compiling, binding, and running a C program.

This information describes steps to bind a C example program. It does not describe the prelink and link steps. If you are using the prelinker, see Appendix A, “Prelinking and linking z/OS XL C/C++ programs,” on page 601.

The example program that this information describes is shipped with the z/OS XL C compiler in the data set CBC.SCCNSAM.

Example of a C program

The following example shows a simple z/OS XL C program that converts temperatures in Celsius to Fahrenheit. You can either enter the temperatures on the command line or let the program prompt you for the temperature.

In this example, the `main` program calls the function `convert()` to convert the Celsius temperature to a Fahrenheit temperature and to print the result.

CCNUAAM

```
#include <stdio.h>                  1

#include "ccnuaan.h"                2

void convert(double);               3

int main(int argc, char **argv)     4
{
    double c_temp;                  5

    if (argc == 1) {  /* get Celsius value from stdin */

       printf("Enter Celsius temperature: \n");   6

       if (scanf("%f", &c_temp) != 1) {
          printf("You must enter a valid temperature\n");
       }
       else {
          convert(c_temp);          7
       }
    }
```

Figure 1. Celsius-to-Fahrenheit conversion (Part 1 of 2)

```
    else { /* convert the command-line arguments to Fahrenheit */
       int i;

       for (i = 1; i < argc; ++i) {
           if (sscanf(argv[i], "%f", &c_temp) != 1)
              printf("%s is not a valid temperature\n",argv[i]);
           else
              convert(c_temp);    7
       }
    }
     return 0;
}

void convert(double c_temp) {     8
   double f_temp = (c_temp * CONV + OFFSET);
   printf("%5.2f Celsius is %5.2f Fahrenheit\n",c_temp, f_temp);
}
```

Figure 1. Celsius-to-Fahrenheit conversion (Part 2 of 2)

CCNUAAN

```
/***************************************************************
 *  User include file:  ccnuaan.h                            *   9
 ***************************************************************/

#define CONV   (9./5.)
#define OFFSET 32
```

Figure 2. User #include file for the conversion program

1 The `#include` preprocessor directive names the `stdio.h` system file. `stdio.h` contains declarations of standard library functions, such as the `printf()` function used by this program.

The compiler searches the system libraries for the `stdio.h` file. For more information about searches for include files, see "Search sequences for include files" on page 369.

2 The `#include` preprocessor directive names the CCNUAAN user file. CCNUAAN defines constants that are used by the program.

The compiler searches the user libraries for the file CCNUAAN.

If the compiler cannot locate the file in the user libraries, it searches the system libraries.

3 This is a function prototype declaration. This statement declares `convert()` as an external function having one parameter.

4 The program begins execution at this entry point.

5 This is the automatic (local) data definition to `main()`.

6 This `printf` statement is a call to a library function that allows you to format your output and print it on the standard output device. The `printf()` function is declared in the standard I/O header file `stdio.h` included at the beginning of the program.

7 This statement contains a call to the `convert()` function, which was declared earlier in the program as receiving one double value, and not returning a value.

8 This is a function definition. In this example, the declaration for this function appears immediately before the definition of the `main()` function. The code for the function is in the same file as the code for the `main()` function.

9 This is the user include file containing the definitions for CONV and OFFSET.

If you need more details on the constructs of the z/OS XL C language, see *z/OS XL C/C++ Language Reference* and *z/OS XL C/C++ Run-Time Library Reference*.

Compiling, binding, and running the z/OS XL C example

You can compile, bind, and run z/OS XL C programs under z/OS batch, TSO, or the z/OS shell. You cannot run the IPA Link step under TSO. For more information, see Chapter 7, "Compiling," on page 335, Chapter 9, "Binding z/OS XL C/C++ programs," on page 401, and Chapter 11, "Running a C or C++ application," on page 455.

This document uses the term *user prefix* to refer to the high-level qualifier of your data sets. For example, in `PETE.TESTHDR.H`, the user prefix is `PETE`. Under TSO, your prefix is set or queried by the PROFILE command.

Note: The z/OS XL C compiler does not support TSO PROFILE NOPREFIX.

Under z/OS batch

Copy the IBM-supplied sample program and header file into your data set. For example, if your user prefix is `PETE`, store the sample program (CCNUAAM) in `PETE.TEST.C(CTOF)` and the header file in `PETE.TESTHDR.H(CCNUAAN)`. You can use the IBM-supplied cataloged procedure EDCCBG to compile, bind, and run the example program as follows:

```
//DOCLG       EXEC   PROC=EDCCBG,INFILE='PETE.TEST.C(CTOF)',
//            CPARM='LSEARCH(''''PETE.TESTHDR.+'''')'
//GO.SYSIN    DD DATA,DLM=@@
19
@@
```

Figure 3. JCL to compile, bind, and run the example program using the EDCCBG procedure

In Figure 3, the LSEARCH statement describes where to find the user include files. The system header files will be searched in the data sets specified on the SEARCH compiler option, which defaults to CEE.SCEEH.+. The GO.SYSIN statement indicates that the input that follows it is given for the execution of the program.

XPLINK under z/OS batch

Figure 4 shows the JCL for building with XPLINK.

```
//DOCLG       EXEC   PROC=EDCXCBG,INFILE='PETE.TEST.C(CTOF)',
//            CPARM='LSEARCH(''''PETE.TESTHDR.+'''')'
//GO.SYSIN    DD DATA,DLM=@@
19
@@
```

Figure 4. JCL to build with XPLINK

Non-XPLINK and XPLINK under TSO

Copy the IBM-supplied sample program and header file into your data set. For example, if your user prefix is `PETE`, store the sample z/OS XL C program (CCNUAAM) in `PETE.TEST.C(CTOF)` and the header file in `PETE.TESTHDR.H(CCNUAAN)`.

Steps for compiling, binding, and running the example program using TSO commands

Before you begin: Ensure that the Language Environment run-time libraries SCEERUN and SCEERUN2, and the z/OS XL C compiler are in the STEPLIB, LPALST, or LNKLST concatenation.

Perform the following steps to compile, bind, and run the example program using TSO commands:

1. Compile the z/OS XL C source. You can use the REXX EXEC CC to invoke the z/OS XL C compiler under TSO as follows:

   ```
   %CC TEST.C(CTOF) (LSEARCH(TESTHDR.H)

   -- or, for XPLINK --

   %CC TEST.C(CTOF) (LSEARCH(TESTHDR.H) XPLINK
   ```

   The REXX EXEC CC compiles CTOF with the default compiler options and stores the resulting object module in `PETE.TEST.C.OBJ(CTOF)`.

   The compiler searches for user header files in the PDS `PETE.TESTHDR.H`, which you specified at compile time by the LSEARCH option. The system header files are searched in the data sets specified with the SEARCH compiler option, which defaults to CEE.SCEEH.+.

   For more information see "Compiling under TSO" on page 346.

2. Perform a bind:

   ```
   CXXBIND OBJ(TEST.C.OBJ(CTOF)) LOAD(TEST.C.LOAD(CTOF))

   -- or, for XPLINK --

   CXXBIND OBJ(TEST.C.OBJ(CTOF)) LOAD(TEST.C.LOAD(CTOF)) XPLINK
   ```

   CXXBIND binds the object module `PETE.TEST.C.OBJ(CTOF)` to create an executable module CTOF in the PDSE `PETE.TEST.C.LOAD`, with the default bind options. See Chapter 9, "Binding z/OS XL C/C++ programs," on page 401 for more information.

3. Run the program:

   ```
   CALL TEST.C.LOAD(CTOF)
   ```

   Example: When a message appears asking you to enter a Celsius temperature, enter, for example, 25.

   Result: The load module displays the following output: `25.00 Celsius is 77.00 Fahrenheit`

   CALL runs CTOF from `PETE.TEST.C.LOAD` with the default run-time options in effect. See Chapter 11, "Running a C or C++ application," on page 455 for more information.

Non-XPLINK and XPLINK under the z/OS UNIX System Services shell

Steps for compiling, binding, and running the example program using UNIX commands

Before you begin: Put the source in the z/OS UNIX file system and ensure that the Language Environment run-time libraries SCEERUN and SCEERUN2, and the z/OS XL C compiler are in the STEPLIB, LPALST, or LNKLST concatenation.

Perform the following steps to compile, bind, and run the example program using z/OS UNIX commands:

1. From the z/OS shell, type the following:

   ```
   cp "//'cbc.sccnsam(ccnuaam)'" ccnuaam.c
   cp "//'cbc.sccnsam(ccnuaan)'" ccnuaan.h
   ```

2. Compile and bind:

   ```
   c89 -o ctof ccnuaam.c

   -- or, for XPLINK --

   c89 -o ctof -Wc,xplink -Wl,xplink ccnuaam.c
   ```

3. Run the program:

   ```
   ./ctof
   ```

   Example: When a message appears asking you to enter a Celsius temperature, enter, for example, 25.

   Result: The load module displays the following output: `25.00 Celsius is 77.00 Fahrenheit`

Chapter 3. z/OS XL C++ examples

This information describes the basic steps to compile, bind, and run z/OS XL C++ example programs under z/OS batch, TSO, or the z/OS shell.

If you have not yet compiled a C++ program, some information in this information may be unfamiliar. Refer to Chapter 7, “Compiling,” on page 335, Chapter 9, “Binding z/OS XL C/C++ programs,” on page 401, and Chapter 11, “Running a C or C++ application,” on page 455 for a detailed description on compiling, binding, and running a C++ program.

The example programs that this information describes are shipped with the z/OS XL C++ compiler. Example programs with the names `CCNUxxx` are shipped in the data set CCN.SCCNSAM. Example programs with the names `CLB3xxxx` are shipped in the z/OS UNIX System Services file system in `/usr/lpp/cbclib/sample`.

Example of a C++ program

The following example shows a z/OS XL C++ program that prompts you to enter a birth date. The program output is the corresponding biorhythm chart.

The program is written using an object-oriented method. A class that is called `BioRhythm` is defined. It contains an object `birthDate` of class `BirthDate`, which is derived from the class `Date`. An object that is called `bio` of the class `BioRhythm` is declared.

The example contains two files. File CCNUBRH contains the classes that are used in the main program. File CCNUBRC contains the remaining source code. The example files CCNUBRC and CCNUBRH are shipped with the z/OS XL C++ compiler in data sets CBC.SCCNSAM(CCNUBRC) and CBC.SCCNSAM(CCNUBRH).

If you need more details on the constructs of the z/OS XL C++ language, see *z/OS XL C/C++ Language Reference* or *z/OS XL C/C++ Run-Time Library Reference*.

© Copyright IBM Corp. 1998, 2009

CCNUBRH

```
//
// Sample Program: Biorhythm
// Description   : Calculates biorhythm based on the current
//                 system date and birth date entered
//
// File 1 of 2-other file is CCNUBRC

class Date {
  public:
    Date();
    int DaysSince(const char *date);

  protected:
    int curYear, curDay;
    static const int dateLen = 10;
    static const int numMonths = 12;
    static const int numDays[];
};


class BirthDate : public Date {
  public:
    BirthDate();
    BirthDate(const char *birthText);
    int DaysOld() { return(DaysSince(text)); }

  private:
    char text[Date::dateLen+1];
};
```

Figure 5. Header file for the biorhythm example (Part 1 of 2)

```
class BioRhythm {
  public:
    BioRhythm(char *birthText) : birthDate(birthText) {
      age = birthDate.DaysOld();
    }
    BioRhythm() : birthDate() {
      age = birthDate.DaysOld();
    }
    ~BioRhythm() {}

    int AgeInDays() {
      return(age);
    }
    double Physical()  {
      return(Cycle(pCycle));
    }
    double Emotional() {
      return(Cycle(eCycle));
    }
    double Intellectual() {
      return(Cycle(iCycle));
    }
    int ok() {
      return(age >= 0);
    }

  private:
    int age;
    double Cycle(int phase) {
      return(sin(fmod((double)age, (double)phase) / phase * M_2PI));
    }
    BirthDate birthDate;
    static const int pCycle=23;     // Physical cycle - 23 days
    static const int eCycle=28;     // Emotional cycle - 28 days
    static const int iCycle=33;     // Intellectual cycle - 33 days
};
```

Figure 5. Header file for the biorhythm example (Part 2 of 2)

CCNUBRC

```
//
// Sample Program: Biorhythm
// Description   : Calculates biorhythm based on the current
//                 system date and birth date entered
//
// File 2 of 2-other file is CCNUBRH

#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <iostream>
#include <iomanip>

#include "ccnubrh.h"  //BioRhythm class and Date class
using namespace std;
static ostream& operator << (ostream&, BioRhythm&);


int main(void) {

  BioRhythm bio;
  int code;

  if (!bio.ok()) {
    cerr << "Error in birthdate specification - format is yyyy/mm/dd";
    code = 8;
  }
  else {
    cout << bio;  // write out birthdate for bio
    code = 0;
  }
  return(code);
}

const int Date::dateLen ;
const int Date::numMonths;
const int Date::numDays[Date::numMonths] = {
  31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    };

const int BioRhythm::pCycle;
const int BioRhythm::eCycle;
const int BioRhythm::iCycle;

ostream& operator<<(ostream& os, BioRhythm& bio) {
  os << "Total Days  : " << bio.AgeInDays() << "\n";
  os << "Physical    : " << bio.Physical() << "\n";
  os << "Emotional   : " << bio.Emotional() << "\n";
  os << "Intellectual: " << bio.Intellectual() << "\n";

return(os);
}
```

Figure 6. z/OS XL C++ Biorhythm example program (Part 1 of 3)

```
 Date::Date() {
  time_t lTime;
  struct tm *newTime;

  time(&lTime);
  newTime = localtime(&lTime);
  cout << "local time is " << asctime(newTime) << endl;

  curYear = newTime->tm_year + 1900;
  curDay  = newTime->tm_yday + 1;

}

BirthDate::BirthDate(const char *birthText) {
  strcpy(text, birthText);
}

BirthDate::BirthDate() {
  cout << "Please enter your birthdate in the form yyyy/mm/dd\n";
  cin >> setw(dateLen+1) >> text;
}

Date::DaysSince(const char *text) {

  int year, month, day, totDays, delim;
  int daysInYear = 0;
  int i;
  int leap = 0;

  int rc = sscanf(text, "%4d%c%2d%c%2d",
                 &year, &delim, &month, &delim, &day);
  --month;
  if (rc != 5 || year  < 0 || year  > 9999 ||
                month < 0 || month >   11 ||
                day   < 1 || day   >   31 ||
               (day   > numDays[month]&& month != 1)) {
    return(-1);
  }
  if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
     leap = 1;

  if (month == 1 && day > numDays[month]) {
    if (day > 29)
      return(-1);
    else if (!leap)
      return (-1);
  }
```

Figure 6. z/OS XL C++ Biorhythm example program (Part 2 of 3)

```
   for (i=0;i<month;++i) {
    daysInYear += numDays[i];
  }
  daysInYear += day;

  // correct for leap year
  if (leap == 1 &&
     (month > 1 || (month == 1 && day == 29)))
     ++daysInYear;

  totDays = (curDay - daysInYear) + (curYear - year)*365;

  // now, correct for leap year
  for (i=year+1; i < curYear; ++i) {
    if ((i % 4 == 0 && i % 100 != 0) || i % 400 == 0) {
      ++totDays;
    }
  }
  return(totDays);
}
```

Figure 6. z/OS XL C++ Biorhythm example program (Part 3 of 3)

Compiling, binding, and running the z/OS XL C++ example

You can compile, bind, and run z/OS XL C++ programs under z/OS batch, TSO, or the z/OS shell. You cannot run the IPA Link step under TSO. For more information, see Chapter 7, “Compiling,” on page 335, Chapter 9, “Binding z/OS XL C/C++ programs,” on page 401, and Chapter 11, “Running a C or C++ application,” on page 455.

This document uses the term *user prefix* to refer to the high-level qualifier of your data sets. For example, in CEE.SCEERUN, the user prefix is CEE.

Note: The z/OS XL C++ compiler does not support TSO PROFILE NOPREFIX.

Under z/OS batch

Copy the IBM-supplied sample program and header file into your data set. For example, if your user prefix is PETE, store the sample program CCNUBRC in PETE.TEST.C(CCNUBRC), and the header file CCNUBRH in PETE.TESTHDR.H(CCNUBRH). You can use the IBM-supplied cataloged procedure CBCCBG to compile, bind, and run the source code as follows:

```
//*
//* COMPILE, BIND AND RUN
//*
//DOCLG    EXEC  CBCCBG,
//         INFILE='PETE.TEST.C(CCNUBRC)',
//         CPARM='OPTFILE(DD:CCOPT)'
//COMPILE.CCOPT DD *
          LSEARCH('PETE.TESTHDR.H')
          SEARCH('CEE.SCEEH.+','CBC.SCLBH.+')
/*
//* ENTER A DATE IN THE FORM YYYY/MM/DD
//GO.SYSIN DD *
  1997/10/19
/*
```

Figure 7. JCL to compile, bind, and run the example program using the CBCCBG procedure

In Figure 7 on page 32, the LSEARCH statement describes where to find the user include files, and the SEARCH statement describes where to find the system include files. The GO.SYSIN statement indicates that the input that follows it is given for the execution of the program.

XPLINK under z/OS batch

The following example shows how to compile, bind, and run a program with XPLINK using the CBCXCBG procedure:

```
//*
//* COMPILE, BIND AND RUN
//*
//DOCLG    EXEC  CBCXCBG,
//         INFILE='PETE.TEST.C(CCNUBRC)',
//         CPARM='OPTFILE(DD:CCOPT)'
//COMPILE.CCOPT DD *
         LSEARCH('PETE.TESTHDR.H')
         SEARCH('CEE.SCEEH.+','CBC.SCLBH.+')
/*
//* ENTER A DATE IN THE FORM YYYY/MM/DD
//GO.SYSIN DD *
  1997/10/19
/*
```

Figure 8. JCL to compile, bind, and run the example program with XPLINK using the CBCXCBG procedure

For more information on compiling, binding, and running, see Chapter 7, “Compiling,” on page 335, Chapter 9, “Binding z/OS XL C/C++ programs,” on page 401, and Chapter 11, “Running a C or C++ application,” on page 455.

Non-XPLINK and XPLINK under TSO

Copy the IBM-supplied sample program and header file into your data set. For example, if your user prefix is PETE, store the sample program CCNUBRC in PETE.TEST.C(CCNUBRC), and the header file CCNUBRH in PETE.TESTHDR.H(CCNUBRH).

Steps for compiling, binding, and running the C++ example program using TSO commands

Before you begin: Ensure that the Language Environment run-time libraries SCEERUN and SCEERUN2, the z/OS class library DLLs, and the z/OS XL C++ compiler are in the STEPLIB, dynamic LPA, or Link List concatenation.

Perform the following steps to compile, bind, and run the example program using TSO commands:

1. Compile the z/OS XL C++ source. You can use the REXX EXEC CXX to invoke the z/OS XL C++ compiler under TSO as follows:

   ```
   CXX 'PETE.TEST.C(CCNUBRC)' ( LSEARCH('PETE.TESTHDR.H') OBJECT(BIO.TEXT)
           SEARCH('CEE.SCEEH.+','CBC.SCLBH.+')

   -- or, for XPLINK --

    CXX 'PETE.TEST.C(CCNUBRC)' ( LSEARCH('PETE.TESTHDR.H') OBJECT(BIO.TEXT)
    SEARCH('CEE.SCEEH.+','CBC.SCLBH.+') XPLINK
   ```

   CXX compiles CCNUBRC with the specified compiler options and stores the resulting object module in PETE.BIO.TEXT(CCNUBRC).

   The compiler searches for user header files in the PDS PETE.TESTHDR.H , which you specified at compile time with the LSEARCH option. The compiler searches for system header files in the PDS CEE.SCEEH.+ and PDS CBC.SCLBH.+, which you specified at compile time with the SEARCH option.

For more information see "Compiling under TSO" on page 346.

---

2. Bind:

```
CXXBIND OBJ(BIO.TEXT(CCNUBRC)) LOAD(BIO.LOAD(BIORUN))

-- or, for XPLINK --

 CCXXBIND OBJ(BIO.TEXT(CCNUBRC)) LOAD(BIO.LOAD(BIORUN)) XPLINK
```

CXXBIND binds the object module `PETE.BIO.TEXT(CCNUBRC)`, and creates an executable module `BIORUN` in `PETE.BIO.LOAD` PDSE with the default bind options.

Note: To avoid a bind error, the data set `PETE.BIO.LOAD` must be a PDSE, not a PDS.

For more information see Chapter 9, "Binding z/OS XL C/C++ programs," on page 401.

---

3. Run the program:

```
CALL BIO.LOAD(BIORUN)
```

Example: When you are asked to enter your birthdate, enter, for example, 1999/01/03.

Result: The following information displays:

```
Total Days  : 1116
Physical    : -0.136167
Emotional   : -0.781831
Intellectual: -0.909632
```

CALL runs the module `BIORUN` from the PDSE `PETE.BIO.LOAD` with the default run-time options.

For more information see "Running an application under TSO" on page 458.

Non-XPLINK and XPLINK under the z/OS UNIX shell

Steps for compiling, binding, and running the C++ example program using UNIX commands

Before you begin: Put the source in the z/OS UNIX file system. From the z/OS shell type:

```
cp "//'cbc.sccnsam(ccnubrc)'" ccnubrc.C
cp "//'cbc.sccnsam(ccnubrh)'" ccnubrh.h
```

This example uses the current working directory so make sure that you are in the directory you want to use. Use the **pwd** command to display the current working directory, the **mkdir** command to create a new directory, and the **cd** command to change directories.

Ensure that the Language Environment run-time libraries SCEERUN and SCEERUN2, the z/OS class library DLLs, and the z/OS XL C++ compiler are in the STEPLIB, dynamic LPA, or Link List concatenation.

Perform the following steps to compile, bind, and run the example program using z/OS UNIX commands:

1. Compile and bind:

   ```
   c++ -o bio ccnubrc.C

   -- or, for XPLINK --

   c++ -o bio -Wc,xplink -Wl,xplink ccnubrc.C
   ```

   Note: You can use **c++** to compile source that is stored in a data set.

2. Run the program:

   ```
   ./bio
   ```

   Example: When you are asked to enter your birthdate, enter, for example, 1999/01/03.

   Result: The following information displays:

   ```
   Total Days  : 1116
   Physical    : -0.136167
   Emotional   : -0.781831
   Intellectual: -0.909632
   ```

Example of a C++ template program

A *class template* or generic class is a blueprint that describes how members of a set of related classes are constructed.

The following example shows a simple z/OS XL C++ program that uses templates to perform simple operations on linked lists. It resides in the z/OS UNIX System Services file system in the directory `/usr/lpp/cbclib/sample`. The `main` program, CLB3ATMP.CPP (see "CLB3ATMP.CPP" on page 36), uses three header files that are from the Standard C++ Library: `list`, `string`, and `iostream`. It has one class template: `list`.

CLB3ATMP.CPP

```
#include <list>
#include <string>
#include <iostream>
using namespace std;

template <class Item> class IOList {
  public:
    IOList() : myList() {}
    void write();
    void read(const char *msg);
    void append(Item item) {
      myList.push_back(item);
    }
  private:
    list<Item> myList;
};

template <class Item> void IOList<Item>::write() {
  ostream_iterator<Item> oi(cout, " ");
  copy(myList.begin(), myList.end(), oi);
  cout << '\n';
}

template <class Item> void IOList<Item>::read(const char *msg) {
  Item item;
  cout << msg << endl;
  istream_iterator<Item> ii(cin);
  copy(ii, istream_iterator<Item>(), back_insert_iterator<list<Item> >(myList));
}

int main() {
  IOList<string> stringList;
  IOList<int>    intList;

  char line1[] = "This program will read in a list of ";
  char line2[] = "strings, integers and real numbers";
  char line3[] = "and then print them out";

  stringList.append(line1);
  stringList.append(line2);
  stringList.append(line3);
  stringList.write();
  intList.read("Enter some integers (/* to terminate)");
  intList.write();

  string name1 = "Bloe, Joe";
  string name2 = "Jackson, Joseph";


  if (name1 < name2)
    cout << name1 << " comes before " << name2;
  else
    cout << name2 << " comes before " << name1;
  cout << endl;
```

Figure 9. z/OS XL C++ template program (Part 1 of 2)

```
  int num1 = 23;
  int num2 = 28;
  if (num1 < num2)
    cout << num1 << " comes before " << num2;
  else
    cout << num2 << "comes before " << num1;
  cout << endl;

  return(0);
}
```

Figure 9. z/OS XL C++ template program (Part 2 of 2)

Compiling, binding, and running the C++ template example

This information describes the commands to compile, bind and run the template example under z/OS batch, TSO, and the z/OS shell.

Under z/OS batch

Steps for compiling, binding, and running the C++ template example program under z/OS batch

Before you begin: Ensure that the Language Environment run-time libraries SCEERUN and SCEERUN2, and the z/OS XL C++ compiler are in STEPLIB, LPALST, or the LNKLST concatenation.

Perform the following step to compile, bind, and run the C++ template example program under z/OS batch:

- Change `<userhlq>` to your own user prefix in the example JCL.

CCNUNCL

```
//Jobcard info
//PROC JCLLIB ORDER=(CBC.SCCNPRC,
//   CEE.SCEEPROC)
//*
//* Compile MAIN program,creating an object deck
//*
//MAINCC EXEC CBCC,
//       OUTFILE='<userhlq>.SAMPLE.OBJ(CLB3ATMP),DISP=SHR ',
//       CPARM='XPLINK,OPTF(DD:COPTS)'
//SYSIN DD PATH='/usr/lpp/cbclib/sample/clb3atmp/clb3atmp.cpp'
//COPTS DD *
 SEARCH('CEE.SCEEH.+', 'CBC.SCLBH.+')
/*
//*
//* Bind the program
//*
//BIND EXEC CBCXB,
//     INFILE='<userhlq>.SAMPLE.OBJ(CLB3ATMP)',
//     OUTFILE='<userhlq>.SAMPLE.LOAD(CLB3ATMP),DISP=SHR'
//*
//* Run the program
//*
//GO   EXEC CBCXG,
//     INFILE='<userhlq>.SAMPLE.LOAD',
//     GOPGM=CLB3ATMP
//GO.SYSIN DD *
 1 2 5 3 7 8 3 2 10 11
/*
```

Figure 10. JCL to compile, bind and run the template example

Under TSO

Steps for compiling, running, and binding the C++ template example program using TSO commands

Before you begin: Ensure that the Language Environment run-time libraries SCEERUN and SCEERUN2, the z/OS Class Library DLLs, and the z/OS XL C++ compiler are in STEPLIB, LPALST, or the LNKLST concatenation.

Perform the following steps to compile, bind, and run the C++ template example program using TSO commands:

1. Compile the source files:

   a.
   ```
   cxx /usr/lpp/cbclib/sample/clb3atmp.cpp (lsearch(/usr/lpp/cbclib/
   sample) search('cee.sceeh.+','cbc.sclbh.+')
   obj(sample.obj(clb3atmp)) tempinc(//tempinc)
   ```
   This step compiles `CLB3ATMP` with the default compiler options, and stores the object module in *userhlq*.SAMPLE.OBJ(CLB3ATMP), where *userhlq* is your user prefix. The template instantiation files are written to the PDS *userhlq*.TEMPINC.

   b.
   ```
   cxx TEMPINC (lsearch(/usr/lpp/cbclib/sample)
   search('cee.sceeh.+','cbc.sclbh.+')
   ```
   This step compiles the PDS TEMPINC and creates the corresponding objects in the PDS *userhlq*.TEMPINC.OBJ.

   See "Compiling under TSO" on page 346 for more information.

2. Create a library from the PDS *userhlq*.TEMPINC.OBJ:
   ```
   C370LIB DIR LIB(TEMPINC.OBJ)
   ```
   For more information see "Creating an object library under TSO" on page 482

3. Bind the program:
   ```
   CXXBIND OBJ(SAMPLE.OBJ(CLB3ATMP)) LIB(TEMPINC.OBJ) LOAD(SAMPLE.LOAD(CLB3ATMP))
   ```
   This step binds the *userhlq*.SAMPLE.OBJ(CLB3ATMP) text deck using the *userhlq*.TEMPINC.OBJ library and the default bind options. It also creates the executable module *userhlq*.SAMPLE.LOAD(CLB3ATMP).

   Note: To avoid a binder error, the data set *userhlq*.SAMPLE.LOAD must be a PDSE.
   For more information see "Binding under TSO using CXXBIND" on page 424.

4. Run the program:
   ```
   CALL SAMPLE.LOAD(CLB3ATMP)
   ```
   This step executes the module *userhlq*.SAMPLE.LOAD(CLB3ATMP) using the default run-time options. For more information see "Running an application under TSO" on page 458.

Under the z/OS UNIX shell

Steps for compiling, binding, and running the C++ template example program using UNIX commands

Before you begin: Ensure that the Language Environment run-time libraries SCEERUN and SCEERUN2, and the z/OS XL C++ compiler are in STEPLIB, LPALST, or the LNKLST concatenation.

Perform the following steps to compile, run, and bind the template example program under the z/OS shell:

1. Copy sample files to your own directory, as follows:

   ```
   cp /usr/lpp/cbclib/sample/clb3atmp/* your_dir/.
   ```

2. Then, to compile and bind:

   ```
     c++ -+ -o clb3atmp clb3atmp.cpp
   ```

   This command compiles `clb3atmp.cpp` and then compiles the `./tempinc` directory (which is created if it does not already exist). It then binds using all the objects in the `./tempinc` directory. An archive file, or C370LIB object library is not created.

3. Run the program:

   ```
    ./clb3atmp
   ```

Chapter 4. Compiler Options

This information describes the options that you can use to alter the compilation of your program.

Specifying compiler options

You can override your installation default options when you compile your z/OS XL C/C++ program, by specifying an option in one of the following ways:

- In the option list when you invoke the IBM-supplied REXX EXECs.
- In the CPARM parameter of the IBM-supplied cataloged procedures, when you are compiling under z/OS batch.

  See Chapter 7, "Compiling," on page 335, and Chapter 12, "Cataloged procedures and REXX EXECs," on page 465 for details.
- In your own JCL procedure, by passing a parameter string to the compiler.
- In an options file. See "OPTFILE | NOOPTFILE" on page 194 for details.
- For z/OS XL C, in a **#pragma options** preprocessor directive within your source file. See "Specifying z/OS XL C compiler options using #pragma options" on page 45 for details.

  Compiler options that you specify on the command line or in the CPARM parameter of IBM-supplied cataloged procedures can override compiler options that are used in **#pragma options**. The exception is CSECT, where the **#pragma csect** directive takes precedence.
- On the command line of the c89 utility, by using the **-Wc**, **-Wl**, and **-Wl,I** options to pass options to the compiler.
- On the command line of the xlc utility, by using the **-q** option or the **-Wc** and **-Wl,I** options to pass options to the compiler.

The following compiler options are inserted in your object module to indicate their status:

AGGRCOPY	
ALIAS	(C compile only)
ANSIALIAS	
ARCHITECTURE	
ARGPARSE	
ASCII	
ASSERT(RESTRICT)	
BITFIELD	
CHARS	
COMPACT	
COMPRESS	
CONVLIT	
CSECT	
CVFT	(C++ compile only)
DEBUG	
DFP	
DLL	
EXECOPS	
EXPORTALL	
FLOAT	
GOFF	

© Copyright IBM Corp. 1998, 2009

GONUMBER	
HGPR	
HOT	
IGNERRNO	
ILP32	
INITAUTO	
INLINE	
IPA	
LANGLVL	
LIBANSI	
LOCALE	
LONGNAME	
LP64	
MAXMEM	
NAMEMANGLING	(C++ compile only)
OBJECTMODEL	(C++ compile only)
OPTIMIZE	
PREFETCH	
PLIST	
REDIR	
RENT	(C compile only)
ROCONST	
ROSTRING	
ROUND	
RTTI	(C++ compile only)
RTCHECK	
SERVICE	
SPILL	
START	
STRICT	
STRICT_INDUCTION	
TARGET	
TEMPLATERECOMPILE	(C++ compile only)
TEMPLATEREGISTRY	(C++ compile only)
TMPLPARSE	(C++ compile only)
TUNE	
UNROLL	
UPCONV	(C compile only)
WSIZEOF	
XPLINK	

IPA considerations

The following sections explain what you should be aware of if you request Interprocedural Analysis (IPA) through the IPA option. Before you use the IPA compiler option, refer to an overview of IPA in *z/OS XL C/C++ Programming Guide*.

Applicability of compiler options under IPA

You should keep the following points in mind when specifying compiler options for the IPA Compile or IPA Link step:

- Many compiler options do not have any special effect on IPA. For example, the PPONLY option processes source code, then terminates processing prior to IPA Compile step analysis.
- Compiler options for IPA(OBJONLY) compiles are the same as for NOIPA compiles.

- `#pragma` directives in your source code, and compiler options you specify for the IPA Compile step, may conflict across compilation units.

  `#pragma` directives in your source code, and compiler options you specify for the IPA Compile step, may conflict with options you specify for the IPA Link step.

  IPA will detect such conflicts and apply default resolutions with appropriate diagnostic messages. The Compiler Options Map section of the IPA Link step listing displays the conflicts and their resolutions.

  To avoid problems, use the same options and suboptions on the IPA Compile and IPA Link steps. Also, if you use `#pragma` directives in your source code, specify the corresponding options for the IPA Link step.
- If you specify a compiler option that is irrelevant for a particular IPA step, IPA ignores it and does not issue a message.

In this information, the description of each compiler option includes its effect on IPA processing.

Interactions between compiler options and IPA suboptions

During IPA Compile step processing, IPA handles conflicts between IPA suboptions and certain compiler options that affect code generation.

If you specify a compiler option for the IPA Compile step, but do not specify the corresponding suboption of the IPA option, the compiler option may override the IPA suboption. Table 5 shows how the OPT, LIST, and GONUMBER compiler options interact with the OPT, LIST, and GONUMBER suboptions of the IPA option. The `xxxx` indicates the name of the option or suboption. `NOxxxx` indicates the corresponding negative option or suboption.

Table 5. Interactions between compiler options and IPA suboptions

Compiler Option	Corresponding IPA Suboption	Value used in IPA Object
no option specified	no suboption specified	NOxxxx
no option specified	NOxxxx	NOxxxx
no option specified	xxxx	xxxx
NOxxxx	no option specified	NOxxxx
NOxxxx	NOxxxx	NOxxxx
NOxxxx	xxxx	xxxx
xxxx	no option specified	xxxx
xxxx	NOxxxx	xxxx [1]
xxxx	xxxx	xxxx

Note: [1]An informational message is produced that indicates that the suboption `NOxxxx` is promoted to `xxxx`.

IPA compiles versus compiles with IPA optimization

The IPA(OBJONLY) compilation is an intermediate level of optimization. This results in a modified regular compile, not an IPA Compile step. Unlike the IPA Compile step, no IPA information is written to the object file.

The object file may be used by an IPA Link step, a prelink (or link), or a bind. If it is used as input to an IPA Link step, IPA Link-time optimizations cannot be performed for this compilation unit because no IPA information is available.

Using special characters

Under TSO

When z/OS UNIX System Services file names contain the special characters

- blank
- backslash
- double quotation mark

A backslash (\) must precede these characters.

Note: Under TSO, a backslash \ must precede special characters in file names and options.

Two backslashes must precede suboptions that contain these special characters:

- left parenthesis (
- right parenthesis)
- comma
- backslash
- blank
- double quotation mark
- less than <
- greater than >

For example:

```
def(errno=\\(*__errno\\(\\)\\))
```

Under the z/OS UNIX System Services shell

The z/OS UNIX System Services shell imposes its own parsing rules. The c89 utility escapes special compiler and run-time characters as needed to invoke the compiler, so you need only be concerned with shell parsing rules.

While the c89 utility uses compiler options, which have parentheses, **xlc** uses the **-q** syntax, which does not use parentheses and is more convenient for shell invocation.

See Chapter 21, “c89 — Compiler invocation using host environment variables,” on page 535 and Chapter 24, “xlc — Compiler invocation using a customizable configuration file,” on page 577 for more information.

Under z/OS batch

When invoking the compiler directly (not through a cataloged procedure), you should type a single quotation mark (') within a string as two single quotation marks (''), as follows:

```
//COMPILE EXEC PGM=CCNDRVR,PARM='OPTFILE(''USERID.OPTS'')'
```

If you are using the same string to pass a parameter to a JCL PROC, use four single quotation marks (''''), as follows:

```
//COMPILE EXEC CBCC,CPARM='OPTFILE(''''USERID.OPTS'''')'
```

Special characters in z/OS UNIX System Services file names that are referenced in DD cards do not need a preceding backslash. For example, the special character blank in the file name `obj 1.o` does not need a preceding backslash when it is used in a `DD` card:

```
//SYSLIN DD PATH='u/user1/obj 1.o'
```

A backslash must precede special characters in z/OS UNIX System Services file names that are referenced in the PARM statement. The special characters are: backslash, blank, and double quotation mark. For example, a backslash precedes the special character blank in the file name `obj 1.o`, when used in the PARM keyword:

```
//STEP1 EXEC PGM=CCNDRVR,PARM='OBJ(/u/user1/obj\ 1.o)'
```

Specifying z/OS XL C compiler options using #pragma options

You can use the **#pragma options** preprocessor directive to override the default values for compiler options. Compiler options that are specified on the command line or in the CPARM parameter of the IBM-supplied cataloged procedures can override compiler options that are used in **#pragma options**. The exception is CSECT, where the **#pragma csect** directive takes precedence. For complete details on the **#pragma options** preprocessor directive, see *z/OS XL C/C++ Language Reference*.

The **#pragma options** preprocessor directive must appear before the first z/OS XL C statement in your input source file. Only comments and other preprocessor directives can precede the **#pragma options** directive. Only the options that are listed below can be specified in a **#pragma options** directive. If you specify a compiler option that is not in the following list, the compiler generates a warning message, and does not use the option.

AGGREGATE	ALIAS
ANSIALIAS	ARCHITECTURE
CHECKOUT	GONUMBER
IGNERRNO	INLINE
LIBANSI	MAXMEM
OBJECT	OPTIMIZE
RENT	SERVICE
SPILL	START
TEST	TUNE
UPCONV	XREF

Notes:

1. When you specify conflicting attributes explicitly, or implicitly by the specification of other options, the last explicit option is accepted. The compiler usually does not issue a diagnostic message indicating that it is overriding any options.
2. When you compile your program with the SOURCE compiler option, an options list in the listing indicates the options in effect at invocation. The values in the list are the options that are specified on the command line, or the default options that were specified at installation. These values do not reflect options that are specified in the **#pragma options** directive.

Specifying compiler options under z/OS UNIX System Services

The c89 and xlc utilities invoke the z/OS XL C/C++ compiler with the C and C++ compiler options. For further information, see "Compiler option defaults" on page 46.

To change compiler options, use an appropriate c89 or xlc utility option. For example, use -I to set the search option that specifies where to search for #include files. If there is no appropriate c89 or xlc option, use **-q** or **-Wc** to specify a desired compiler option. For example, specify **-Wc,expo** to export all functions and variables.

For a complete description of c89, xlc, and related utilities, refer to Chapter 21, “c89 — Compiler invocation using host environment variables,” on page 535 or Chapter 24, “xlc — Compiler invocation using a customizable configuration file,” on page 577.

For compiler options that take file names as suboptions, you can specify a sequential data set, a partitioned data set, or a partitioned data set member by prefixing the name with two slashes (’//’). The rest of the name follows the same syntax rule for naming data sets. Names that are not preceded with two slashes are z/OS UNIX System Services file names. For example, to specify `HQ.PROG.LIST` as the source listing file (`HQ` being the high-level qualifier), use `SOURCE(//'HQ.PROG.LIST')`. The single quotation mark is needed for specifying a full file name with a high-level qualifier.

Note: As of z/OS V1R8, the IPA Link step makes use of 64-bit virtual memory, which may cause the z/OS XL C/C++ compiler to abend if there is insufficient storage. Increasing the default MEMLIMIT system parameter size in the SMFPRMx parmlib member to 3000 MB can overcome the problem. The default takes effect if a job does not specify MEMLIMIT in the JCL JOB or EXEC statement, or REGION=0 in the JCL; the MEMLIMIT specified in an IEFUSI exit routine overrides all other MEMLIMIT settings. For information on the **ulimit** command, which can be used in z/OS UNIX to set MEMLIMIT, see *z/OS UNIX System Services Command Reference*. For additional information about the MEMLIMIT system parameter, see *z/OS MVS Programming: Extended Addressability Guide*.

Compiler option defaults

You can use various options to change the compilation of your program. You can specify compiler options when you invoke the compiler or, in a C program, in a **#pragma options** directive in your source program. Options, that you specify when you invoke the compiler, override installation defaults and compiler options that are specified through a **#pragma options** directive.

The compiler option defaults that are supplied by IBM can be changed to other selected defaults when z/OS XL C/C++ is installed. For further information, see Appendix F, “Customizing default options for z/OS XL C/C++ compiler,” on page 673.

To find out the current defaults, compile a program with only the SOURCE compiler option specified. The compiler listing shows the options that are in effect at invocation. The listing does not reflect options that are specified through a **#pragma options** directive in the source file.

The c89 and xlc utilities that run in the z/OS UNIX System Services shell specify certain compiler options in order to support POSIX standards. For a complete description of these utilities, refer to Chapter 21, “c89 — Compiler invocation using host environment variables,” on page 535, Chapter 24, “xlc — Compiler invocation using a customizable configuration file,” on page 577, or to the *z/OS UNIX System Services Command Reference*. For some options, these utilities specify values that are different than the supplied defaults in MVS batch or TSO environments. However, for many options, they specify the same values as in MVS batch or TSO.

There are also some options that these utilities do not specify explicitly. In those cases, the default value is the same as in batch or TSO. An option that you specify explicitly using these z/OS UNIX System Services utilities overrides the setting of the same option if it is specified using a **#pragma options** directive. The exception is CSECT, where the **#pragma csect** directive takes precedence.

In effect, invoking the compiler with the c89 and xlc utilities overrides the default values for many options, compared to running the compiler in MVS batch or TSO. For example, the c89 utility specifies the RENT option, while the compiler default in MVS batch or TSO is NORENT. Any overrides of the defaults by the c89 and xlc utilities are noted in the DEFAULT category for the option. As the compiler defaults can always be changed during installation, you should always consult the compiler listing to verify the values passed to the compiler. See "Using the z/OS XL C compiler listing" on page 265 and "Using the z/OS XL C++ compiler listing" on page 302 for more information.

The c89 utility remaps the following options to the values shown. Note that these values are set for a regular (non-IPA) compile. These values will change if you invoke IPA Compile, IPA Link, or specify certain other options. For example, specifying the **c89 -V** option changes the settings of many of the compiler listing options. See Chapter 21, "c89 — Compiler invocation using host environment variables," on page 535 or Chapter 24, "xlc — Compiler invocation using a customizable configuration file," on page 577 for more information and also refer to the default information provided for each compiler option.

The **c89** options remapped are as follows:

```
LOCALE(POSIX)
LANGLVL(ANSI)
OE
LONGNAME
RENT
OBJECT(file_name.o)
NOLIST(/dev/fd1)
NOSOURCE(/dev/fd1)
NOPPONLY(NOCOMMENTS,NOLINES,/dev/fd1,2048)
FLAG(W)
DEFINE(errno=\\(*__errno\\(\\)\\))
DEFINE(_OPEN_DEFAULT=1)
```

The **c89** command name supported by the xlc utility has the same defaults as the **c89** command name supported by the c89 utility.

The **cc** options remapped are as follows:

```
NOANSIALIAS
LOCALE(POSIX)
LANGLVL(COMMONC)
OE
LONGNAME
RENT
OBJECT(file_name.o)
NOLIST(/dev/fd1)
NOSOURCE(/dev/fd1)
NOPPONLY(NOCOMMENTS,NOLINES,/dev/fd1,2048)
FLAG(W)
DEFINE(errno=\\(*__errno\\(\\)\\))
DEFINE(_OPEN_DEFAULT=0)
DEFINE(_NO_PROTO=1)
```

The **cc** command name supported by the xlc utility has the same defaults as the **cc** command name supported by the c89 utility.

The **c++** options remapped are as follows:

```
LOCALE(POSIX)
OE
OBJECT(file_name.o)
NOINLRPT(/dev/fd1)
NOLIST(/dev/fd1)
NOSOURCE(/dev/fd1)
NOPPONLY(NOCOMMENTS,NOLINES,/dev/fd1,2048)
FLAG(W)
DEFINE(errno=\\(*__errno\\(\\)\\))
DEFINE(_OPEN_DEFAULT=1)
```

All C++ command names (**xlc**, **cxx**, **c++**, **xlc++**) supported by the xlc utility have the same defaults as the **c++** and **cxx** commands supported by the c89 utility.

The **xlc** and **c99** command names supported by the xlc utility have the same defaults as the **c89** command name supported by the c89 utility, except for the following:

- LANGLVL(EXTENDED) is the default for the **xlc** command name
- LANGLVL(STDC99) is the default for the **c99** command name
- SSCOMM is the default for both the **c99** and **xlc** command names

Note that the **locale** option is set according to the environment where the **cc**, **c89**, and **c++** commands are invoked. The current execution locale is determined by the values associated with environment variables LANG and LC_ALL. The following list shows the order of precedence for determining the current execution locale:

- If you specify LC_ALL, the current execution locale will be the value associated with LC_ALL.
- If LC_ALL was not specified but LANG was specified, the current execution locale will be the value associated with LANG.
- If neither of the two environment variables is specified, the current execution locale will default to ″C″.
- If the current execution locale is ″C″, the compiler will be invoked with LOCALE(POSIX); otherwise, it will be invoked with the current execution locale.

Note that for SEARCH, the *value* is determined by the following:

- Additional include search directories identified by the **c89 -I** options. Refer to Chapter 21, “c89 — Compiler invocation using host environment variables,” on page 535 for more information.
- z/OS UNIX System Services environment variable settings: *prefix*_INCDIRS, *prefix*_INCLIBS, and *prefix*_CYSLIB. They are normally set during compiler installation to reflect the compiler and run-time include libraries. Refer to “Environment variables” on page 550 in Chapter 21, “c89 — Compiler invocation using host environment variables,” on page 535 for more information.

Refer to “SEARCH | NOSEARCH” on page 219 for more information on SEARCH.

For the remainder of the compiler options, the c89 utility default matches the C/C++ default. Some of these are explicitly specified by **c89**, **cc**, or **c++**. Therefore if the installation changes the default options, you may find that **c89**, **cc**, or **c++** continues to use the default options. You can use the _C89_OPTIONS, _CC_OPTIONS, or _CXX_OPTIONS environment variable to override these settings if necessary. Note that certain options are required for the correct execution of **c89**, **cc**, or **c++**.

Summary of compiler options

Most compiler options have a positive and negative form. The negative form is the positive with NO before it. For example, NOXREF is the negative form of XREF.

Table 6 lists the compiler options in alphabetical order, their abbreviations, and the defaults that are supplied by IBM. Suboptions inside square brackets are optional.

Note: For a description of the compiler options that can be specified with **xlc**, type **xlc** without arguments to access the help file.

The C, C++, and IPA Link columns, which are shown in Table 6, indicate where the option is accepted by the compiler but this acceptance does not necessarily cause an action; for example, IPA LINK accepts the MARGINS option but ignores it. This acceptance also means that a diagnostic message is not generated. ″C″ refers to a C language compile step. ″C++″ refers to a C++ language compile step. These options are accepted regardless of whether they are for NOIPA, IPA (OBJONLY), or IPA(NOLINK).

Table 6. Compiler options, abbreviations, and IBM-supplied defaults

Compiler Option (Abbreviated Names are underlined)	IBM-supplied Default	C	C++	IPA Link	More Information
AGGRCOPY (OVERLAP | NOOVERLAP)	AGGRC(NOOVERL)	✓	✓	✓	See 63
AGGREGATE | NOAGGREGATE	NOAGG	✓		✓	See 64
ALIAS[(name)] | NOALIAS	NOALI	✓		✓	See 65
ANSIALIAS | NOANSIALIAS	ANS	✓	✓	✓	See 66
ARCHITECTURE(*n*)	ARCH(5)	✓	✓	✓	See 69
ARGPARSE | NOARGPARSE	ARG	✓	✓	✓	See 72
ARMODE | NOARMODE	NOARMODE	✓			See 73
ASCII | NOASCII	NOASCII	✓	✓	✓	See 74
ASMDATASIZE	ASMDATASIZE(256)	✓			See 75
ASSERT(RESTRICT) | ASSERT(NORESTRICT)	ASSERT(RESTRICT)	✓	✓	✓	See 76
ATTRIBUTE[(FULL)] | NOATTRIBUTE	NOATT		✓	✓	See 77
BITFIELD(SIGNED|UNSIGNED)	BITF(UNSIGNED)	✓	✓	✓	See 78
CHARS(SIGNED | UNSIGNED)	CHARS(UNSIGNED)	✓	✓	✓	See 78
CHECKOUT(subopts) | NOCHECKOUT	NOCHE	✓		✓	See 79
CICS[(subopts)] | NOCICS	NOCICS	✓	✓	✓	See 81
COMPACT | NOCOMPACT	NOCOMPACT	✓	✓	✓	See 83
COMPRESS | NOCOMPRESS	NOCOMPRESS	✓	✓	✓	See 85
CONVLIT[(subopts)] | NOCONVLIT[(subopts)]	NOCONV (, NOWCHAR)	✓	✓	✓	See 86
CSECT([qualifier]) | NOCSECT([qualifier])	NOCSE for NOGOFF or CSE() for GOFF	✓	✓	✓	See 88
CVFT | NOCVFT	CVFT		✓		See 91
DBRMLIB[(filename)]	DBRMLIB(DD:DBRMLIB)	✓	✓	✓	See 92
DEBUG[(subopts)] | NODEBUG[(subopts)]	NODEBUG	✓	✓		See 94
DEFINE(name1[= | =def1], name2[= | =def2],...)	**Note:** No default user definitions.	✓	✓	✓	See 98
DFP | NODFP	NODFP	✓	✓	✓	See 100

Table 6. Compiler options, abbreviations, and IBM-supplied defaults (continued)

Compiler Option (Abbreviated Names are underlined)	IBM-supplied Default	C	C++	IPA Link	More Information
DIGRAPH | NODIGRAPH	DIGR	✓	✓	✓	See 101
DLL(CBA | NOCBA) | NODLL(CBA | NOCBA)	For C: NODLL(NOCBA) For C++: DLL(NOCBA)	✓	✓	✓	See 102
ENUMSIZE(subopts)	ENUM(SMALL)	✓	✓	✓	See 105
EPILOG(subopt)	**Note:** The compiler generates default epilog code for the functions that do not have user-supplied epilog code.	✓			See 106
EVENTS[(filename)] | NOEVENTS	NOEVENT	✓	✓	✓	See 108
EXECOPS | NOEXECOPS	EXEC	✓	✓	✓	See 109
EXH | NOEXH	EXH		✓		See 110
EXPMAC | NOEXPMAC	NOEXP	✓	✓	✓	See 111
EXPORTALL | NOEXPORTALL	NOEXPO	✓	✓	✓	See 111
FASTTEMPINC | NOFASTTEMPINC	NOFASTT		✓		See 112
FLAG(severity) | NOFLAG	FL(I)	✓	✓	✓	See 113
FLOAT(subopts)	FLOAT(HEX, FOLD, NOMAF, NORRM, AFP(NOVOLATILE))	✓	✓	✓	See 114
GENASM[(filename)] | NOGENASM	NOGENASM	✓			See 120
GOFF | NOGOFF	NOGOFF	✓	✓	✓	See 121
GONUMBER | NOGONUMBER	NOGONUM	✓	✓	✓	See 123
HALT(num)	HALT(16)	✓	✓	✓	See 124
HALTONMSG(msgno) | NOHALTONMSG	NOHALTON	✓	✓	✓	See 125
HGPR | NOHGPR	NOHGPR	✓	✓	✓	See 125
HOT | NOHOT	NOHOT	✓	✓		See 127
IGNERRNO | NOIGNERRNO	NOIGNER	✓	✓	✓	See 127
INFO[(subopts)] | NOINFO	For C++: IN(LAN) For C: NOIN	✓	✓		See 129
INITAUTO(number [,word]) | NOINITAUTO	NOINITA	✓	✓	✓	See 132
INLINE(subopts) | NOINLINE [(subopts)]	C/C++ NOOPT: NOINL(AUTO, NOREPORT, 100, 1000) C/C++ OPT: INL(AUTO, NOREPORT, 100, 1000) IPA Link NOOPT: NOINL(AUTO, NOREPORT, 1000, 8000) IPA Link OPT: INL (AUTO, NOREPORT, 1000, 8000)	✓	✓	✓	See 133
INLRPT[(filename)] | NOINLRPT[(filename)]	NOINLR	✓	✓	✓	See 137

Table 6. Compiler options, abbreviations, and IBM-supplied defaults (continued)

Compiler Option (Abbreviated Names are underlined)	IBM-supplied Default	C	C++	IPA Link	More Information
IPA[(subopts)] | NOIPA[(subopts)]	NOIPA(NOLINK, OBJECT, OPT, NOLIST, NOGONUMBER, NOATTRIBUTE, NOXREF, LEVEL(1), NOMAP, DUP, ER, NONCAL, NOUPCASE, NOCONTROL, NOPDF1, NOPDF2, NOPDFNAME)	✓	✓	✓	See 139
KEYWORD(name) | NOKEYWORD(name)	Recognizes all C++ keywords		✓	✓	See 146
LANGLVL(subopts)	LANG(EXTENDED)	✓	✓	✓	See 147
LIBANSI | NOLIBANSI	NOLIB	✓	✓	✓	See 162
LIST[(filename)] | NOLIST [(filename)]	NOLIS	✓	✓	✓	See 163
LOCALE[(name)] | NOLOCALE	NOLOC	✓	✓	✓	See 165
LONGNAME | NOLONGNAME	C:NOLO C++: LO	✓	✓	✓	See 167
LP64 | ILP32	ILP32	✓	✓	✓	See 169
LSEARCH(subopts) | NOLSEARCH	NOLSE	✓	✓	✓	See 171
MAKEDEP[(filename)]	**Note:** This option is only supported using **-q** syntax.	✓	✓		See 177
MARGINS | NOMARGINS	NOMAR		✓		See 178
MARGINS(m,n) | NOMARGINS	V-format: NOMAR F-format: MAR(1,72)	✓		✓	See 178
MAXMEM(size) | NOMAXMEM	MAXM(2097152)	✓	✓	✓	See 180
MEMORY | NOMEMORY	MEM	✓	✓	✓	See 181
METAL | NOMETAL	NOMETAL	✓			See 182
NAMEMANGLING(subopt)	NAMEMANGLING(zOSV1R2)		✓		See 185
NESTINC(num) | NONESTINC	NEST(255)	✓	✓	✓	See 188
OBJECT[(filename)] | NOOBJECT [(filename)]	OBJ	✓	✓	✓	See 188
OBJECTMODEL(subopt)	OBJECTMODEL(COMPAT)		✓	✓	See 190
OE[(filename)] | NOOE[(filename)]	NOOE	✓	✓	✓	See 192
OFFSET | NOOFFSET	NOOF	✓	✓	✓	See 193
OPTFILE[(filename)] | NOOPTFILE[(filename)]	NOOPTF	✓	✓	✓	See 194
OPTIMIZE[(level)] | NOOPTIMIZE	NOOPT	✓	✓	✓	See 196
PHASEID | NOPHASEID	NOPHASEID	✓	✓	✓	See 200
PLIST(HOST | OS)	PLIST(HOST)	✓	✓	✓	See 200
PORT(PPS | NOPPS) | NOPORT(PPS | NOPPS)	NOPORT(NOPPS)		✓		See 201
PPONLY[(subopts)] | NOPPONLY [(subopts)]	NOPP	✓	✓	✓	See 203
PREFETCH | NOPREFETCH	PREFETCH	✓	✓		See 206
PROLOG(subopt)	**Note:** The compiler generates default prolog code for the functions that do not have user-supplied prolog code.	✓			See 206
REDIR | NOREDIR	RED	✓	✓	✓	See 208

Table 6. Compiler options, abbreviations, and IBM-supplied defaults (continued)

Compiler Option (Abbreviated Names are underlined)	IBM-supplied Default	C	C++	IPA Link	More Information
RENT | NORENT	NORENT	✓		✓	See 209
REPORT | NOREPORT	NOREPORT	✓	✓	✓	See 210
RESERVED_REG(subopt)	**Note:** No default user definitions.	✓			See 211
ROCONST | NOROCONST	C: NOROC C++: ROC	✓	✓	✓	See 212
ROSTRING | NOROSTRING	ROS	✓	✓	✓	See 214
ROUND(opt)	For IEEE: ROUND(N). For HEX: ROUND(Z). For DFP: ROUND(DN).	✓	✓	✓	See 215
RTCHECK(subopts) | NORTCHECK	NORTCHECK	✓	✓	✓	See 217
RTTI | NORTTI	NORTTI		✓		See 218
SEARCH(opt1,opt2,...) | NOSEARCH	For C++, SE(//'CEE.SCEEH.+', //'CBC.SCLBH.+') For C, SE(//'CEE.SCEEH.+')	✓	✓	✓	See 219
SEQUENCE | NOSEQUENCE	NOSEQ		✓		See 221
SEQUENCE(m,n) | NOSEQUENCE	V-format: NOSEQ F-format: SEQ(73,80)	✓		✓	See 221
SERVICE(string) | NOSERVICE	NOSERV	✓	✓	✓	See 222
SHOWINC | NOSHOWINC	NOSHOW	✓	✓	✓	See 223
SHOWMACROS | NOSHOWMACROS	NOSHOWM	✓	✓		See 224
SKIPSRC (SHOW | HIDE)	SKIPS(SHOW)	✓	✓	✓	See 225
SOURCE[(filename)] | NOSOURCE[(filename)]	NOSO	✓	✓	✓	See 226
SPILL(size) | NOSPILL[(size)]	SP(128)	✓	✓	✓	See 227
SPLITLIST | NOSPLITLIST	NOSPLITLIST	✓	✓	✓	See 229
SQL | NOSQL	NOSQL	✓	✓	✓	See 232
SSCOMM | NOSSCOMM	NOSS	✓		✓	See 234
START | NOSTART	STA	✓	✓	✓	See 235
STATICINLINE | NOSTATICINLINE	NOSTATICI	✓	✓	✓	See 236
STRICT | NOSTRICT	STRICT	✓	✓	✓	See 237
STRICT_INDUCTION | NOSTRICT_INDUCTION	NOSTRICT_INDUC	✓	✓	✓	See 238
SUPPRESS(msg-no) | NOSUPPRESS	NOSUPP	✓	✓	✓	See 239
TARGET(suboption)	TARG(LE, CURRENT)	✓	✓	✓	See 240
TEMPINC[(filename)] | NOTEMPINC[(filename)]	PDS: TEMPINC(TEMPINC) z/OS UNIX System Services file system directory: TEMPINC(./tempinc)		✓		See 244
TEMPLATERECOMPILE | NOTEMPLATERECOMPILE	TEMPLATEREC		✓		See 246
TEMPLATEREGISTRY | NOTEMPLATEREGISTRY	NOTEMPL		✓	✓	See 247

Table 6. Compiler options, abbreviations, and IBM-supplied defaults (continued)

Compiler Option (Abbreviated Names are underlined)	IBM-supplied Default	C	C++	IPA Link	More Information
TERMINAL | NOTERMINAL	TERM	✓	✓	✓	See 248
TEST[(subopts)] | NOTEST[(subopts)]	C: NOTEST (HOOK, SYM, BLOCK, LINE, PATH) C++: NOTEST(HOOK)	✓	✓	✓	See 249
TMPLPARSE(subopts)	TMPLPARSE(NO)		✓		See 253
TUNE(n)	TUN(5)	✓	✓	✓	See 254
UNDEFINE(name)	**Note:** No default.	✓	✓	✓	See 256
UNROLL(subopts)	UNROLL(AUTO)	✓	✓	✓	See 257
UPCONV | NOUPCONV	NOUPC	✓		✓	See 257
WARN64 | NOWARN64	NOWARN64	✓	✓	✓	See 258
WARN0X | NOWARN0X	NOWARN0X	✓	✓	✓	See 259
WSIZEOF| NOWSIZEOF	NOWSIZEOF	✓	✓	✓	See 260
XPLINK[(subopts)] | NOXPLINK[(subopts)]	NOXPL	✓	✓	✓	See 260
XREF | NOXREF	NOXR	✓	✓	✓	See 264

Compiler output options

The options in Table 7 control the type of file output the compiler produces, as well as the locations of the output. These are the basic options that determine the compiler components that will be invoked, the preprocessing, compilation, and linking steps that will (or will not) be taken, and the kind of output to be generated.

Table 7. Compiler output options

Option	Description	C Compile	C++ Compile	IPA Link	More Information
DBRMLIB	Provides the location for the database request module used in conjunction with the SQL option.	✓	✓	✓	See 92
GENASM	Instructs the compiler to generate HLASM source code instead of object code for the program being compiled.	✓			See 120
MAKEDEP	Analyzes each source file to determine what dependency it has on other files and places this information into an output file.	✓	✓		See 177
OBJECT	Produces an object module, and stores it in the file that you specify, or in the data set associated with SYSLIN.	✓	✓	✓	See 188
PPONLY	Specifies that only the preprocessor is to be run and not the compiler.	✓	✓	✓	See 203
SHOWMACROS	Emits macro definitions at the end of compilation to preprocessed output.	✓	✓		See 224

Compiler input options

The options in Table 8 on page 54 specify the type and location of your source files.

Table 8. Compiler input options

Option	Description	C Compile	C++ Compile	IPA Link	More Information
LSEARCH	Specifies the directories or data sets to be searched for user include files.	✓	✓	✓	See 171
MARGINS	Specifies, inclusively, the range of source column numbers that will be compiled.	✓	✓	✓	See 178
NESTINC	Specifies the number of nested include files to be allowed in your source program.	✓	✓	✓	See 188
OE	Specifies the rules used when searching for files specified with `#include` directives.	✓	✓	✓	See 192
SEARCH	Specifies the directories or data sets to be searched for system include files.	✓	✓	✓	See 219
SEQUENCE	Specifies the columns used for sequence numbers.	✓	✓	✓	See 221

Language element control options

The options in Table 9 allow you to specify the characteristics of the source code. You can also use these options to enforce or relax language restrictions and enable or disable language extensions.

Table 9. Language element control options

Option	Description	C Compile	C++ Compile	IPA Link	More Information
CICS	Enables CICS statements to be embedded in C/C++ source and passes them through the compiler without the need for an explicit preprocessing step.	✓	✓	✓	See 81
DEFINE	Defines a macro as in a `#define` preprocessor directive.	✓	✓	✓	See 98
DIGRAPH	Enables recognition of digraph key combinations or keywords to represent characters not found on some keyboards.	✓	✓	✓	See 101
KEYWORD	Controls whether the specified *name* is treated as a keyword or an identifier whenever it appears in your C++ source.		✓	✓	See 146
LANGLVL	Determines whether source code and compiler options should be checked for conformance to a specific language standard, or subset or superset of a standard.	✓	✓	✓	See 147
SQL	Enables the compiler to process embedded SQL statements.	✓	✓	✓	See 232
SSCOMM	Allows comments to be specified by two slashes (//), which supports C++ style comments in C code.	✓		✓	See 234
STATICINLINE	Controls whether inline functions are treated as having `static` or `extern` linkage.		✓	✓	See 236
UNDEFINE	Undefines preprocessor macro names.	✓	✓	✓	See 256

C++ template options

You can use the options in Table 10 to control how the C++ compiler handles templates.

Table 10. C++ template options

Option	Description	C Compile	C++ Compile	IPA Link	More Information
FASTTEMPINC	Defers generation of object code until the final version of all template definitions have been determined.		✓		See 112
TEMPINC	Generates separate template instantiation files for template functions and class declarations, and places these files in a directory or PDS, which can be optionally specified.		✓		See 244
TEMPLATERECOMPILE	Helps to manage dependencies between compilation units that have been compiled using the TEMPLATEREGISTRY option.		✓		See 246
TEMPLATEREGISTRY	Maintains records of all templates as they are encountered in the source and is designed to ensure that only one instantiation of each template is made.		✓	✓	See 247
TMPLPARSE	Controls whether parsing and semantic checking are applied to template definitions or only to template instantiations.		✓		See 253

Object code control options

The options in Table 11 affect the characteristics of the object code generated by the compiler.

Table 11. Object code control options

Option	Description	C Compile	C++ Compile	IPA Link	More Information
ALIAS	Generates ALIAS binder control statements, which help the binder locate modules in a load library, for each required entry point.	✓		✓	See 65
ARGPARSE	Parses arguments provided on the invocation line.	✓	✓	✓	See 72
ARMODE	Specifies that all functions in the C source file will operate in access-register (AR) mode. ARMODE must be used with the METAL compiler option.	✓			See 73
ASMDATASIZE	Provides the default data area size for the data areas defined by user-supplied assembly statements.	✓			See 75
COMPRESS	Suppresses the generation of function names in the function control block, thereby reducing the size of your application's load module.	✓	✓	✓	See 85

Table 11. Object code control options (continued)

Option	Description	C Compile	C++ Compile	IPA Link	More Information
CSECT	Instructs the compiler to generate CSECT names in the output object module.	✓	✓	✓	See 88
CVFT	Shrinks the size of the writeable static area (WSA), and reduces the size of construction virtual function tables (CVFT), which in turn reduces the load module size to improve your application's performance.		✓		See 91
DLL	Generates object code for DLLs or DLL applications.	✓	✓	✓	See 102
| EPILOG (C only)	Enables you to provide your own function exit code for all your functions that have `extern` scope.	✓			See 106
EXECOPS	Allows you to specify run-time options on the invocation line for the generated executable.	✓	✓	✓	See 109
EXH	Controls whether C++ exception handling is enabled in the module being compiled.		✓		See 110
EXPORTALL	Exports all externally defined functions and variables in the compilation unit so that a DLL application can use them.	✓	✓	✓	See 111
GOFF	Instructs the compiler to produce an object file in the Generalized Object File Format (GOFF).	✓	✓	✓	See 121
ILP32	Instructs the compiler to generate AMODE 31 code.	✓	✓	✓	See 169
LOCALE	Specifies the locale to be used by the compiler as the current locale throughout the compilation unit.	✓	✓	✓	See 165
LONGNAME	Provides support for external names of mixed case and up to 1024 characters long.	✓	✓	✓	See 167
LP64	Instructs the compiler to generate AMODE 64 code utilizing the z/Architecture 64-bit instructions.	✓	✓	✓	See 169
METAL	Generates HLASM code that has no Language Environment run-time dependencies and follows the MVS linkage conventions for passing parameters, returning values, and setting up function save areas.	✓			See 182
OBJECTMODEL	Sets the object model to be used for structures, unions, and classes.		✓	✓	See 190
PLIST	Specifies that the original operating system parameter list should be available.	✓	✓	✓	See 200
| PROLOG (C only)	Enables you to provide your own function entry code for all your functions that have `extern` scope.	✓			See 206
REDIR	Allows redirection of `stderr`, `stdin`, and `stdout` from the command line.	✓	✓	✓	See 208

Table 11. Object code control options (continued)

Option	Description	C Compile	C++ Compile	IPA Link	More Information
RENT	Generates reentrant code.	✓		✓	See 209
RESERVED_REG	Instructs the compiler not to use the specified general purpose register (GPR) during the compilation.	✓			See 211
ROCONST	Specifies the storage location for constant values.	✓	✓	✓	See 212
ROSTRING	Specifies the storage type for string literals.	✓	✓	✓	See 214
RTTI	Generates run-time type identification (RTTI) information for exception handling and for use by the `typeid` and `dynamic_cast` operators.		✓		See 218
START	Generates a CEESTART, which is an object that controls initialization at execution, when necessary.	✓	✓	✓	See 235
TARGET	Generates an object module for the targeted operating system or run-time library.	✓	✓	✓	See 240
WSIZEOF	Causes the `sizeof` operator to return the widened size for function return types.	✓	✓	✓	See 260
XPLINK	Uses a z/OS linkage specifically designed to increase performance.	✓	✓	✓	See 260

Floating-point and integer control options

Specifying the details of how your applications perform calculations can allow you to take better advantage of your system's floating-point performance and precision, including how to direct rounding. However, keep in mind that strictly adhering to IEEE floating-point specifications can impact the performance of your application. Using the options in Table 12, you can control trade-offs between floating-point performance and adherence to IEEE standards.

The table also lists options that allow you to control the characteristics of integer variables, values and types.

Table 12. Floating-point and integer control options

Option	Description	C Compile	C++ Compile	IPA Link	More Information
BITFIELD	Specifies whether bit fields are `signed` or `unsigned`.	✓	✓	✓	See 78
CHARS	Determines whether all variables of type `char` are treated as either `signed` or `unsigned`.	✓	✓	✓	See 78
DFP	Provides support for decimal floating-point types.	✓	✓	✓	See 100
ENUMSIZE	Specifies the amount of storage occupied by enumerations.	✓	✓	✓	See 105

Table 12. Floating-point and integer control options (continued)

Option	Description	C Compile	C++ Compile	IPA Link	More Information
FLOAT	Selects different strategies for speeding up or improving the accuracy of floating-point calculations.	✓	✓	✓	See 114
ROUND	Specifies the rounding mode for the compiler to use when evaluating constant floating-point expressions at compile time.	✓	✓	✓	See 215

Error-checking and debugging options

The options in Table 13 allow you to detect and correct problems in your source code. In some cases, these options can alter your object code, increase your compile time, or introduce runtime checking that can slow down the execution of your application. The option descriptions indicate how extra checking can impact performance.

To control the amount and type of information you receive regarding the behavior and performance of your application, consult the "Listings, messages, and compiler information options" on page 59 section.

Table 13. Error-checking and debugging options

Option	Description	C Compile	C++ Compile	IPA Link	More Information
CHECKOUT	Produces informational messages for possible programming errors.	✓		✓	See 79
DEBUG	Instructs the compiler to generate debug information.	✓	✓		See 94
EVENTS	Produces an event file that contains error information and source file statistics.	✓	✓	✓	See 108
GONUMBER	Generates line number tables that correspond to the input source file for Debug Tool and CEEDUMP processing.	✓	✓	✓	See 123
HALT	Stops compilation before producing any object, executable, or assembler source files if the maximum severity of compile-time messages equals or exceeds the severity specified for this option.	✓	✓	✓	See 124
HALTONMSG	Stops compilation before producing any object, executable, or assembler source files if a specified error message is generated.	✓	✓	✓	See 125
INFO	Produces groups of informational messages.	✓	✓		See 129
INITAUTO	Initializes automatic variables to a specific value for debugging purposes.	✓	✓	✓	See 132
| RTCHECK	Generates compare-and-trap code that performs certain types of runtime checking.	✓	✓	✓	See 217
SERVICE	Places a *string* in the object module, which is displayed in the traceback if the application fails abnormally.	✓	✓	✓	See 222

Table 13. Error-checking and debugging options (continued)

Option	Description	C Compile	C++ Compile	IPA Link	More Information
TEST	Generates information that Debug Tool needs to debug your program.	✓	✓	✓	See 249
WARN64	Generates diagnostic messages, which enable checking for possible data conversion problems between 32-bit and 64-bit compiler modes.	✓	✓	✓	See 258
WARN0X (C++ only)	Generates messages about differences caused by the migration from C++98 standard to C++0x standard.		✓		See 259

Listings, messages, and compiler information options

The options in Table 14 allow you to control the listing file, as well as how and when to display compiler messages. You can use these options in conjunction with those in the "Error-checking and debugging options" on page 58 section to provide a more robust overview of your application when checking for errors and unexpected behavior.

Table 14. Listings, messages, and compiler information options

Option	Description	C Compile	C++ Compile	IPA Link	More Information
AGGREGATE	Lists structures and unions, and their sizes.	✓		✓	See 64
ATTRIBUTE	Produces a compiler listing that includes the attribute component of the attribute and cross-reference section of the listing.		✓	✓	See 77
EXPMAC	Lists all expanded macros in the source listing.	✓	✓	✓	See 111
FLAG	Limits the diagnostic messages to those of a specified level or higher.	✓	✓	✓	See 113
INLRPT	Generates a report on the status of inlined functions.	✓	✓	✓	See 137
LIST	Produces a compiler listing file that includes a pseudo assembly listing.	✓	✓	✓	See 163
OFFSET	Lists offset addresses relative to entry points of functions.	✓	✓	✓	See 193
PHASEID	Causes each compiler component (phase) to issue an informational message as each phase begins execution, which assists you with determining the maintenance level of each compiler component (phase). This message identifies the compiler phase module name, product identification, and build level.	✓	✓	✓	See 200
REPORT	Produces pseudo-C code listing files that show how sections of code have been optimized.	✓	✓	✓	See 210

Table 14. Listings, messages, and compiler information options (continued)

Option	Description	C Compile	C++ Compile	IPA Link	More Information
SHOWINC	When used with the SOURCE option to generate a listing file, selectively shows user or system header files in the source section of the listing file.	✓	✓	✓	See 223
SKIPSRC	Controls whether or not source statements skipped by the compiler are shown in the listing, when the -qsource option is in effect.	✓	✓	✓	See 225
SOURCE	Produces a compiler listing file that includes the source section of the listing.	✓	✓	✓	See 226
SPLITLIST	Enables the z/OS XL C/C++ compiler to write the IPA Link phase listing to multiple PDS members, PDSE members, or z/OS UNIX System Services files.	✓	✓	✓	See 229
SUPPRESS	Prevents specific informational or warning messages from being displayed or added to the listing file, if one is generated.	✓	✓	✓	See 239
TERMINAL	Directs diagnostic messages to be displayed on the terminal.	✓	✓	✓	See 248
XREF	Produces a compiler listing that includes a cross-reference listing of all identifiers.	✓	✓	✓	See 264

Optimization and tuning options

You can control the optimization and tuning process, which can improve the performance of your application at run time, using the options in Table 15. Remember that not all options benefit all applications. Trade-offs sometimes occur between an increase in compile time, a reduction in debugging capability, and the improvements that optimization can provide.

Table 15. Optimization and tuning options

Option	Description	C Compile	C++ Compile	IPA Link	More Information
AGGRCOPY	Enables destructive copy operations for structures and unions, which can improve performance.	✓	✓	✓	See 63
ANSIALIAS	Indicates to the compiler that the code strictly follows the type-based aliasing rule in the ISO C and C++ standards, and can therefore be compiled with higher performance optimization of the generated code.	✓	✓	✓	See 66
ARCHITECTURE	Specifies the machine architecture for which the executable program instructions are to be generated.	✓	✓	✓	See 69
ASSERT(RESTRICT)	Enables optimizations for restrict qualified pointers.	✓	✓	✓	See 76
COMPACT	Avoids optimizations that increase object file size.	✓	✓	✓	See 83

Table 15. Optimization and tuning options (continued)

Option	Description	C Compile	C++ Compile	IPA Link	More Information
HGPR	Enables the compiler to exploit 64-bit General Purpose Registers (GPRs) in 32-bit programs targeting z/Architecture hardware.	✓	✓	✓	See 125
HOT	Performs high-order loop analysis and transformations (HOT) during optimization.	✓	✓		See 127
IGNERRNO	Allows the compiler to perform optimizations that assume `errno` is not modified by system calls.	✓	✓	✓	See 127
INLINE	Attempts to inline functions instead of generating calls to those functions, for improved performance.	✓	✓	✓	See 133
IPA	Enables or customizes a class of optimizations known as interprocedural analysis (IPA).	✓	✓	✓	See 139
LIBANSI	Indicates whether or not functions with the name of an ANSI C library function are in fact ANSI C library functions and behave as described in the ANSI standard.	✓	✓	✓	See 162
MAXMEM	Limits the amount of memory used for local tables, and that the compiler allocates while performing specific, memory-intensive optimizations, to the specified number of kilobytes.	✓	✓	✓	See 180
OPTIMIZE	Specifies whether to optimize code during compilation and, if so, at which level.	✓	✓	✓	See 196
PREFETCH	Inserts prefetch instructions automatically where there are opportunities to improve code performance.	✓	✓		See 206
STRICT	Used to prevent optimizations done by default at optimization levels OPT(3), and, optionally at OPT(2), from re-ordering instructions that could introduce rounding errors.	✓	✓	✓	See 237
STRICT_INDUCTION	Prevents the compiler from performing induction (loop counter) variable optimizations. These optimizations may be unsafe (may alter the semantics of your program) when there are integer overflow operations involving the induction variables.	✓	✓	✓	See 238
TUNE	Tunes instruction selection, scheduling, and other implementation-dependent performance enhancements for a specific implementation of a hardware architecture.	✓	✓	✓	See 254
UNROLL	Controls loop unrolling, for improved performance.	✓	✓	✓	See 257

Portability and migration options

The options in Table 16 can help you maintain application behavior compatibility on past, current, and future hardware, operating systems and compilers, or help move your applications to an XL compiler with minimal change.

Table 16. Portability and migration options

Option	Description	C Compile	C++ Compile	IPA Link	More Information
ASCII	Provides ASCII/NLS support.	✓	✓	✓	See 74
CONVLIT	Turns on string literal codepage conversion.	✓	✓	✓	See 86
NAMEMANGLING	Specifies the name mangling scheme for external symbol names which have C++ linkage.		✓		See 185
PORT	Adjusts the error recovery action that the compiler takes when it encounters an ill-formed **#pragma pack** directive.		✓		See 201
UPCONV	Specifies whether the `unsigned` specification is preserved when integral promotions are performed.	✓		✓	See 257

Compiler customization options

The options in Table 17 allow you to specify alternate locations for configuration files, and internal compiler operation. You should only need to use these options in specialized installation or testing scenarios.

Table 17. Compiler customization options

Option	Description	C Compile	C++ Compile	IPA Link	More Information
MEMORY	Improves compile-time performance by using a memory file in place of a temporary work file, if possible.	✓	✓	✓	See 181
OPTFILE	Specifies where the compiler should look for additional compiler options.	✓	✓	✓	See 194
SPILL	Specifies the size (in bytes) of the internal program storage areas used by the optimizer for register spills.	✓	✓	✓	See 227

Description of compiler options

The following sections describe the compiler options and their usage. Compiler options are listed alphabetically. All compiler options are supported by both the XL C and XL C++ compiler, unless the option title is followed by ″(C only)″ or ″(C++ only)".

For each option, the following information is provided:

Category
The functional category to which the option belongs is listed here.

Pragma equivalent
Many compiler options allow you to use an equivalent pragma directive to

apply the option's functionality within the source code, limiting the scope of the option's application to a single source file, or even selected sections of code. Where an option supports the **#pragma options** (*option_name*) and/or **#pragma** *name* form of the directive, this is indicated.

Purpose
This section provides a brief description of the effect of the option (and equivalent pragmas), and why you might want to use it.

Syntax
This section provides the syntax for the option.

Defaults
In most cases, the default option setting is clearly indicated in the syntax diagram. However, for many options, there are multiple default settings, depending on other compiler options in effect. This section indicates the different defaults that may apply.

Parameters
This section describes the suboptions that are available for the option.

Usage This section describes any rules or usage considerations you should be aware of when using the option. These can include restrictions on the option's applicability, valid placement of pragma directives, precedence rules for multiple option specifications, and so on.

IPA effects
Where appropriate, provides information on the effect of the option during the IPA compile and/or IPA link steps.

Predefined macros
Many compiler options set macros that are protected (that is, cannot be undefined or redefined by the user). Where applicable, any macros that are predefined by the option, and the values to which they are defined, are listed in this section.

Examples
Where appropriate, examples of the command-line syntax are provided in this section.

Related information
Where appropriate, provides cross-references to related information.

AGGRCOPY

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Enables destructive copy operations for structures and unions, which can improve performance.

Syntax

```
                ┌─NOOVERL─┐
►►──AGGRC──(────┴─OVERL───┴──)──────────────────────────────────────►◄
```

Defaults

AGGRCOPY(NOOVERLAP)

Parameters

OVERLAP
: Specifies that the source and destination in a structure assignment might overlap in memory. Programs that do not comply to the ANSI C standard as it pertains to non-overlap of source and destination assignment may need to be compiled with the OVERLAP suboption.

NOOVERLAP
: Instructs the compiler to assume that the source and destination for structure and union assignments do not overlap. This assumption lets the compiler generate faster code.

Usage

The AGGRCOPY option instructs the compiler on whether or not the source and destination assignments for structures can overlap. They cannot overlap according to ISO Standard C rules. For example, in the assignment `a = b;`, where `a` and `b` are `structs`, `a` is the destination and `b` is the source.

IPA effects

The IPA Compile step generates information for the IPA Link step. The AGGRCOPY option affects the regular object module if you requested one by specifying the IPA(OBJECT) option.

The IPA Link step accepts the AGGRCOPY option, but ignores it.

The IPA Link step merges and optimizes the application code, and then divides it into sections for code generation. Each of these sections is a partition. The IPA Link step uses information from the IPA Compile step to determine if a subprogram can be placed in a particular partition. Only compatible subprograms are included in a given partition.

The value of the AGGRCOPY option for a partition is set to the value of the first subprogram that is placed in the partition. During IPA inlining, subprograms with different AGGRCOPY settings may be combined in the same partition. When this occurs, the resulting partition is always set to AGGRCOPY(OVERLAP).

Predefined macros

None.

AGGREGATE | NOAGGREGATE (C only)

Category

Listings, messages, and compiler information

Pragma equivalent

#pragma options (aggregate) (C only), **#pragma options (noaggregate)** (C only)

Purpose

Lists structures and unions, and their sizes.

Syntax

```
      ┌─NOAGG─┐
►►────┴─AGG───┴──────────────────────────────────────────────────────►◄
```

Defaults

NOAGGREGATE

For the z/OS UNIX System Services utilities, the default for a regular compile is NOAGGREGATE. To specify AGGREGATE, you must specify -V.

Usage

When the AGGREGATE compiler option is in effect, the compiler includes a layout of all `struct` or `union` types in the compiler listing.

Depending on the `struct` or `union` declaration, the maps are generated as follows:

- If the typedef name refers to a `struct` or `union`, one map is generated for the `struct` or `union` for which the typedef name refers to. If the typedef name can be qualified with the `_Packed` keyword, then a packed layout of the `struct` or `union` is generated as well. Each layout map contains the offset and lengths of the structure members and the union members. The layout map is identified by the `struct/union` tag name (if one exists) and by the typedef names.
- If the `struct` or `union` declaration has a tag, two maps are created: one contains the unpacked layout, and the other contains the packed layout. The layout map is identified by the `struct/union` tag name.
- If the `struct` or `union` declaration does not have a tag, one map is generated for the `struct` or `union` declared. The layout map is identified by the variable name that is specified on the `struct` or `union` declaration.

Predefined macros

None.

ALIAS | NOALIAS (C only)

Category

Object code control

Pragma equivalent

#pragma options (alias) (C only), **#pragma options (noalias)** (C only)

Purpose

Generates ALIAS binder control statements, which help the binder locate modules in a load library, for each required entry point.

When ALIAS is in effect with no suboption, the compiler selects an existing CSECT name from the program, and nominates it on the NAME statement

When you use an empty set of parentheses, as in ALIAS(), or specify NOALIAS, the compiler does not generate a NAME control statement.

Syntax

```
      ┌─NOALI──────────────────┐
►►────┴─ALI─┬──────────────┬───┴─────────────────────────────────────►◄
            ├─()───────────┤
            └─(──name──)───┘
```

Defaults
NOALIAS

Parameters
name

If you specify ALIAS(*name*), the compiler generates the following:

- Control statements in the object module.
- A NAME control statement in the form NAME *name* `(R)`. `R` indicates that the binder should replace the member in the library with the new member.

The compiler generates one ALIAS control statement for every external entry point that it encounters during compilation. These control statements are then appended to the object module.

Usage
If you specify the ALIAS option with LONGNAME, the compiler does not generate an ALIAS control statement.

Predefined macros
None.

Related information
For complete details on ALIAS and NAME control statements, see *z/OS MVS Program Management: User's Guide and Reference*.

ANSIALIAS | NOANSIALIAS

Category
Optimization and tuning

Pragma equivalent
#pragma options (ansialias) (C only), **#pragma options (noansialias)** (C only)

Purpose
Indicates to the compiler that the code strictly follows the type-based aliasing rule in the ISO C and C++ standards, and can therefore be compiled with higher performance optimization of the generated code.

When ANSIALIAS is in effect, you are making a promise to the compiler that your source code obeys the constraints in the ISO standard. On the basis of using this compiler option, the compiler front end passes aliasing information to the optimizer, which performs optimization accordingly.

When NOANSIALIAS is in effect, the optimizer assumes that a given pointer of a given type can point to an external object or any object whose address is taken, regardless of type. This assumption creates a larger aliasing set at the expense of performance optimization.

Syntax

```
     ┌─ANS───┐
►►───┴─NOANS─┴─────────────────────────────────────────────────►◄
```

Defaults
ANSIALIAS

The **cc** compiler invocation command for a regular compile in the z/OS UNIX System Services environment uses NOANSIALIAS as the default option.

Usage

When type-based aliasing is used during optimization, the optimizer assumes that pointers can only be used to access objects of the same type.

Type-based aliasing improves optimization in the following ways.

- It provides precise knowledge of what pointers can and cannot point at.
- It allows more loads to memory to be moved up and stores to memory moved down past each other, which allows the delays that normally occur in the original written sequence of statements to be overlapped with other tasks. These re-arrangements in the sequence of execution increase parallelism, which is desirable for optimization.
- It allows the removal of some loads and stores that otherwise might be needed in case those values were accessed by unknown pointers.
- It allows more identical calculations to be recognized (″commoning″).
- It allows more calculations that do not depend on values modified in a loop to be moved out of the loop (″code motion″).
- It allows better optimization of parameter usage in inlined functions.

Simplified, the rule is that you cannot safely dereference a pointer that has been cast to a type that is not closely related to the type of what it points at. The ISO C and C++ standards define the closely related types.

The following are not subject to type-based aliasing:

- Types that differ only in reference to whether they are signed or unsigned. For example, a pointer to a signed int can point to an `unsigned int`.
- Character pointer types (`char`, `unsigned char`, and in C but not C++ `signed char`).
- Types that differ only in their `const` or `volatile` qualification. For example, a pointer to a `const int` can point to an `int`.
- C++ types where one is a class derived from the other.

z/OS XL C/C++ compilers often expose type-based aliasing violations that other compilers do not. The C++ compiler corrects most but not all suspicious and incorrect casts without warnings or informational messages. For examples of aliasing violations that are detected and quietly fixed by the compiler, see the discussion of the `reinterpret_cast` operator in the *z/OS XL C/C++ Language Reference*.

In addition to the specific optimizations to the lines of source code that can be obtained by compiling with the ANSIALIAS compiler option, other benefits and advantages, which are at the program level, are described below:

- It reduces the time and memory needed for the compiler to optimize programs.
- It allows a program with a few coding errors to compile with optimization, so that a relatively small percentage of incorrect code does not prevent the optimized compilation of an entire program.
- It positively affects the long-term maintainability of a program by supporting ISO-compliant code.

It is important to remember that even though a program compiles, its source code may not be completely correct. When you weigh tradeoffs in a project, the short-term expedience of getting a successful compilation by forgoing performance optimization should be considered with awareness that you may be nurturing an incorrect program. The performance penalties that exist today could worsen as the compilers that base their optimization on strict adherence to ISO rules evolve in their ability to handle increased parallelism.

The ANSIALIAS compiler option only takes effect if the OPTIMIZE option is in effect.

If you specify LANGLVL(COMMONC), the ANSIALIAS option is automatically turned off. If you want ANSIALIAS turned on, you must explicitly specify it. Using LANGLVL(COMMONC) and ANSIALIAS together may have undesirable effects on your code at a high optimization level. See "LANGLVL" on page 147 for more information on LANGLVL(COMMONC).

A comment that indicates the ANSIALIAS option setting is generated in your object module to aid you in diagnosing a problem with your program.

Although type-based aliasing does not apply to the `volatile` and `const` qualifiers, these qualifiers are still subject to other semantic restrictions. For example, casting away a `const` qualifier might lead to an error at run time.

IPA effects

If the ANSIALIAS option is specified, then the IPA Link step phase will take advantage of the knowledge that the program will adhere to the standard C/C++ aliasing rules in order to improve its variable aliasing calculations.

Predefined macros

None.

Examples

The following example executes as expected when compiled unoptimized or with the NOANSIALIAS option; it successfully compiles optimized with ANSIALIAS, but does not necessarily execute as expected. On non-IBM compilers, the following code may execute properly, even though it is incorrect.

```
extern int y = 7.;

void main() {
     float x;
     int i;
     x = y;
     i = *(int *) &x;
     printf("i=%d. x=%f.\n", i, x);
}
```

In this example, the value in object `x` of type `float` has its stored value accessed via the expression `* (int *) &x`. The access to the stored value is done by the `*` operator, operating on the expression `(int *) &x`. The type of that expression is `(int *)`, which is not covered by the list of valid ways to access the value in the ISO standard, so the program violates the standard.

When ANSIALIAS (the default) is in effect, the compiler front end passes aliasing information to the optimizer that, in this case, an object of type `float` could not possibly be pointed to by an `(int *)` pointer (that is, that they could not be aliases for the same storage). The optimizer performs optimization accordingly. When it compares the instruction that stores into `x` and the instruction that loads out of `*(int *)`, it believes it is safe to put them in either order. Doing the load before the store will make the program run faster, so it interchanges them. The program becomes equivalent to:

```
extern int y = 7.;

void main() {
     float x;
     int i;
     int temp;
     temp = *(int *) &x;  /* uninitialized */
     x = y;
     i = temp;
     printf("i=%d. x=%f.\n", i, x);
}
```

The value stored into variable `i` is the old value of `x`, before it was initialized, instead of the new value that was intended. IBM compilers apply some optimizations more aggressively than some other compilers so correctness is more important.

Related information

For C, the CHECKOUT(CAST) compiler option can help you locate some but not all suspicious casts and ANSIALIAS violations. See "CHECKOUT | NOCHECKOUT (C only)" on page 79 to see how to obtain more diagnostic information.

ARCHITECTURE

Category

Optimization and tuning

Pragma equivalent

#pragma options (architecture) (C only)

Purpose

Specifies the machine architecture for which the executable program instructions are to be generated.

Syntax

```
►►──ARCH──(──n──)──────────────────────────────────────────►◄
```

Defaults

ARCH(5) for TARGET(zOSV1R7) and above.

Parameters

n Specifies the group to which a model number belongs.

Current groups of models that are supported include the following:

0 Produces code that is executable on all models.

1 Produces code that uses instructions available on the following system machine models:

- 9021-520, 9021-640, 9021-660, 9021-740, 9021-820, 9021-860, and 9021-900
- 9021-xx1 and 9021-xx2
- 9672-Rx1, 9672-Rx2 (G1), 9672-Exx, and 9672-Pxx

Specifically, these ARCH(1) machines and their follow-ons add the C Logical String Assist hardware instructions. These instructions are exploited by the compiler, when practical, for a faster and more compact implementation of some functions, for example, `strcmp()`.

2 It produces code that uses instructions available on the following system machine models:

- 9672-Rx3 (G2), 9672-Rx4 (G3), 9672-Rx5 (G4), and 2003

Specifically, these ARCH(2) machines and their follow-ons add the Branch Relative instruction (Branch Relative and Save - BRAS), and the halfword Immediate instruction set (for example, Add Halfword Immediate - AHI) which may be exploited by the compiler for faster processing.

3 Produces code that uses instructions available on the 9672-xx6 (G5), 9672-xx7 (G6), and follow-on models.

Specifically, these ARCH(3) machines and their follow-ons add a set of facilities for IEEE floating-point representation, as well as 12 additional floating-point registers and some new floating-point support instructions that may be exploited by the compiler.

Note that ARCH(3) is required for execution of a program that specifies the FLOAT(IEEE) compiler option. However, if the program is executed on a physical processor that does not actually provide these ARCH(3) facilities, any program check (operation or specification exception), resulting from an attempt to use features associated with IEEE floating point or the additional floating point registers, will be intercepted by the underlying operating system, and simulated by software. There will be a significant performance degradation for the simulation.

4 Produces code that uses instructions available on the 2064-xxx (z900) and 2066-xxx (z800) models in ESA/390 mode.

Specifically, the following instructions are used for long long operations:

- 32-bit Add-With-Carry (ALC, ALCR) for long long addition (rather than requiring a branch sequence)
- 32-bit Subtract-With-Borrow (SLB, SLBR) for long long subtraction (rather than requiring a branch sequence)
- Inline sequence with 32-bit Multiply-Logical (ML, MLR) for long long multiplication (rather than calling `@@MULI64`)

5 Is the default value. Produces code that uses instructions available on the 2064-xxx (z900) and 2066-xxx (z800) models in z/Architecture mode.

Specifically, this is required for execution of a program in 64-bit mode. If you specify the LP64 compiler option, the compiler will use ARCH(5) as the default. If you explicitly set ARCH to a lower level, the compiler will issue a warning and ignore your setting. ARCH(5) specifies the target machine architecture and the application can be either 31-bit or 64-bit.

6 Produces code that uses instructions available on the 2084-xxx (z990) and 2086-xxx (z890) models in z/Architecture mode.

Specifically, these ARCH(6) machines and their follow-ons add the long-displacement facility. For further information on the long-displacement facility, refer to *z/Architecture Principles of Operation*.

7 Produces code that uses instructions available on the 2096-xxx (IBM System z9® BC) and 2094-xxx models (IBM System z9 EC) in z/Architecture mode.

Specifically, these ARCH(7) machines and their follow-ons add instructions supported by the extended-immediate facility, which may be exploited by the compiler. Also, these machines add instructions supported by the decimal floating-point facility, which are generated if the DFP compiler option is specified and there are decimal floating-point data types in the source code. For further information on these facilities, refer to *z/Architecture Principles of Operation*.

8 Produces code that uses instructions available on the 2097-xxx models (IBM System z10™ EC) in z/Architecture mode.

Specifically, these ARCH(8) machines and their follow-ons add instructions supported by the general instruction extensions facility, which may be exploited by the compiler. Also, these machines add instructions supported by the decimal floating-point facility, which are generated if the DFP compiler option is specified and there are decimal floating-point data types in the source code. For further information on these facilities, refer to *z/Architecture Principles of Operation*.

Usage

When ARCHITECTURE is in effect, the compiler selects the instruction set available during the code generation of your program based on the specified machine architecture.

Specifying a higher ARCH level generates code that uses newer and faster instructions instead of the sequences of common instructions.

Note: Your application will not run on a lower architecture processor than what you specified using the ARCH option. Use the ARCH level that matches the lowest machine architecture where your program will run.

If you specify a group that does not exist or is not supported, the compiler uses the default, and issues a warning message.

Notes:

1. Code that is compiled at ARCH(1) runs on machines in the ARCH(1) group and later machines, including those in the ARCH(2) and ARCH(3) groups. It may not run on earlier machines. Code that is compiled at ARCH(2) may not run on ARCH(1) or earlier machines. Code that is compiled at ARCH(3) may not run on ARCH(2) or earlier machines.
2. For the system machine models, x indicates any value. For example, 9672-Rx4 means 9672-RA4 through to 9672-RX4, not just 9672-RX4.

IPA effects

If you specify the ARCHITECTURE option for any compilation unit in the IPA Compile step, the compiler generates information for the IPA Link step. This option also affects the regular object module if you request one by specifying the IPA(OBJECT) option.

The IPA Link step merges and optimizes the application code, and then divides it into sections for code generation. Each of these sections is a partition.

If you specify the ARCH option on the IPA Link step, it uses the value of that option for all partitions. The IPA Link step Prolog and all Partition Map sections of the IPA Link step listing display that value.

If you do not specify the option on the IPA Link step, the value used for a partition depends on the value that you specified for the IPA Compile step for each compilation unit that provided code for that partition. If you specified the same value for each compilation unit, the IPA Link step uses that value. If you specified different values, the IPA Link step uses the lowest level of ARCH.

The level of ARCH for a partition determines the level of TUNE for the partition. For more information on the interaction between ARCH and TUNE, see “TUNE” on page 254.

The Partition Map section of the IPA Link step listing, and the object module display the final option value for each partition. If you override this option on the IPA Link step, the Prolog section of the IPA Link step listing displays the value of the option.

The Compiler Options Map section of the IPA Link step listing displays the option value that you specified for each IPA object file during the IPA Compile step.

Predefined macros

__ARCH__ is predefined to the integer value of the ARCH compiler option.

Related information

Use the ARCH option in conjunction with the TUNE option. For more information on the interaction between ARCH and TUNE, see “TUNE” on page 254.

ARGPARSE | NOARGPARSE

Category

Object code control

Pragma equivalent

None.

Purpose

Parses arguments provided on the invocation line.

When ARGPARSE is in effect, arguments supplied to your program on the invocation line are parsed and passed to the `main()` routine in the C argument format, commonly `argc` and `argv`. `argc` contains the argument count, and `argv` contains the tokens after the command processor has parsed the string.

When NOARGPARSE is in effect, arguments on the invocation line are not parsed, `argc` has a value of `2`, and `argv` contains a pointer to the string.

Syntax

```
     ┌─ARG───┐
►►───┴─NOARG─┴─────────────────────────────────────────────────►◄
```

Defaults

ARGPARSE

Usage

If you specify NOARGPARSE, you cannot specify REDIR. The compiler will turn off REDIR with a warning since the whole string on the command line is treated as an argument and put into `argv`.

Note: NOARGPARSE is ignored for programs that utilize `spawn()` or `exec()` or for any program that is started by the z/OS UNIX System Services shell or by the BPXBATCH utility.

This option has no effect under CICS.

IPA effects

If you specify ARGPARSE for any compilation unit in the IPA Compile step, the compiler generates information for the IPA Link step. This option also affects the regular object module if you request one by specifying the IPA(OBJECT) option.

If you specify this option for both the IPA Compile and the IPA Link steps, the setting on the IPA Link step overrides the setting on the IPA Compile step. This applies whether you use ARGPARSE and NOARGPARSE as compiler options, or specify them using the **#pragma runopts** directive on the IPA Compile step.

If you specified ARGPARSE on the IPA Compile step, you do not need to specify it again on the IPA Link step to affect that step. The IPA Link step uses the information generated for the compilation unit that contains the `main()` function. If it cannot find a compilation unit that contains `main()`, it uses the information generated by the first compilation unit that it finds.

Predefined macros

None.

ARMODE | NOARMODE (C only)

Category

Object code control

Pragma equivalent

None.

Purpose

Specifies that all functions in the C source file will operate in access-register (AR) mode. ARMODE must be used with the METAL compiler option.

When ARMODE is in effect, all functions in the compilation unit will be compiled in AR mode. AR mode functions can access data stored in additional data spaces supported by IBM System z hardware.

To override the effect of the ARMODE option and selectively re-set particular functions to be in non-AR mode (or primary address space control mode), use `__attribute__((noarmode))`. For more information on this attribute, see **armode | noarmode type attribute (C only)** and *z/OS Metal C Programming Guide and Reference*.

When NOARMODE is in effect, functions are not in AR mode unless `__attribute__((armode))` is specifically specified for the functions.

Syntax

```
    ┌─NOARMODE─┐
►►──┴─ARMODE───┴──────────────────────────────────────────────►◄
```

Defaults
NOARMODE

Usage
If the ARMODE compiler option is specified, all functions in the compilation unit will be compiled in AR mode.

Note: If the `armode` attribute is specified on a function in a compilation unit, it overrides the compiler option.

AR mode enables a program to manipulate large amounts of data in memory by utilizing `__far` pointers. This means that a program working with a large table, for example, would not need to use temporary disk files to move the data in and out of disk storage. It also means that program logic can be less complicated, easier to maintain, and less error prone. Currently, only assembler can make use of AR Mode directly.

Note: The ARMODE compiler option is available only when the METAL option is specified. If the METAL option is not specified and the ARMODE compiler option is specified, an error message will be issued.

Predefined macros
None.

Related information
For more information on the METAL compiler option, see "METAL | NOMETAL (C only)" on page 182.

For more information on `__far` pointers, see *z/OS XL C/C++ Language Reference*.

ASCII | NOASCII

Category
Portability and migration

Pragma equivalent
None.

Purpose
Provides ASCII/NLS support.

When ASCII is in effect, the compiler performs the following:
- Uses XPLink linkage unless explicitly overwritten by the NOXPLINK option. Note that the ASCII run-time functions require XPLINK. The system headers check the __XPLINK__ macro (which is predefined when the XPLINK option is turned on). The prototypes for the ASCII run-time functions will not be exposed under NOXPLINK. Specifying the NOXPLINK option explicitly will prevent you from using the ASCII run-time functions. ASCII and NOXPLINK are accepted if the source does not contain the `main()` function, otherwise, an error is emitted.
- Uses ISO8859-1 for its default codepage rather than IBM-1047 for character constants and string literals.
- Sets a flag in the program control block to indicate that the compile unit is ASCII.

When NOASCII is in effect, the compiler uses the IBM-1047 codepage for character constants and string literals, unless the codepage is affected by other related options; for example, the CONVLIT, LOCALE, or DEF(__STRING_CODESET__) compiler options.

Syntax

```
       ┌─NOASCII─┐
►►─────┴─ASCII───┴──────────────────────────────────────────────►◄
```

Defaults

NOASCII

Usage

Use the ASCII option and the ASCII version of the run-time library if your application must process ASCII data natively at execution time.

Note: You can use EBCDIC instead of NOASCII. The two names are synonymous. There is no negative form for EBCDIC, which means that NOEBCDIC is not supported. Since EBCDIC is the default, there is usually no need to specify it. If you must specify it, use EBCDIC instead of NOASCII as the former is self-documenting.

Predefined macros

__CHARSET_LIB is defined to 1 when the ASCII compiler option is in effect and it is defined to a value of 0 when the NOASCII compiler option is in effect.

Related information

For more information on the XPLINK compiler option, see "XPLINK | NOXPLINK" on page 260.

ASMDATASIZE (C only)

Category

Object code control

Pragma equivalent

None.

Purpose

Provides the default data area size for the data areas defined by user-supplied assembly statements.

Syntax

```
►►──ASMDS──(──num──)────────────────────────────────────────────►◄
```

Defaults

ASMDATASIZE(256)

Parameters

num
   It is a positive integer number. The default value is 256.

Usage

The ASMDATASIZE compiler option can be specified only if the GENASM compiler option is in effect.

Predefined macros

__IBM_INLINE_ASM_SUPPORT is always predefined to 1.

Related information

For more information on the GENASM compiler option, see “GENASM | NOGENASM (C only)” on page 120.

ASSERT(RESTRICT) | ASSERT(NORESTRICT)

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Enables optimizations for restrict qualified pointers.

Syntax

```
                    ┌─RESTRICT───┐
►►──ASSERT──(───────┴─NORESTRICT─┴──)──────────────────────────────────►◄
```

Defaults

ASSERT(RESTRICT)

Parameters

RESTRICT
Optimizations based on restrict qualified pointers are enabled.

NORESTRICT
Optimizations based on restrict qualified pointers are disabled.

Usage

Restrict qualified pointers were introduced in the C99 Standard and provide exclusive initial access to the object that they point to. This means that two restrict qualified pointers, declared in the same scope, designate distinct objects and thus should not alias each other (in other words, they are disjoint). The compiler can use this aliasing in optimizations that may lead to additional performance gains.

Optimizations based on restrict qualified pointers will occur unless the user explicitly disables them with the option ASSERT(NORESTRICT).

ASSERT(RESTRICT) does not control whether the keyword restrict is a valid qualifier or not. Syntax checking of the restrict qualifier is controlled by the language level or KEYWORD option.

You are responsible for ensuring that if a restrict pointer *p* references an object *A*, then within the scope of *p,* only expressions based on the value of *p* are used to access *A*. A violation of this rule is not diagnosed by the compiler and may result in incorrect results. This rule only applies to ASSERT(RESTRICT).

Predefined macros

None.

Related information

For more information on related compiler options, see:

- “LANGLVL” on page 147
- “KEYWORD | NOKEYWORD (C++ only)” on page 146

ATTRIBUTE | NOATTRIBUTE (C++ only)

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Produces a compiler listing that includes the attribute component of the attribute and cross-reference section of the listing.

Syntax

```
     ┌─NOATT──────────────┐
►►───┴─ATT──┬────────────┬┴──────────────────────────────────────────────────►◄
            └─(──FULL──)─┘
```

Defaults

NOATTRIBUTE

In the z/OS UNIX System Services environment, this option is turned on by specifying -V when using the **cxx** command.

Parameters

FULL

The ATTRIBUTE(FULL) option produces a listing of all identifiers that are found in your code, even those that are not referenced.

IPA effects

During the IPA Compile step, the compiler saves symbol storage offset information in the IPA object file as follows:

- For C, if you specify the XREF, IPA(ATTRIBUTE), or IPA(XREF) options or the **#pragma options(XREF)**
- For C++, if you specify the ATTR, XREF, IPA(ATTRIBUTE), or IPA(XREF) options

If regular object code/data is produced using the IPA(OBJECT) option, the cross reference sections of the compile listing will be controlled by the ATTR and XREF options.

If you specify the ATTR or XREF options for the IPA Link step, it generates External Symbol Cross Reference and Static Map listing sections for each partition.

The IPA Link step creates a Storage Offset listing section if during the IPA Compile step you requested the additional symbol storage offset information for your IPA objects.

Predefined macros

None.

BITFIELD(SIGNED) | BITFIELD(UNSIGNED)

Category

Floating-point and integer control

Pragma equivalent

None.

Purpose

Specifies whether bit fields are signed or unsigned.

Syntax

```
                   ┌─UNSIGNED─┐
►►──BITFIELD──(──┴─SIGNED───┴──)─────────────────────────────────────►◄
```

Defaults

BITFIELD(UNSIGNED)

Parameters

SIGNED
Bit fields are signed.

UNSIGNED
Bit fields are unsigned.

Predefined macros

None.

CHARS(SIGNED) | CHARS(UNSIGNED)

Category

Floating-point and integer control

Pragma equivalent

#pragma chars

Purpose

Determines whether all variables of type `char` are treated as either `signed` or `unsigned`.

Syntax

```
                ┌─UNSIGNED─┐
►►──CHARS──(──┴─SIGNED───┴──)────────────────────────────────────────►◄
```

Defaults

CHARS(UNSIGNED)

Parameters

UNSIGNED
Variables defined as `char` are treated as `unsigned char`.

SIGNED
Variables defined as `char` are treated as `signed char`.

Predefined macros

- _CHAR_SIGNED is predefined to 1 when the CHARS(SIGNED) compiler option is in effect; otherwise it is undefined.
- _CHAR_UNSIGNED is predefined to 1 when the CHARS(UNSIGNED) compiler option is in effect; otherwise it is undefined.

CHECKOUT | NOCHECKOUT (C only)

Category

Error checking and debugging

Pragma equivalent

#pragma checkout, **#pragma options (checkout)** (C only), **#pragma options (nocheckout)** (C only)

Purpose

Produces informational messages for possible programming errors. The messages can help you to debug your C programs.

Syntax

```
     ┌─NOCHE─┐
►►───┴─CHE───┴──┬─────────────────────────┬──────────────────────────────────────►◄
                │       ┌─,──────────┐     │
                │       ▼            │     │
                └─(─────suboption────┴──)─┘
```

Defaults

NOCHECKOUT

Parameters

suboption is one of the suboptions that are shown in Table 18.

The following table lists the CHECKOUT suboptions, their abbreviations, and the messages they generate.

Note: Default CHECKOUT suboptions are underlined.

Table 18. CHECKOUT suboptions, abbreviations, and descriptions

CHECKOUT Suboption	Abbreviated Name	Description
ACCURACY | NOACCURACY	AC | NOAC	Assignments of `long` values to variables that are not `long`

Table 18. CHECKOUT suboptions, abbreviations, and descriptions (continued)

CHECKOUT Suboption	Abbreviated Name	Description
CAST | NOCAST	CA | NOCA	Potential violation of ANSI type-based aliasing rules in explicit pointer type castings. Implicit conversions, for example, those due to assignment statements, are already checked with a warning message for incompatible pointer types. See "ANSIALIAS | NOANSIALIAS" on page 66 for more information on ANSI type-based aliasing. Also see "DLL | NODLL" on page 102 for DLL function pointer casting restrictions.
ENUM | NOENUM	EN | NOEN	Usage of enumerations
EXTERN | NOEXTERN	EX | NOEX	Unused variables that have external declarations
GENERAL | NOGENERAL	GE | NOGE	General checkout messages
GOTO | NOGOTO	GO | NOGO	Appearance and usage of `goto` statements
INIT | NOINIT	I | NOI	Variables that are not explicitly initialized
PARM | NOPARM	PAR | NOPAR	Function parameters that are not used
PORT | NOPORT	POR | NOPOR	Non-portable usage of the z/OS XL C language
PPCHECK | NOPPCHECK	PPC | NOPPC	All preprocessor directives
PPTRACE | NOPPTRACE	PPT | NOPPT	Tracing of include files by the preprocessor
TRUNC | NOTRUNC	TRU | NOTRU	Variable names that are truncated by the compiler
ALL	ALL	Turns on all of the suboptions for CHECKOUT
NONE	NONE	Turns off all of the suboptions for CHECKOUT

Usage

| **Note:** As of z/OS V1R6, the INFO option is supported for both C and C++. IBM
| recommends that you use INFO instead of CHECKOUT. For information on
| using INFO as a replacement for CHECKOUT, see "INFO | NOINFO" on
| page 129.

You can specify CHECKOUT with or without suboptions. If you include suboptions, you can specify any number with commas between them. If you do not include suboptions, the compiler uses the default for CHECKOUT at your installation.

Note: If you used the CHECKOUT option and did not receive an informational message, ensure that the setting of the FLAG option is FLAG(I)

Suboptions that are specified in a **#pragma options(NOCHECKOUT(subopts))** directive, or NOCHECKOUT(subopts), apply if CHECKOUT is specified on the command line.

You can turn the CHECKOUT option off for certain files or statements of your source program by using a **#pragma checkout(suspend)** directive. Refer to *z/OS XL C/C++ Language Reference* for more information regarding this pragma directive.

Predefined macros

None.

Examples

You can specify the CHECKOUT option on the invocation line and using the **#pragma options** preprocessor directive for C. When you use both methods at the same time, the options are merged. If an option on the invocation line conflicts with an option in the **#pragma options** directive, the option on the invocation line takes precedence. The following examples illustrate these rules.

Source file:
  #pragma options (NOCHECKOUT(NONE,ENUM))

Invocation line:
  CHECKOUT (GOTO)

Result:
  CHECKOUT (NONE,ENUM,GOTO)

Source file:
  #pragma options (NOCHECKOUT(NONE,ENUM))

Invocation line:
  CHECKOUT (ALL,NOENUM)

Result:
  CHECKOUT (ALL,NOENUM)

Related information

See the “INFO | NOINFO” on page 129 compiler option section, for information on C++ support for similar functionality.

CICS | NOCICS

Category

Language element control

Pragma equivalent

None.

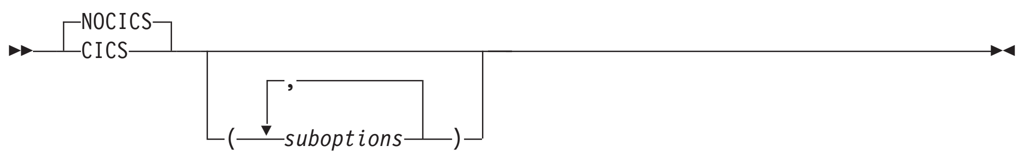
Purpose

Enables CICS statements to be embedded in C/C++ source and passes them through the compiler without the need for an explicit preprocessing step.

When the CICS option is in effect, the compiler can pass suboptions to the integrated CICS translator. CICS suboptions are passed directly to the CICS translator and have no other effect on compilation of C/C++ source.

When the NOCICS option is in effect, the compiler will treat the CICS-specific keywords as normal identifiers.

Syntax



Defaults

NOCICS

Parameters

For more information on CICS suboptions, refer to *CICS Application Programming Guide*.

Usage

Integrated CICS translation enables you to embed CICS statements in C/C++ source and pass them through the compiler without the need for an explicit preprocessing step. This permits a more seamless operation of C/C++ within the CICS environment, especially under z/OS UNIX System Services, and may help with program readability and application maintenance. Comments and macros are also permitted within embedded CICS commands. Integrated CICS translation is supported for use with CICS Transaction Server for z/OS V3R1 and above.

The CICS compiler option must be used when compiling source containing embedded CICS statements.

The CICS option will appear in the options listing with the suboptions.

The #pragma XOPTS directive may also be used to pass options to the integrated CICS translator. #pragma XOPTS is not a pragma equivalent to the CICS compiler option; therefore, the CICS suboptions passed via #pragma XOPTS will be recognized only when the CICS option is specified. Refer to *z/OS XL C/C++ Language Reference* for more information regarding this pragma directive.

A CICS embedded command will take the form of: EXEC CICS_COMMAND_KEYWORD *xxxx*; where:

- EXEC is a context-sensitive keyword. If the token following it is not recognized as a CICS keyword, then it is treated as part of the user name space.
- CICS_COMMAND_KEYWORD is either CICS, DLI, or CPSM. CICS, DLI, and CPSM are context-sensitive keywords and have special meaning only in the EXEC statement. These tokens may be used by the application program in user-defined names.
- *xxxx* is a command appropriate to the CICS_COMMAND_KEYWORD.

All text from EXEC up to the first semicolon will be processed by the CICS translator after C/C++ preprocessing. The command may span multiple lines.

Note: All keywords are case-insensitive, which means any combination of upper and lower-case characters may be used.

A CICS embedded command is expanded into a block statement and therefore can occur only at points in the code where a block statement is allowed.

A CICS embedded keyword will have the form of:
CICS_KEYWORD(CICS_KEYWORD_VALUE) where:

- CICS_KEYWORD is either DFHVALUE, DFHRESP, or EYUVALUE. These are reserved keywords and cannot be used in any other context. These keywords are case-insensitive.
- CICS_KEYWORD_VALUE is a value that is appropriate to the CICS_KEYWORD.

The compiler will send the entire string to the CICS translator after macro substitution and all comments are stripped out.

The following items are permitted within both CICS embedded commands and CICS embedded keywords:

- C/C++ comments
- C/C++ macros
- `#if` directives
- `#include` directives

Use of all other preprocessor directives, including all `#pragma` directives, will result in undefined behavior. The comments will be stripped, and macros and permitted directives will be expanded before the command or keyword is sent to the CICS translator.

When the CICS option is specified with the PPONLY option, both CICS embedded commands and CICS embedded keywords will be preserved after all preprocessor macro substitution. #pragma XOPTS will be preserved as well.

Notes:

1. The compiler will not check compiler options for compatibility with CICS. You can migrate to a later version of CICS without upgrading the compiler and still take advantage of previously incompatible features. You need to ensure that you have the required level of CICS Transaction Server on the target machines because the compiler does not check the TARGET option.
2. The compiler will not check for compatibility between pre-translators. The compiler will allow multiple pre-translators to operate on a single source file; for example, EXEC CICS statements may be intermixed with EXEC SQL statements. You must ensure that this is semantically correct.

Predefined macros

__CICS__ is predefined to 1 when the CICS compiler option is in effect; otherwise, it is not defined.

COMPACT | NOCOMPACT

Category

Optimization and tuning

Pragma equivalent

#pragma option_override(subprogram_name, ″OPT(COMPACT)″)

Purpose

Avoids optimizations that increase object file size.

When the COMPACT option is in effect, the compiler favours those optimizations which tend to limit object file size.

When the NOCOMPACT option is in effect, the compiler may use optimizations which result in an increased object file size.

Syntax

```
     ┌─NOCOMPACT─┐
►►───┴─COMPACT───┴──────────────────────────────────────────────────►◄
```

Defaults

NOCOMPACT

Usage

During optimizations performed as part of code generation, for both NOIPA and IPA, choices must be made between those optimizations which tend to result in faster but larger code and those which tend to result in smaller but slower code. The COMPACT option influences these choices.

Because of the interaction between various optimizations, including inlining, code compiled with the COMPACT option may not always generate smaller code and data. To determine the final status of inlining, generate and check the inline report. Not all subprograms are inlined when COMPACT is specified.

To evaluate the use of the COMPACT option for your application:

- Compare the size of the objects generated with COMPACT and NOCOMPACT
- Compare the size of the modules generated with COMPACT and NOCOMPACT
- Compare the execution time of a representative workload with COMPACT and NOCOMPACT

If the objects and modules are smaller with an acceptable change in execution time, then you can consider the benefit of using COMPACT.

As new optimizations are added to the compiler, the behavior of the COMPACT option may change. You should re-evaluate the use of this option for each new release of the compiler and when you change the application code.

IPA effects

During a compilation with IPA Compile-time optimizations active, any subprogram-specific COMPACT option specified by **#pragma option_override(subprogram_name, ″OPT(COMPACT)″)** directives will be retained.

The IPA Compile step generates information for the IPA Link step. This option also affects the regular object module if you request one by specifying the IPA(OBJECT) option.

If you specify the COMPACT option for the IPA Link step, it sets the compilation unit values of the COMPACT option that you specify. The IPA Link step Prolog listing section will display the value of this option.

If you do not specify COMPACT option in the IPA Link step, the setting from the IPA Compile step for each compilation unit will be used.

In either case, subprogram-specific COMPACT options will be retained.

The IPA Link step merges and optimizes your application code, and then divides it into sections for code generation. Each of these sections is a partition. The IPA Link step uses information from the IPA Compile step to determine if a subprogram can be placed in a particular partition. Only compatible subprograms are included in a given partition. Compatible subprograms have the same COMPACT setting.

The COMPACT setting for a partition is set to the specification of the first subprogram that is placed in the partition. Subprograms that follow are placed in partitions that have the same COMPACT setting. A NOCOMPACT subprogram is placed in a NOCOMPACT partition, and a COMPACT subprogram is placed in a COMPACT partition.

The option value that you specified for each IPA object file on the IPA Compile step appears in the IPA Link step Compiler Options Map listing section.

The Partition Map sections of the IPA Link step listing and the object module END information section display the value of the COMPACT option. The Partition Map also displays any subprogram-specific COMPACT values.

Predefined macros

None.

COMPRESS | NOCOMPRESS

Category

Object code control

Pragma equivalent

None.

Purpose

Suppresses the generation of function names in the function control block, thereby reducing the size of your application's load module.

Syntax

```
      ┌─NOCOMPRESS─┐
►►────┴─COMPRESS───┴──────────────────────────────────────────────────►◄
```

Defaults

NOCOMPRESS

Usage

Function names are used by the dump service to provide you with meaningful diagnostic information when your program encounters a fatal program error. They are also used by tools such as Debug Tool and the Performance Analyzer. Without these function names, the reports generated by these services and tools may not be complete.

If COMPRESS and TEST or DEBUG are in effect at the same time, the compiler issues a warning message and ignores the COMPRESS option.

IPA effects

The IPA Compile step generates information for the IPA Link step. COMPRESS also affects the regular object module if you request one by specifying the IPA(OBJECT) option.

If you specify the COMPRESS option for the IPA Link step, it uses the value of the option that you specify. The IPA Link step Prolog listing section will display the value of the option that you specify.

If you do not specify COMPRESS option in the IPA Link step, the setting from the IPA Compile step will be used.

The IPA Link step merges and optimizes your application code, and then divides it into sections for code generation. Each of these sections is a partition. The IPA Link step uses information from the IPA Compile step to determine if a subprogram can be placed in a particular partition. Only compatible subprograms are included in a given partition. Compatible subprograms have the same COMPRESS setting.

The COMPRESS setting for a partition is set to the specification of the first subprogram that is placed in the partition. Subprograms that follow are placed in partitions that have the same COMPRESS setting. A NOCOMPRESS mode subprogram is placed in a NOCOMPRESS partition, and a COMPRESS mode subprogram is placed in a COMPRESS partition.

The option value that you specified for each IPA object file on the IPA Compile step appears in the IPA Link step Compiler Options Map listing section.

The Partition Map sections of the IPA Link step listing and the object module `END` information section display the value of the COMPRESS option.

Predefined macros

None.

Related information

For more information on related compiler options, see:

- "TEST | NOTEST" on page 249
- "DEBUG | NODEBUG" on page 94

CONVLIT | NOCONVLIT

Category

Portability and migration

Pragma equivalent

#pragma convlit

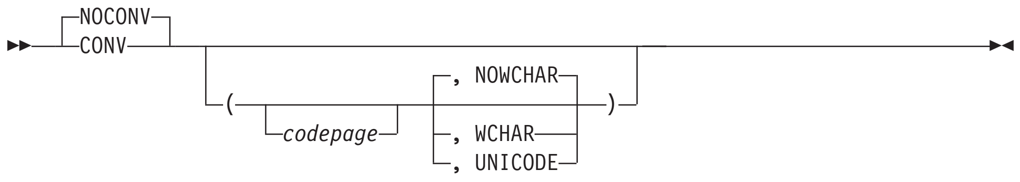
Purpose

Turns on string literal codepage conversion.

When the CONVLIT option is in effect, the compiler changes the assumed codepage for character and string literals within the compilation unit.

When the NOCONVLIT option is in effect (or CONV without a suboption), the default codepage, or the codepage specified by the LOCALE option is used.

Syntax



Defaults

NOCONVLIT(, NOWCHAR)

Parameters

codepage
: You can use an optional suboption to specify the *codepage* that you want to use for string literals.

NOWCHAR
: The default is NOWCHAR. Only wide character constants and string literals made up of single byte character set (SBCS) characters are converted. If there are any shift-out (SO) and shift-in (SI) characters in the literal, the compilation will end with an error message.

WCHAR
: Instructs the compiler to change the codepage for wide character constants and string literals declared with the L'' or L"" prefix. Although optional, the WCHAR suboption is positional, and must appear as the second suboption to the CONVLIT option.

UNICODE
: The z/OS XL C/C++ compiler interprets the CONVLIT(, UNICODE) suboption as a request to convert the wide string literals and wide character constants (`wchar_t`) to Unicode (UCS-2) regardless of the code page used for conversion of string literals and character constants (`char`). The conversion is supported for wide string literals and wide character constants that are coded using characters from the basic character set defined by the *Programming languages - C (ISO/IEC 9899:1999)* standard. The behavior is undefined if wide string literals and wide character constants are coded using characters outside the basic character set.

Usage

The CONVLIT option affects all the source files that are processed within a compilation unit, including user header files and system header files. All string literals and character constants within a compilation unit are converted to the specified *codepage* unless you use **#pragma convlit(suspend)** and **#pragma convlit(resume)** to exclude sections of code from conversion. See *z/OS XL C/C++ Language Reference* for more information on **#pragma convlit**.

The CONVLIT option only affects string literals within the compilation unit. The following determines the codepage that the rest of the program uses:

- If you specified a LOCALE, the remainder of the program will be in the codepage that you specified with the LOCALE option.
- If you did not specify a LOCALE, the remainder of the program will be in the default codepage IBM-1047.

The CONVLIT option does not affect the following types of string literals:

- literals in the `#include` directive
- literals in the `#pragma` directive
- literals used to specify linkage, for example, `extern "C"`
- literals used for the __`func`__ variables

The semantics of the **#pragma convlit(suspend/resume)** are not affected by this extension. No string literals or character constants (wide included), will be converted if **#pragma convlit(suspend)** is in effect.

The interaction between the **#pragma convlit(suspend/resume)** and **#pragma convert** is not affected by this extension. String literals and character constants will continue to be converted by the currently active **#pragma convert**. Wide string literals and wide character constants are not affected by **#pragma convert**, even when the new CONVLIT(, UNICODE) suboption is specified.

If you specify PPONLY with CONVLIT, the compiler ignores CONVLIT.

If you specify the CONVLIT option, the *codepage* appears after the locale name and locale code set in the Prolog section of the listing. The option appears in the END card at the end of the generated object module.

Notes:

1. Although you can continue to use the __STRING_CODE_SET__ macro, you should use the CONV option instead. If you specify both the macro and the option, the compiler diagnoses it and uses the option regardless of the order in which you specify them
2. The **#pragma convert** directive provides similar functionality to the CONVLIT option. It has the advantage of allowing more than one character encoding to be used for string literals in a single compilation unit. For more information on the **#pragma convert** directive, see *z/OS XL C/C++ Language Reference*.

IPA effects

The CONVLIT option only controls processing for the IPA step for which you specify it.

During the IPA Compile step, the compiler uses the code page that is specified by the CONVLIT option to convert the character string literals.

Predefined macros

None.

Examples

The result of the following specifications is the same:

- `NOCONV(IBM-1027) CONV`
- `CONV(IBM-1027)`

Related information

For more information on the LOCALE compiler option, see “LOCALE | NOLOCALE” on page 165.

CSECT | NOCSECT

Category

Object code control

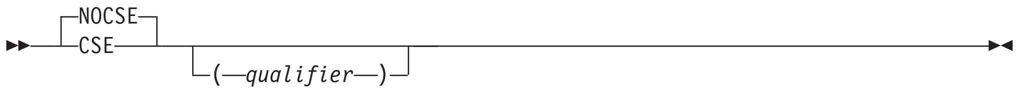
Pragma equivalent

#pragma csect

Purpose

Instructs the compiler to generate CSECT names in the output object module.

Syntax



Defaults

For NOGOFF, the default option is NOCSECT. For GOFF, the default option is CSECT().

Parameters

qualifier
    Enables the compiler to generate long CSECT names.

Usage

When the CSECT option is in effect, the compiler should ensure that the code, static data, and test sections of your object module are named. Use this option if you will be using SMP/E to service your product and to aid in debugging your program.

For C, when you specify CSECT(*qualifier*) and the NOGOFF option is in effect, the LONGNAME option is assumed.

For GOFF, both the NOLONGNAME and LONGNAME options are supported.

The CSECT option names sections of your object module differently depending on whether you specified CSECT with or without a qualifier.

If you specify the CSECT option without the *qualifier* suboption, the CSECT option names the code, static data, and test sections of your object module as *csectname*, where *csectname* is one of the following:
- The member name of your primary source file, if it is a PDS member
- The low-level qualifier of your primary source file, if it is a sequential data set
- The source file name with path information and the right-most extension information removed, if it is a z/OS UNIX System Services file.
- For NOGOFF and for C only, if the NOLONGNAME option is in effect, then the *csectname* is truncated to 8 characters long starting from the left. For GOFF, the full *csectname* is always used. For NOGOFF and for C++ only, the *csectname* is always truncated to 8 characters long starting from the left.

code CSECT	Is named with *csectname* name in uppercase.
data CSECT	Is named with *csectname* in lower case.
test CSECT	When you use the TEST option together with the CSECT option, the debug information is placed in the test CSECT. The test CSECT is the static CSECT name with the prefix $. If the static CSECT name is 8 characters long, the right-most character is dropped and the compiler issues an informational message except in the case of GOFF. The test CSECT name is always truncated to 8 characters.

For example, if you compile `/u/cricket/project/mem1.ext.c`:

- with the options NOGOFF and CSECT, the test CSECT will have the name `$mem1.ex`
- with the options GOFF and CSECT, the test CSECT will have the name `$mem1.ext`

If you specify the CSECT option with the *qualifier* suboption, the CSECT option names the code, static data, and test sections of your object module as *qualifier#basename#suffix*, where:

qualifier
: Is the suboption you specified as a *qualifier*

basename
: Is one of the following:
- The member name of your primary source file, if it is a PDS member
- There is no basename, if your primary source file is a sequential data set or instream JCL
- The source file name with path information and the right-most extension information removed, if it is a z/OS UNIX file

suffix
: Is one of the following:

C	For code CSECT
S	For static CSECT
T	For test CSECT

Notes:

1. If the *qualifier* suboption is longer than 8 characters, you must use the binder.
2. The *qualifier* suboption takes advantage of the capabilities of the binder, and may not generate names acceptable to the Language Environment Prelinker.
3. The # that is appended as part of the #C, #S, or #T suffix is not locale-sensitive.
4. The string that is specified as the *qualifier* suboption has the following restrictions:
   - Leading and trailing blanks are removed
   - You can specify a string of any length. However if the complete CSECT name exceeds 1024 bytes, it is truncated starting from the left.
5. If the source file is either sequential or instream in your JCL, you must use the **#pragma csect** directive to name your CSECT. Otherwise, you may receive an error message at bind time.

The CSECT names for all the sections (including the code, static data and test sections) must conform to the following rules:

- The first character must be an alphabetic character. An alphabetic character is a letter from A through Z, or from a through z, or _, $(code point `X'5B'`), #(code point `X'7B'`) or @(code point `X'7C'`). The other characters in the CSECT name may be alphabetic characters, digits, or a combination of the two.
- No other special characters may be included in the CSECT name.
- No spaces are allowed in the CSECT name.
- No double-byte data is allowed in the CSECT name.

IPA effects

For the IPA Link step, this option has the following effects:

1. If you specify the CSECT option, the IPA Link step names all of the CSECTs that it generates.

The IPA Link step determines whether the IPA Link control file contains CSECT name prefix directives. If you did not specify the directives, or did not specify enough CSECT entries for the number of partitions, the IPA Link step automatically generates CSECT name prefixes for the remaining partitions, and issues an error diagnostic message each time.

The form of the CSECT name that IPA Link generates depends on whether the CSECT or CSECT(*qualifier*) format is used.

2. If you do not specify the CSECT option, but you have specified CSECT name prefix directives in the IPA Link control file, the IPA Link step names all CSECTs in a partition. If you did not specify enough CSECT entries for the number of partitions, the IPA Link step automatically generates a CSECT name prefix for each remaining partition, and issues a warning diagnostic message each time.
3. If you do not specify the CSECT option, and do not specify CSECT name prefix directives in the IPA Link control file, the IPA Link step does not name the CSECTs in a partition.

The IPA Link step ignores the information that is generated by **#pragma csect** on the IPA Compile step.

Predefined macros

None.

Examples

For example, if you compile `/u/cricket/project/mem1.ext.c` with the options TEST and CSECT(example), the compiler constructs the CSECT names as follows:

```
example#mem1.ext#C
example#mem1.ext#S
example#mem1.ext#T
```

The *qualifier* suboption of the CSECT option allows the compiler to generate long CSECT names.

For example, if you compile `/u/cricket/project/reallylongfilename.ext.c` with the options TEST and CSECT(example), the compiler constructs the CSECT names as follows:

```
example#reallylongfilename.ext#C
example#reallylongfilename.ext#S
example#reallylongfilename.ext#T
```

When you specify CSECT(*qualifier*), the code, data, and test CSECTs are always generated. The test CSECT has content only if you also specify the TEST option.

If you use `CSECT("")` or `CSECT()`, the CSECT name has the form *basename#suffix*, where *basename* is:

- `@Sequential@` for a sequential data set
- `@InStream@` for instream JCL

CVFT | NOCVFT (C++ only)

Category

Object code control

Pragma equivalent

None.

Purpose
Shrinks the size of the writeable static area (WSA) by eleminating the construction virtual function tables (CVFT), which in turn may reduce the load module size to improve your application's performance.

When the CVFT option is in effect, the compiler doesn't shrink the size of the WSA.

NOCVFT prevents constructors from tracking which virtual function to call at different stages of the construction process. Only constructors that call virtual functions within a class hierarchy that uses virtual inheritance or does base address calculation are affected. Use NOCVFT if none of the constructors in your application call virtual functions from within the class hierarchy or you don't cast to another subobject's base, either directly or indirectly.

Syntax

```
     ┌─CVFT───┐
►►───┴─NOCVFT─┴──────────────────────────────────────────────────────►◄
```

Defaults
CVFT

Usage
The CVFT option is shown on the listing prolog and the text deck end card.

IPA effects
The IPA Link step issues a diagnostic message if you specify the CVFT option for that step.

Predefined macros
None.

DBRMLIB

Category
Compiler output

Pragma equivalent
None.

Purpose
Provides the location for the database request module used in conjunction with the SQL option.

Syntax

```
►►──DBRM──┬─────────────────────────────────────────────┬────────────►◄
          └─(──┬─// Partitioned data set─────────────┬─)─┘
               ├─// Partitioned data set (member)────┤
               ├─z/OS UNIX System Services filename──┤
               └─z/OS UNIX System Services directory─┘
```

Defaults
DBRMLIB(DD:DBRMLIB)

Parameters

Partitioned data set
: Specifies the partitioned data set for the database request module. It must be either a relative data set name, or an absolute data set name enclosed in single quotation marks. In either case, it must also be prepended by //.

Partitioned data set (member)
: Specifies the partitioned data set (member) for the database request module. It must be prepended by //.

z/OS UNIX System Services filename
: Specifies the z/OS UNIX System Services file name for the database request module.

z/OS UNIX System Services directory
: Specifies the z/OS UNIX System Services directory for the database request module.

Usage

When the DBRMLIB option is in effect, the compiler specifies the output for the database request module (DBRM), which is generated by the SQL option. The DBRM output contains the embedded SQL statements and host variable information extracted from the source program, information that identifies the program, and ties the DBRM to the translated source statements. It becomes the input to the DB2 bind process.

Note: The DBRMLIB option can only be specified when the SQL option is also specified.

As of z/OS V1R9, the compiler has been extended to support PDS and z/OS UNIX directory compiles with the SQL option, making it possible to compile all members of a PDS or all files in a z/OS UNIX directory in one single JCL job step.

Predefined macros

None.

Examples

If you do not specify a file name for the DBRMLIB option, the compiler generates a file name as follows:

- If you are calling the compiler from a JCL, the compiler uses the source file name to form the name of the DBRM data set. The high-level qualifier is replaced with the userid under which the compiler is running, and .DBRM is appended as the low-level qualifier. For example, with the User ID “USER01”:
  - If you compile a source file //’USER01.PDS.C(FOO)’, the generated name of the DBRM file would be //’USER01.PDS.C.DBRM(FOO)’.
  - If you compile a source file //’USER01.SEQ.C’, the generated name of the DBRM file would be //’USER01.SEQ.C.DBRM’.

    Note: The DB2 SQL coprocessor may not support sequential data sets.
  - If you compile a source file `/home/user01/foo.c`, the generated DBRM file would be `./foo.dbrm`.
- If you are calling the compiler from z/OS UNIX System Services and using the xlc utility, the compiler stores the DBRM output in a file that is based on the source file name. For example, if compiling with the User ID “USER01”:
  - If you compile a source file //’USER01.PDS.C(FOO)’, the generated name of the DBRM file would be //’USER01.PDS.DBRM(FOO)’.

– If you compile a source file //'USER01.SEQ.C', the generated name of the DBRM file would be //'USER01.SEQ.DBRM'.
– If you compile a z/OS UNIX file `/home/user01/foo.c`, the generated DBRM file would be `./foo.dbrm`.

- Like xlc, the c89 utility always generates a default destination name based on the source file name. By default, the behavior is the same as in the xlc case. However, this can be changed by setting _OSUFFIX_HOSTQUAL and _OSUFFIX_HOSTRULE to 0 which yields the same behavior as in the JCL case.

If you do not specify the DBRMLIB compiler option at all (in combination with the SQL compiler option), an empty DBRMLIB compiler option (without a file name specified) will be implicitly assumed by the compiler. If you have explicitly specified a DBRMLIB file using the DBRMLIB DD, the DBRMLIB file name specified in the DD will be used.

The DBRM file is considered as output from the compiler. For further details on valid input and output file combinations, refer to "Output from the compiler" on page 336.

When the DBRMLIB compiler option is specified in JCL, and a DBRMLIB DD statement is also specified, the option will take precedence over the DD statement.

The compiler does not verify the DCB attributes of the data set; you must ensure the data set is created with the correct attributes, as expected by DB2 Universal Database. Refer to *DB2 for z/OS Application Programming and SQL Guide* at http://www.ibm.com/software/data/db2/zos/library.html for details.

DEBUG | NODEBUG

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Instructs the compiler to generate debug information.

Syntax

```
                                   ┌─DWARF─┐
     ┌─DEBUG───┐   ┌─(──FORMAT──(──┴─ISD───┴──)──)─┐
►►───┴─NODEBUG─┴───┴───────────────────────────────┴──────────────────────────────►
```

```
►─┬──────────────────────────────────────────────────────────────┬──────────►◄
  └─(──┬─FILE──(──┬───────────────────────────────┬──)──┬──)─┘
       │          └─┬─Sequential data set───────────┘     │
       │            ├─Partitioned data set─────────┤      │
       │            ├─Partitioned data set (member)─┤     │
       │            ├─Path─────────────────────────┤      │
       │            └─Path name────────────────────┘      │
       ├─LEVEL──(──0──)───────────────────────────────────┤
       ├─HOOK │ NOHOOK──┬─────────────────────────────┬───┤
       │                │   ┌─,─────────────┐         │   │
       │                │   ▼               │         │   │
       │                └─(───┬─LINE │ NOLINE────┬──┴─)─┘   │
       │                      ├─BLOCK │ NOBLOCK─┤           │
       │                      ├─PATH │ NOPATH───┤           │
       │                      ├─FUNC │ NOFUNC───┤           │
       │                      ├─CALL │ NOCALL───┤           │
       │                      ├─NONE────────────┤           │
       │                      ├─ALL─────────────┤           │
       │                      └─PROFILE─────────┘           │
       └─SYMBOL │ NOSYMBOL─────────────────────────────────┘
```

Defaults

- NODEBUG
- For FORMAT, the default is DWARF.
- For LEVEL, the default is LEVEL(0).
- For HOOK, the defaults are HOOK(ALL) for NOOPTIMIZE and HOOK(NONE,PROFILE) for OPTIMIZE.
- For SYMBOL, the defaults are SYMBOL for NOOPTIMIZE and NOSYMBOL for OPTIMIZE.
- When METAL is specified, the default is NOHOOK.

Parameters

FORMAT

Has the following suboptions: ISD and DWARF. ISD produces the same debug information as the TEST option. This suboption is available only with ILP32. If this format is used, the FILE suboption is ignored.

The DWARF suboption produces debug information in the DWARF Version 3 debugging information format, stored in the file specified by the FILE suboption. This is the only format supported when LP64 or METAL is specified.

FILE

Specifies the name of the output file for FORMAT(DWARF). It can be a *sequential data set*, a *partitioned data set*, a *partitioned data set (member)* , a *z/OS UNIX System Services file*, or a *z/OS UNIX System Services directory*.

If you do not specify a file name, the compiler uses the SYSCDBG DD statement, or its alternative, if you allocated it. Otherwise, the compiler constructs a file name as follows:

- If you are compiling a data set, the compiler uses the source file name to form the name of the output data set. The high-level qualifier is replaced with the userid under which the compiler is running, and .DBG is appended as the low-level qualifier.

- If you are compiling a z/OS UNIX file, the compiler stores the debug information in a file that has the name of the source file with a `.dbg` extension.

For example, if `TSYOU19` is compiling `TSPERF.EON.SOURCE(EON)` with the DEBUG option and does not specify a file name, the default output file name will be `TSYOU19.EON.SOURCE.DBG(EON)`.

For a PDS or z/OS UNIX file system directory compile, the FILE option specifies the PDS or z/OS UNIX file system directory where the output files are generated.

The default for **c89** is `FILE(./filename.dbg)`.

The compiler resolves the full path name for this file name, and places it in the generated object file. This information can be used by program analysis tools to locate the output file for FORMAT(DWARF). You can examine this generated file name in the compiler listing file (see “LIST | NOLIST” on page 163 for instructions on how to create a compiler listing file), as shown in the following example:

```
                        PPA4: Compile Unit Debug Block
000140  0000001A                                 =F'26'             DWARF File Name
000144  ****                                     C'/hfs/fullpath/filename.dbg'
```

If the compiler cannot resolve the full path name for the file name (for example, because the search permission was denied for a component of the file name), the compiler will issue a warning message, and the relative file name will be used instead.

Note: DEBUG(FILE(filename)) is not supported when the METAL compiler option is specified.

LEVEL

Controls the amount of debug information produced. LEVEL(0) is the only level currently supported.

HOOK

Notes:

1. A METAL compilation does not generate hook instructions, therefore DEBUG(HOOK) is not supported when the METAL compiler option is specified.
2. If the OPTIMIZE compiler option is specified, the only valid suboptions for HOOK are CALL and FUNC. If other suboptions are specified, they will be ignored.

Controls the generation of LINE, BLOCK, PATH, CALL, and FUNC hook instructions. Hook instructions appear in the compiler Pseudo Assembly listing in the following form:

```
EX r0,HOOK..[type of hook]
```

The type of hook that each hook suboption controls is summarized in the list below:

- LINE
  - STMT - General statement
- BLOCK
  - BLOCK-ENTRY - Beginning of block
  - BLOCK-EXIT - End of block

  - MULTIEXIT - End of block and procedure
- PATH
  - LABEL - A label
  - DOBGN - Start of a loop
  - TRUEIF - True block for an if statement
  - FALSEIF - False block for an if statement
  - WHENBGN - Case block
  - OTHERW - Default case block
  - GOTO - Goto statement
  - POSTCOMPOUND - End of a PATH block
- CALL
  - CALLBGN - Start of a call sequence
  - CALLRET - End of a call sequence
- FUNC
  - PGM-ENTRY - Start of a function
  - PGM-EXIT - End of a function

There is also a set of shortcuts for specifying a group of hooks:

NONE It is the same as specifying NOLINE, NOBLOCK, NOPATH, NOCALL, and NOFUNC. It instructs the compiler to suppress all hook instructions.

ALL It is the same as specifying LINE, BLOCK, PATH, CALL, and FUNC. It instructs the compiler to generate all hook instructions. This is the ideal setting for debugging purposes.

PROFILE
It is the same as specifying CALL and FUNC. It is the ideal setting for tracing the program with the Performance Analyzer.

SYMBOL

Generates symbol information that gives you access to variable and other symbol information.

Usage

When the DEBUG option is in effect, the compiler generates debug information based on the DWARF Version 3 debugging information format, which has been developed by the UNIX International Programming Languages Special Interest Group (SIG), and is an industry standard format.

Note: Starting with z/OS V1R5, the Performance Analyzer only works with DEBUG(FORMAT(ISD)). This also means that it will not work on 64-bit executables.

If you specify OPTIMIZE and DEBUG(FORMAT(DWARF)), no symbolic debug information is generated, but function entry, function exit, function call and function return hooks are generated. If you specify OPTIMIZE and DEBUG(FORMAT(ISD)), the behavior is the same as OPTIMIZE and TEST.

If you specify the INLINE and DEBUG compiler options when NOOPTIMIZE is in effect, INLINE is ignored.

You can specify the DEBUG option and TARGET to a release prior to z/OS V1R5. However, if the debug format is DWARF, you must debug using dbx on a z/OS V1R5 (and above) system.

In the z/OS UNIX System Services environment, **-g** implies DEBUG.

IPA effects

For the IPA Compile step, you can specify all of the DEBUG suboptions that are appropriate for the language of the code that you are compiling. However, they affect processing only if you have requested code generation, and only the conventional object file is affected. If you specify the NOOBJECT suboption of the IPA compiler option on the IPA Compile step, the IPA Compile step ignores the DEBUG option, even though the listing still indicates that the DEBUG option has been specified.

The IPA Link step only supports the generation of function calls, entry, exit, and return hooks for profiling. If you specify any other DEBUG suboptions for the IPA Link step, it turns them off and issues a warning message. If IPA(OBJONLY) object files from the IPA Compile step are specified in the IPA Link step, only hooks for profiling are supported in the input object file.

Note: For an IPA-optimized application, the DEBUG option inserts additional instructions (referred to as hooks) into the application for profiling. This is different from the IPA Profile Directed Feedback option, which is used for tuning optimizations. During the compilation phase, in addition to the existing IPA compilation options, you must specify DEBUG(HOOK(NONE,PROFILE), NOSYMBOL). During the IPA Link phase, in addition to the existing IPA Link options, you must specify DEBUG(HOOK(NONE,PROFILE)). Use of these options can affect the performance of your routine. You may need to remove the options and recompile your routine before delivering your application. See the “TEST | NOTEST” on page 249 for more information on debugging applications linked with IPA.

Predefined macros

None.

Examples

If you specify both TEST and DEBUG, the last valid specification is used. If you specify DEBUG and NODEBUG multiple times, the compiler uses the last specified option with the last specified suboption. For example, the following specifications have the same result:

```
cc -Wc,"NODEBUG(FORMAT(DWARF),HOOK(ALL))" -Wc,"DEBUG(NOSYMBOL)" hello.c
cc -WC,"DEBUG(FORMAT(DWARF),HOOK(ALL),NOSYMBOL)" hello.c
```

DEFINE

Category

Language element control

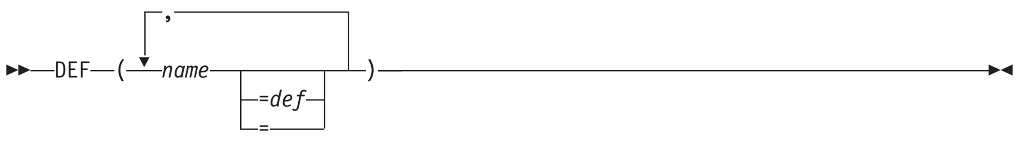
Pragma equivalent

None.

Purpose

Defines a macro as in a `#define` preprocessor directive.

Syntax



Defaults

No default user definitions.

For the z/OS UNIX System Services **c99**, **c89**, and **c++** commands, the default for a regular compile is:

```
DEFINE(errno=(*__errno()))

DEFINE(_OPEN_DEFAULT=1)
```

For the z/OS UNIX System Services **cc** command, the default for a regular compile is:

```
DEFINE(errno=(*__errno()))

DEFINE(_OPEN_DEFAULT=0)

DEFINE(_NO_PROTO=1)
```

Parameters

DEFINE(*name*)
: Is equal to the preprocessor directive `#define` *`name`* `1`.

DEFINE(*name*=*def*)
: Is equal to the preprocessor directive `#define` *`name def`*.

DEFINE(*name*=)
: Is equal to the preprocessor directive `#define` *name*.

Usage

When the DEFINE option is in effect, the preprocessor macros that take effect before the compiler processes the file are defined.

You can use the DEFINE option more than once.

If the suboptions that you specify contain special characters, see "Using special characters" on page 44 for information on how to escape special characters.

In the z/OS UNIX System Services environment, you can unset variables specified by -D, or automatically specified by **c89**, using -U when using the **c89**, **cc**, or **c++** commands.

Note: **c89** preprocesses -D and -U flags before passing them onto the compiler. **xlc** just passes -D and -U to the compiler, which interprets them as DEFINE and UNDEFINE. For more information, see Chapter 21, "c89 — Compiler invocation using host environment variables," on page 535 or Chapter 24, "xlc — Compiler invocation using a customizable configuration file," on page 577.

Predefined macros

To use the __STRING_CODE_SET__ macro to change the code page that the compiler uses for character string literals, you must define it with the DEFINE compiler option; for example:

```
DEFINE(__STRING_CODE_SET__="ISO8859-1")
```

Examples

Note: There is no command-line equivalent for function-like macros that take parameters such as the following:

```
#define max(a,b)  ((a)>(b)?(a):(b))
```

DFP | NODFP

Category

Floating-point and integer control

Pragma equivalent

None.

Purpose

Provides support for decimal floating-point types.

When the DFP option is in effect, decimal floating-point support is enabled.

When the NODFP option is in effect, decimal floating-point support is disabled, which includes disabling the `_Decimal32`, `_Decimal64`, and `_Decimal128` keywords.

Syntax

```
      ┌─NODFP─┐
►►────┴─DFP───┴──────────────────────────────────────────────────►◄
```

Defaults

NODFP

Usage

The decimal floating-point format assists with avoiding potential rounding problems, which can result from using binary or hexadecimal floating-point types to handle decimal calculations.

When DFP is enabled the following decimal type specifiers are supported:

- `_Decimal32`
- `_Decimal64`
- `_Decimal128`

For further information on these reserved keywords, decimal literal support, and decimal floating-point type conversions, see *z/OS XL C/C++ Language Reference*.

Note: The DFP option can only be used with ARCH values greater than or equal to 7.

Predefined macros

__IBM_DFP__ is predefined to 1 when the DFP compiler option is in effect.

DIGRAPH | NODIGRAPH

Category
Language element control

Pragma equivalent
None.

Purpose
Enables recognition of digraph key combinations or keywords to represent characters not found on some keyboards.

Note: A *digraph* is a combination of keys that produces a character that is not available on some keyboards.

Syntax

```
      ┌─DIGR───┐
►►────┴─NODIGR─┴──────────────────────────────────────────────────────►◄
```

Defaults
DIGRAPH

Usage
Table 19 shows the digraphs that z/OS XL C/C++ supports:

Table 19. Digraphs

Key Combination	Character Produced
<%	{
%>	}
<:	[
:>	]
%:	#
%%[1]	#
%:%:	##
%%%%[1]	##

Table 20 shows additional keywords that z/OS XL C++ supports:

Table 20. Additional keywords

Keyword	Characters produced
bitand	&
and	&&
bitor	|
or	||
xor	^
compl	~

1. The digraphs %% and %%%% are not digraphs in the C Standard. For compatibility with z/OS XL C++, however, they are supported by z/OS XL C. Use the %: and %:%: digraphs instead of %% and %%%% whenever possible.

Table 20. Additional keywords (continued)

Keyword	Characters produced
and_eq	&=
or_eq	|=
xor_eq	^=
not	!
not_eq	!=

IPA effects

The IPA Link step issues a diagnostic message if you specify the DIGRAPH option on that step.

Predefined macros

__DIGRAPHS__ is predefined to 1 when the DIGRAPH compiler option is in effect.

Examples

Note: Digraphs are not replaced in string literals, comments, or character literals. For example:

```
char * s = "<%%>";   // stays "<%%>"

switch (c) {
  case '<%' : ...    // stays '<%'
  case '%>' : ...    // stays '%>'
}
```

Related information

See *z/OS XL C/C++ Language Reference* for more information on digraphs.

DLL | NODLL

Category

Object code control

Pragma equivalent

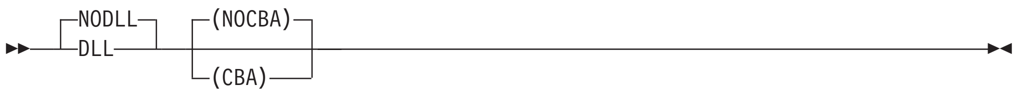
None.

Purpose

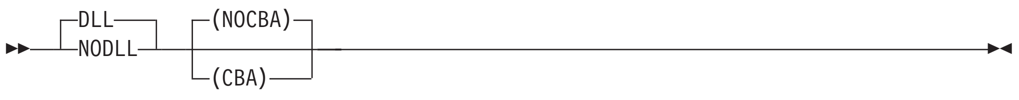
Generates object code for DLLs or DLL applications.

Syntax

For C and IPA Link:



For C++:



Defaults

For a C compile and the IPA Link step, the default option is NODLL(NOCBA). For a C++ compile, the default option is DLL(NOCBA).

Parameters

NOCALLBACKANY
This is the default. If you specify NOCALLBACKANY, no changes will be made to the function pointer in your compile unit. The abbreviation for NOCALLBACKANY is NOCBA.

CALLBACKANY
If you specify CALLBACKANY, all calls through function pointers will accommodate function pointers created by applications compiled without the DLL option. This accommodation accounts for the incompatibility of function pointers created with and without the DLL compiler option. The CALLBACKANY suboption is not supported when the XPLINK option is used. When function pointers having their origins (that is, where the address of a function is taken and assigned to a function pointer) in XPLINK code in the same or another DLL, or NOXPLINK NODLL code in another DLL, or non-XPLINK DLL code in another DLL, are passed to exported XPLINK functions, the compiler inserts code to check whether or not the function pointers received as actual arguments are valid (useable directly) XPLINK function pointers, and converts them if required. This provides results that are similar in many respects to the function pointer conversion provided when DLL(CALLBACKANY) is specified for non-XPLINK code. Other function pointers that have their origins in non-XPLINK code, including function pointer parameters passed to non-exported functions or otherwise acquired, are not converted automatically by XPLINK compiled code. Use of such function pointers will cause the application to fail. The abbreviation for CALLBACKANY is CBA.

Usage

When the DLL option is in effect, the compiler is instructed to produce DLL code. The DLL code can export or import functions and external variables.

Note: You should write your code according to the rules listed in the *z/OS XL C/C++ Programming Guide*, and compile with the NOCALLBACKANY suboption. Use the suboption CALLBACKANY only when you have calls through function pointers and C code compiled without the DLL option. CALLBACKANY causes *all* calls through function pointers to incur overhead due to internally-generated calls to library routines that determine whether the function pointed to is in a DLL (in which case internal control structures need to be updated), or not. This overhead is unnecessary in an environment where all function pointers were created either in C++ code or in C code compiled with the DLL option.

For information on how to create or use DLLs, and on when to use the appropriate DLL options and suboptions, see *z/OS XL C/C++ Programming Guide*.

Notes:

1. If NODLL is specified for C++, it will be ignored.
2. You must use the LONGNAME and RENT options with the DLL option. If you use the DLL option without RENT and LONGNAME, the z/OS XL C compiler automatically turns them on. However, when the XPLINK option is used, though RENT and LONGNAME are the default options, both NOLONGNAME and NORENT are allowed.

3. In code compiled with the XPLINK compiler option, function pointers are compared using the address of the descriptor. No special considerations, such as dereferencing, are required to initialize the function pointer prior to comparison.
4. In code compiled with the NOXPLINK compiler option, you cannot cast a non-zero integer `const` type to a DLL function pointer type as shown in the following example:

```
void (*foo)();

void main() {
             /* ... */

         if (foo != (void (*)()) (50L) ) {
             /* do something other than calling foo */
         }
}
```

   This conditional expression will cause an abend at execution time because the function pointer (with value `50L`) needs to be dereferenced to perform the comparison. The compiler will check for this type of casting problem if you use the CHECKOUT(CAST) option along with the DLL option. See "CHECKOUT | NOCHECKOUT (C only)" on page 79 for more information on obtaining diagnostic information for C applications.

IPA effects

The IPA Compile step generates information for the IPA Link step. The CALLBACKANY option also affects the regular object module if you request one by specifying the IPA(OBJECT) option.

The IPA Link step accepts the DLL compiler option, but ignores it.

The IPA Link step uses information from the IPA Compile step to classify an IPA object module as DLL or non-DLL as follows:

- C code that is compiled with the DLL option is classified as DLL.
- C++ code is classified as DLL
- C code that is compiled with the NODLL option is classified as non-DLL.

Note: If you are using IPA and specify the DLL compiler option, your code should export at least one function.

Each partition is initially empty and is set as DLL or non-DLL, when the first subprogram (function or method) is placed in the partition. The setting is based on the DLL or non-DLL classification of the IPA object module which contained the subprogram. Procedures from IPA object modules with incompatible DLL values will not be inlined. This results in reduced performance. For best performance, compile your application as all DLL code or all non-DLL code.

The IPA Link step allows you to input a mixture of IPA objects that are compiled with DLL(CBA) and DLL(NOCBA). The IPA Link step does not convert function pointers from the IPA Objects that are compiled with the option DLL(NOCBA).

You should only export subprograms (functions and C++ methods) or variables that you need for the interface to the final DLL. If you export subprograms or variables unnecessarily (for example, by using the EXPORTALL option), you severely limit IPA optimization. Global variables are not coalesced, and unreachable or 100% inlined code is not pruned.

Predefined macros

- For C, __DLL__ is predefined to 1 when the DLL option is in effect; otherwise it is undefined.
- For C++, __DLL__ is always predefined to 1.

ENUMSIZE

Category

Floating-point and integer control

Pragma equivalent

#pragma enum

Purpose

Specifies the amount of storage occupied by enumerations

Syntax

```
               ┌─SMALL───┐
►►──ENUM──(────┼─INT─────┼──)─────────────────────────────────────────────────►◄
               ├─INTLONG─┤
               ├─1───────┤
               ├─2───────┤
               ├─4───────┤
               └─8───────┘
```

Defaults

ENUM(SMALL)

Parameters

SMALL
: Specifies that enumerations occupy a minimum amount of storage, which is either 1, 2, 4, or 8 bytes of storage, depending on the range of the `enum` constants.

INT
: Specifies that enumerations occupy 4 bytes of storage and are represented by `int`.

INTLONG
: Valid only when LP64 is specified and for C++ only. It specifies that enumerations occupy 8 bytes of storage and are represented by `long` if the range of the `enum` constants exceed the limit for `int`. Otherwise, the enumerations occupy 4 bytes of storage and are represented by `int`.

1 Specifies that enumerations occupy 1 byte of storage.

2 Specifies that enumerations occupy 2 bytes of storage

4 Specifies that enumerations occupy 4 bytes of storage.

8 Specifies that enumerations occupy 8 bytes of storage. This suboption is only valid with LP64.

Usage

When the ENUMSIZE option is in effect, you can select the type used to represent all `enum` constants defined in a compilation unit.

The following tables illustrate the preferred sign and type for each range of `enum` constants:

Table 21. ENUM constants for C and C++

ENUM Constants	small	1	2	4	8 *	int	intlong * (C++ only)
0..127	unsigned char	signed char	short	int	long	int	int
-128..127	signed char	signed char	short	int	long	int	int
0..255	unsigned char	unsigned char	short	int	long	int	int
0..32767	unsigned short	ERROR	short	int	long	int	int
-32768..32767	short	ERROR	short	int	long	int	int
0..65535	unsigned short	ERROR	unsigned short	int	long	int	int
0..2147483647	unsigned int	ERROR	ERROR	int	long	int	int
$-2^{31}..2^{31}-1$	int	ERROR	ERROR	int	long	int	int
0..4294967295	unsigned int	ERROR	ERROR	unsigned int	long	unsigned int (C++ only) ERROR for C	unsigned int
$0..(2^{63}-1)$ *	unsigned long	ERROR	ERROR	ERROR	long	ERROR	long
$-2^{63}..(2^{63}-1)$ *	long	ERROR	ERROR	ERROR	long	ERROR	long
$0..2^{64}$ *	unsigned long	ERROR	ERROR	ERROR	unsigned long	ERROR	unsigned long

Note: The rows and columns marked with asterisks (*) in this table are only valid when the LP64 option is in effect.

Predefined macros

The __ENUM_OPT macro is defined only by the C compiler, which predefines it to 1 when the ENUMSIZE option is in effect; otherwise it is undefined.

Examples

If the specified storage size is smaller than that required by the range of `enum` constants, an error is issued by the compiler; for example:

```
#include <limits.h>
#pragma enum(1)
enum e_tag {
    a = 0,
     b = SHRT_MAX /* error */
 } e_var;
 #pragma enum(reset)
```

EPILOG (C only)

Category

Object code control

Pragma equivalent

#pragma epilog (C only)

Purpose

Enables you to provide your own function exit code for all functions that have `extern` scope, or for all extern and static functions.

Syntax

```
►►──EPILOG──(──┬─"text-string"──────────────────────┬──)──────────────────────►◄
               ├─EXTERN──┬──(──"text-string"──)────┘
               └─ALL─────┘
```

Defaults

The compiler generates default epilog code for the functions that do not have user-supplied epilog code.

Parameters

text-string

text-string is a C string, which must contain valid HLASM statements.

If the *text-string* consists of white-space characters only or if the *text-string* is not provided, then the compiler ignores the option specification. If the *text-string* does not contain any white-space characters, then the compiler will insert leading spaces in front. Otherwise, the compiler will insert the *text-string* into the function epilog location of the generated assembler source. The compiler does not understand or validate the contents of the *text-string*. In order to satisfy the assembly step later, the given *text-string* must form valid HLASM code with the surrounding code generated by the compiler.

Note: Special characters like `newline` and `quote` are shell (or command line) meta characters, and maybe preprocessed before reaching the compiler. It is advisable to avoid using them. The intended use of this option is to specify an assembler macro as the function epilog.

For more information on valid HLASM statements, see **#pragma epilog**.

EXTERN

If the EPILOG option is specified with this suboption or without any suboption, the epilog applies to all functions that have external linkage in the compilation unit.

ALL

If the EPILOG option is specified with this suboption, the epilog also applies to static functions defined in the compilation unit.

Usage

For more information on METAL C default epilog code, see *z/OS Metal C Programming Guide and Reference*.

Notes:

1. The EPILOG option is only valid when the METAL option is specified.
2. When the EPILOG option is specified multiple times with the same suboption **all** or **extern**, only the function entry code of the last suboption specified will be displayed.

3. The EPILOG option with the suboption **all** overwrites the one with **extern** suboption, or the one without any suboption.

Predefined macros

None.

Related information

For more information on the METAL compiler option, see “METAL | NOMETAL (C only)” on page 182.

See “PROLOG (C only)” on page 206 for information on providing function entry code for system development.

EVENTS | NOEVENTS

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Produces an event file that contains error information and source file statistics.

Syntax

```
      ┌─NOEVENT─┐
►►────┴─EVENT───┴──┬──────────────────────────────────────────────────┬────────────────►◄
                   └─(──┬─Sequential filename──────────────────┬─)─┘
                        ├─Partitioned data set─────────────────┤
                        ├─Partitioned data set (member)────────┤
                        ├─z/OS UNIX System Services filename───┤
                        └─z/OS UNIX System Services directory──┘
```

Defaults

NOEVENTS

Parameters

Sequential filename
: Specifies the sequential data set file name for the event file.

Partitioned data set
: Specifies the partitioned data set for the event file.

Partitioned data set (member)
: Specifies the partitioned data set (member) for the event file.

z/OS UNIX System Services filename
: Specifies the z/OS UNIX System Services file name for the event file.

z/OS UNIX System Services directory
: Specifies the z/OS UNIX System Services directory for the event file.

Usage

The compiler writes the events data to the DD:SYSEVENT ddname, if you allocated one before you called the compiler. If this ddname is not allocated, the compiler will allocate one dynamically using default characteristics (LRECL=4095,RECFM=V,BLKSIZE=4099), and the name is the source file name with SYSEVENT as the lowest-level qualifier. You can control the name by specifying the file name as the suboption of the EVENTS option.

If you specified a suboption, the compiler uses the *data set* that you specified, and ignores the DD:SYSEVENT.

There is no set requirement on the file characteristics for the event file. If you wish to allocate an event file, you should specify a record length that is large enough to contain the longest message that the compiler can emit plus approximately 40 bytes for the control information.

If the source file is a z/OS UNIX System Services file, and you do not specify the event file name as a suboption, the compiler writes the event file in the current working directory. The event file name is the name of the source file with the extension `.err`.

The compiler ignores `#line` directives when the EVENTS option is active, and issues a warning message.

For a description of the layout of the event file, see Appendix E, "Layout of the Events file," on page 669.

Predefined macros
None.

EXECOPS | NOEXECOPS

Category
Object code control

Pragma equivalent
#pragma runopts (EXECOPS)

Purpose
Allows you to specify run-time options on the invocation line for the generated executable.

Syntax

```
      ┌─EXEC───┐
►►────┴─NOEXEC─┴──────────────────────────────────────────────────►◄
```

Defaults
EXECOPS

Usage
When the EXECOPS option is in effect, you can control whether run-time options will be recognized at run time without changing your source code.

If the EXECOPS option is specified on both the command line and in a **#pragma runopts** directive, the option on the command line takes precedence.

IPA effects
If you specify EXECOPS for any compilation unit in the IPA Compile step, the compiler generates information for the IPA Link step. This option also affects the regular object module if you request one by specifying the IPA(OBJECT) option.

If you specify the EXECOPS option for the IPA Compile step, you do not need to specify it again on the IPA Link step. The IPA Link step uses the information generated for the compilation unit that contains the `main()` function. If it cannot find a compilation unit that contains `main()`, it uses information generated for the first compilation unit that it finds.

If you specify this option on both the IPA Compile and the IPA Link steps, the setting on the IPA Link step overrides the setting on the IPA Compile step. This situation occurs whether you use EXECOPS and NOEXECOPS as compiler options, or specify them by using the **#pragma runopts** directive on the IPA Compile step.

Predefined macros
None.

EXH | NOEXH (C++ only)

Category
Object code control

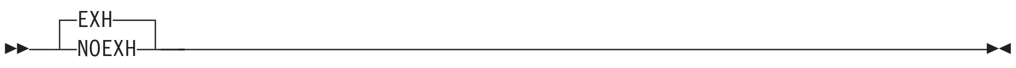
Pragma equivalent
None.

Purpose
Controls whether C++ exception handling is enabled in the module being compiled.

When the EXH option is in effect, you can control the generation of C++ exception handling code.

When the NOEXH option is in effect, the generation of the exception handling code is suppressed, which results in code that runs faster, but it will not be ANSI-compliant if the program uses exception handling.

Syntax



Defaults
EXH

Usage
If you compile a source file with NOEXH, active objects on the stack are not destroyed if the stack collapses in an abnormal fashion. For example, if a C++ object is thrown, or a Language Environment exception or signal is raised, objects on the stack will not have their destructors run.

If NOEXH has been specified and the source file has try/catch blocks or throws objects, the program may not execute as expected.

IPA effects
The IPA Link step issues a diagnostic message if you specify the EXH option for that step.

Predefined macros
_CPPUNWIND is predefined to 1 when the EXH option is in effect; otherwise it is undefined.

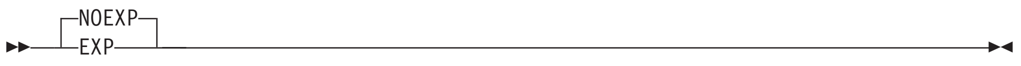
EXPMAC | NOEXPMAC

Category
Listings, messages, and compiler information

Pragma equivalent
None.

Purpose
Lists all expanded macros in the source listing.

Syntax



Defaults
NOEXPMAC

In the z/OS UNIX System Services environment, this option is turned on by specifying -V when using the **c99**, **c89**, **cc** or **c++** commands.

Usage
If you want to use the EXPMAC option, you must also specify the SOURCE compiler option to generate a source listing. If you specify the EXPMAC option but omit the SOURCE option, the compiler issues a warning message, and does not produce a source listing.

Predefined macros
None.

Related information
For more information on the SOURCE compiler option, see “SOURCE | NOSOURCE” on page 226.

EXPORTALL | NOEXPORTALL

Category
Object code control

Pragma equivalent
#pragma export

Purpose
Exports all externally defined functions and variables in the compilation unit so that a DLL application can use them.

Syntax

```
      ┌─NOEXPO─┐
►►────┴─EXPO───┴──────────────────────────────────────────────────►◄
```

Defaults

NOEXPORTALL

Usage

Use the EXPORTALL option if you are creating a DLL and want to export all external functions and variables defined in the DLL. You may not export the `main()` function.

Notes:

1. If you only want to export some of the external functions and variables in the DLL, use **#pragma export**, or the `_Export` keyword for C++.
2. For C, you must use the LONGNAME and RENT options with the EXPORTALL option. If you use the EXPORTALL option without RENT and LONGNAME, the z/OS XL C compiler turns them on.
3. Unused extern inline functions will not be exported when the EXPORTALL option is specified.

IPA effects

The IPA Compile step generates information for the IPA Link step. The EXPORTALL option also affects the regular object module if you request one by specifying the IPA(OBJECT) option.

The IPA Link step accepts the EXPORTALL option, but ignores it.

If you use the EXPORTALL option during the IPA Compile step, you severely limit IPA optimization. Refer to "DLL | NODLL" on page 102 for more information about the effects of this option on IPA processing.

Predefined macros

None.

Related information

For more information on related compiler options, see:

- "LONGNAME | NOLONGNAME" on page 167
- "RENT | NORENT (C only)" on page 209

FASTTEMPINC | NOFASTTEMPINC (C++ only)

Category

C++ template

Pragma equivalent

None.

Purpose

Defers generation of object code until the final version of all template definitions have been determined.

When the FASTTEMPINC option is in effect, the compiler uses a single compilation pass to generate the final object code, resulting in improved compilation time when recursive templates are used in an application. This means that time is not wasted on generating object code that will be discarded and generated again.

When the NOFASTTEMPINC option is in effect, the compiler generates object code each time a `tempinc` source file is compiled. If recursive template definitions in a subsequent `tempinc` source file cause additional template definitions to be added to a previously processed file, an additional recompilation pass is required.

Syntax

```
      ┌─NOFASTT─┐
►►────┴─FASTT───┴──────────────────────────────────────────────────────────►◄
```

Defaults

NOFASTT

Usage

The FASTTEMPINC option may improve template instantiation compilation time when large numbers of recursive templates are used in an application.

Use FASTT if you have large numbers of recursive templates. If your application has very few recursive template definitions, the time saved by not doing code generation may be less than the time spent in source analysis on the additional template compilation pass. In this case, it may be better to use NOFASTT.

IPA effects

The IPA Link step issues a diagnostic message if you specify the FASTT option for that step.

Predefined macros

None.

FLAG | NOFLAG

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Limits the diagnostic messages to those of a specified level or higher.

Syntax

```
      ┌─FL──(──severity──)─┐
►►────┴─NOFL───────────────┴───────────────────────────────────────────────►◄
```

Defaults

FLAG(I)

For the z/OS UNIX System Services utilities, the default for a regular compile is FLAG(W).

To specify FLAG(I) using the c89 utility, you can specify -V.

Parameters

severity
Specifies the minimum severity level.

severity may be one of the following:

I	An informational message.
W	A warning message that calls attention to a possible error, although the statement to which it refers is syntactically valid.
E	An error message that shows that the compiler has detected an error and cannot produce an object deck.
S	A severe error message that describes an error that forces the compilation to terminate.
U	An unrecoverable error message that describes an error that forces the compilation to terminate.

Usage

When the FLAG option is in effect, you can specify the minimum severity level of diagnostic messages to be reported in a listing and displayed on a terminal.

If you specified the options SOURCE or LIST, the messages generated by the compiler appear immediately following the incorrect source line, and in the message summary at the end of the compiler listing.

The NOFLAG option is the same as the FLAG(U) option.

IPA effects

The FLAG option has the same effect on the IPA Link step that it does on a regular compilation.

Predefined macros

None.

Related information

For more information on related compiler options, see

- "SOURCE | NOSOURCE" on page 226
- "LIST | NOLIST" on page 163

FLOAT

Category

Floating-point and integer control

Pragma equivalent

None.

Purpose

Selects different strategies for speeding up or improving the accuracy of floating-point calculations.

| ### Syntax

|

```
             ┌─,─────────────────────────────────┐
             ▼                                   │
►►──FLOAT──(───┬─HEX │ IEEE──────────────────┬───┴──)──────────────►◄
               ├─FOLD │ NOFOLD───────────────┤
               ├─MAF │ NOMAF─────────────────┤
               ├─RRM │ NORRM─────────────────┤
               └─AFP │ NOAFP──┬──────────────┬┘
                              │  ┌─NOVOLATILE─┐  │
                              └─(──┴─VOLATILE───┴─)─┘
```

Defaults

- FLOAT(HEX, FOLD, NOMAF, NORRM, AFP(NOVOLATILE))
- For LP64, the default suboption is IEEE.
- For ILP32, the default is HEX.
- For METAL, the default is IEEE.
- If NOSTRICT and FLOAT(IEEE) are specified, MAF is the default.
- For ARCH(2), the default suboption is NOAFP.
- For ARCH(3) or higher, the default suboption is AFP(NOVOLATILE).
- When there is no sub-option specified for AFP, the default is NOVOLATILE.

Parameters

HEX | IEEE

Specifies the format of floating-point numbers and instructions:

- IEEE instructs the compiler to generate binary floating-point numbers and instructions. The unabbreviated form of this suboption is IEEE754.
- HEX instructs the compiler to generate hexadecimal floating-point formatted numbers and instructions. The unabbreviated form of this suboption is HEXADECIMAL.

FOLD | NOFOLD

Specifies that constant floating-point expressions in function scope are to be evaluated at compile time rather than at run time. This is known as *folding*.

In binary floating-point mode, the folding logic uses the rounding mode set by the ROUND option.

In hexadecimal floating-point mode, the rounding is always towards zero. If you specify NOFOLD in hexadecimal mode, the compiler issues a warning and uses FOLD.

MAF | NOMAF

Makes floating-point calculations faster and more accurate by using floating-point multiply-add instructions where appropriate. The results may not be exactly equivalent to those from similar calculations performed at compile time or on other types of computers. Negative zero results may be produced. This option may affect the precision of floating-point intermediate results.

Note: The suboption MAF does not have any effect on extended floating-point operations.

MAF is not available for hexadecimal floating-point mode.

RRM | NORRM

Run-time Rounding Mode (RRM) prevents floating-point optimizations that are incompatible with runtime rounding to plus and minus infinity modes. It informs the compiler that the floating-point rounding mode may change at run time or that the floating-point rounding mode is not round to nearest at run time.

RRM is not available for hexadecimal floating-point mode.

AFP(VOLATILE | NOVOLATILE) | NOAFP

AFP instructs the compiler to generate code which makes use of the full complement of 16 floating point registers. These include the four original floating-point registers, numbered 0, 2, 4, and 6, and the Additional Floating Point (AFP) registers, numbered 1, 3, 5, 7 and 8 through 15.

AFP is not available before ARCH(3). If the code generated utilizing AFP registers must run on a pre-ARCH(3) machine, emulation is provided by the operating system. Code with AFP registers will not run on a system that is older than G5 and OS/390 V2R6.

Note: This emulation has a significant performance cost to the execution of the application on the non-AFP processors. This is why the default is NOAFP when ARCH(2) or lower is specified.

If VOLATILE is specified then AFP FPRs 8-15 are considered volatile, which means that FPRs 8-15 are not expected to be preserved by the called program.

Note: The AFPs are FPR1, 3, 5, 7 and 8-15. However, FPRs 0-7 are always considered volatile. The AFP(VOLATILE | NOVOLATILE) option only controls how the compiler handles AFP FPRs 8-15, and not all the AFP registers.

Compiling with AFP(VOLATILE) is no longer necessary with the latest CICS, which has Extended MVS Linkage Convention Support and has this implemented natively. However, you may need to recompile your codes with the default AFP(NOVOLATILE) when using the latest CICS. Use the AFP(VOLATILE) sub-option with discretion and evaluate the performance costs prior to using this suboption. See Table 22 for a summary of the various situations:

Table 22. Various situations between AFP(VOLATILE) and CICS

CICS Version	AFP(VOLATILE)	Result	Comment	Action
2.3, 3.1 and 3.2	Not recompiled	Unsafe	Recompile with AFP(VOLATILE) but data corruption could still occur.	Stay on your current CICS or go to CICS 4.1 with Extended MVS Linkage Convention Support.
2.3, 3.1 and 3.2	Recompiled	Performance Impact	Recompile and data is safe. However at a potential performance lost.	Stay on your current CICS or go to CICS 4.1, where recompiling without AFP(VOLATILE) can regain performance lost.
4.1	Not recompiled	OK	Not recompile and move to CICS 4.1.	None. Data is safe when using CICS 4.1 with Extended MVS Linkage Convention Support.
4.1	Recompiled	Performance Impact	Recompile and moved to CICS 4.1.	Continue safely without recompiling. However recompiling without AFP(VOLATILE) can regain performance lost.

| NOAFP limits the compiler to generating code using only the original four floating-point registers, 0, 2, 4, and 6, which are available on all IBM System z machine models.

Usage

When the FLOAT option is in effect, you can select the format of floating-point numbers. The format can be either base 2 IEEE-754 binary format, or base 16 z/Architecture hexadecimal format.

You should use IEEE floating-point in the following situations:

- You deal with data that are already in IEEE floating-point format
- You need the increased exponent range (see *z/OS XL C/C++ Language Reference* for information on exponent ranges with IEEE-754 floating-point)
- You want the changes in programming paradigm provided by infinities and NaN (not a number)

For more information about the IEEE format, refer to the *IEEE 754-1985 IEEE Standard for Binary Floating-Point Arithmetic*.

When you use IEEE floating-point, make sure that you are in the same rounding mode at compile time (specified by the ROUND(*mode*) option), as at run time. Entire compilation units will be compiled with the same rounding mode throughout the compilation. If you switch runtime rounding modes inside a function, your results may vary depending upon the optimization level used and other characteristics of your code; switch rounding mode inside functions with caution.

If you have existing data in hexadecimal floating-point (the original base 16 z/Architecture hexadecimal floating-point format), and have no need to communicate this data to platforms that do not support this format, there is no reason for you to change to IEEE floating-point format.

Applications that mix the two formats are not supported.

The binary floating-point instruction set is physically available only on processors that are part of the ARCH(3) group or higher. You can request FLOAT(IEEE) code generation for an application that will run on an ARCH(2) or earlier processor, if that processor runs on the OS/390 Version 2 Release 6 or higher operating system. This operating system level is able to intercept the use of an ″illegal″ binary floating-point instruction, and emulate the execution of that instruction such that the application logic is unaware of the emulation. This emulation comes at a significant cost to application performance, and should only be used under special circumstances. For example, to run exactly the same executable object module on backup processors within your organization, or because you make incidental use of binary floating-point numbers.

IPA effects

The IPA Compile step generates information for the IPA Link step. This option also affects the regular object module if you request one by specifying the IPA(OBJECT) option.

Note: The option FLOAT(AFP(VOLATILE)) is not supported by IPA. If the option FLOAT(AFP(VOLATILE)) is passed to the IPA Compile or Link phase, then the IPA phase will emit a severe diagnostic message.

The IPA Link step merges and optimizes the application code, and then divides it into sections for code generation. Each of these sections is a partition. The IPA Link step uses information from the IPA Compile step to determine if a subprogram can be placed in a particular partition. Only compatible subprograms are included in a given partition. Compatible subprograms have the same floating-point mode, and the same values for the FLOAT suboptions, and the ROUND and STRICT options:

- Floating-point mode (binary or hexadecimal)

  The floating-point mode for a partition is set to the floating-point mode (binary or hexadecimal) of the first subprogram that is placed in the partition. Subprograms that follow are placed in partitions that have the same floating-point mode; a binary floating-point mode subprogram is placed in a binary floating-point mode partition, and a hexadecimal mode subprogram is placed in a hexadecimal mode partition.

  If you specify FLOAT(HEX) or FLOAT(IEEE) during the IPA Link step, the option is accepted, but ignored. This is because it is not possible to change the floating-point mode after source analysis has been performed.

  The Prolog and Partition Map sections of the IPA Link step listing display the setting of the floating-point mode.

- AFP | NOAFP

  The value of AFP for a partition is set to the AFP value of the first subprogram that is placed in the partition. Subprograms that have the same AFP value are then placed in that partition.

  You can override the setting of AFP by specifying the suboption on the IPA Link step. If you do so, all partitions will contain that value, and the Prolog section of the IPA Link step listing will display the value.

  The Partition Map section of the IPA Link step listing and the END information in the IPA object file display the current value of the AFP suboption.

- FOLD | NOFOLD

  Hexadecimal floating-point mode partitions are always set to FOLD.

  For binary floating-point partitions, the value of FOLD for a partition is set to the FOLD value of the first subprogram that is placed in the partition. Subprograms that have the same FOLD value are then placed in that partition. During IPA inlining, subprograms with different FOLD settings may be combined in the same partition. When this occurs, the resulting partition is always set to NOFOLD.

  You can override the setting of FOLD | NOFOLD by specifying the suboption on the IPA Link step. If you do so, all binary floating-point mode partitions will contain that value, and the Prolog section of the IPA Link step listing will display the value.

  For binary floating-point mode partitions, the Partition Map section of the IPA Link step listing displays the current value of the FOLD suboption.

- MAF | NOMAF

  For IPA object files generated with the FLOAT(IEEE) option, the value of MAF for a partition is set to the MAF value of the first subprogram that is placed in the partition. Subprograms that have the same MAF for this suboption are then placed in that partition.

  For IPA object files generated with the FLOAT(IEEE) option, you can override the setting of MAF | NOMAF by specifying the suboption on the IPA Link step. If you do so, all binary floating-point mode partitions will contain that value, and the Prolog section of the IPA Link step listing will display the value.

  For binary floating-point mode partitions, the Partition Map section of the IPA Link step listing displays the current value of the MAF suboption.

  Hexadecimal mode partitions are always set to NOMAF. You cannot override this setting.

- RRM | NORRM

  For IPA object files generated with the FLOAT(IEEE) option, the value of RRM for a partition is set to the RRM value of the first subprogram that is placed in the partition. During IPA inlining, subprograms with different RRM settings may be combined in the same partition. When this occurs, the resulting partition is always set to RRM.

  For IPA object files generated with the FLOAT(IEEE) option, you can override the setting of RRM | NORRM by specifying the suboption on the IPA Link step. If you do so, all binary floating-point mode partitions will contain that value, and the Prolog section of the IPA Link step listing will display the value.

  For binary floating-point mode partitions, the Partition Map section of the IPA Link step listing displays the current value of the RRM suboption.

  Hexadecimal mode partitions are always set to NORRM. You cannot override this setting.
- ROUND option

  For IPA object files generated with the FLOAT(IEEE) option, the value of the ROUND option for a partition is set to the value of the first subprogram that is placed in the partition.

  You can override the setting of ROUND by specifying the option on the IPA Link step. If you do so, all binary floating-point mode partitions will contain that value, and the Prolog section of the IPA Link step listing will display the value.

  For binary floating-point mode partitions, the Partition Map section of the IPA Link step listing displays the current value of the ROUND suboption.

  Hexadecimal mode partitions are always set to round towards zero. You cannot override this setting.
- STRICT option

  The value of the STRICT option for a partition is set to the value of the first subprogram that is placed in the partition. During IPA inlining, subprograms with different STRICT settings may be combined in the same partition. When this occurs, the resulting partition is always set to STRICT.

  You can override the setting of STRICT by specifying the option on the IPA Link step. If you do so, the Prolog section of the IPA Link step listing will display the value.

  If there are no Compilation Units with subprogram-specific STRICT options, all partitions will have the same STRICT value.

  If there are any Compilation Units with subprogram-specific STRICT options, separate partitions will continue to be generated for those subprograms with a STRICT option, which differs from the IPA Link option.

  The Partition Map sections of the IPA Link step listing and the object module display the value of the STRICT option.

Note: The inlining of subprograms (C functions, C++ functions and methods) is inhibited if the FLOAT suboptions (including the floating-point mode), and the ROUND and STRICT options are not all compatible between compilation units. Calls between incompatible compilation units result in reduced performance. For best performance, compile your applications with consistent options.

Predefined macros

__BFP__ is defined to 1 when you specify binary floating point (BFP) mode by using the FLOAT(IEEE) compiler option.

GENASM | NOGENASM (C only)

Category

Compiler output

Pragma equivalent

None.

Purpose

Instructs the compiler to generate High-Level Assembler (HLASM) source code instead of object code for the program being compiled.

When the GENASM option is in effect, the compiler generates HLASM source code in the output file. The GENASM option also enables the inlined assembly support using the GCC-style of `__asm` statements.

When the NOGENASM option is in effect, the compiler generates object code in the output file.

Syntax

```
   ┌─NOGENASM─┐
►►─┴─GENASM───┴──┬───────────────────────────────────────────────────┬──────────►◄
                 └─(──┬─Sequential filename──────────────────┬──)─┘
                      ├─Partitioned data set─────────────────┤
                      ├─Partitioned data set (member)────────┤
                      ├─z/OS UNIX System Services filename───┤
                      └─z/OS UNIX System Services directory──┘
```

Defaults

NOGENASM

Parameters

Sequential filename
Specifies the sequential data set file name for the output file.

Partitioned data set
Specifies the partitioned data set for the output file.

Partitioned data set (member)
Specifies the partitioned data set (member) for the output file.

z/OS UNIX System Services filename
Specifies the z/OS UNIX System Services file name for the output file.

z/OS UNIX System Services directory
Specifies the z/OS UNIX System Services directory for the output file.

Usage

Note: GENASM is automatically in effect when the METAL option is specified.

Because the GENASM option causes the compiler to generate code in HLASM source code format, no pseudo-assembly listing will be produced for the compilation unit when the LIST option is specified.

GENASM implies NOOBJECT.

When a file name suboption is specified with NOGENASM, the file name then becomes the default.

In batch mode only:

- If you subsequently use the GENASM option without a file name suboption, the compiler uses the file name that you previously specified with the NOGENASM compiler option. For example, the following specifications have the same result:
  ```
  NOGENASM(hello.asm) METAL GENASM
  GENASM(hello.asm) METAL
  ```
- If you do not specify a file name for the GENASM option, the compiler determines the output file name as follows:
  - If the JCL allocates DDNAME SYSLIN, it is used as the output file.
  - If the C source code is from a host data set, the compiler determines the output data set name by replacing the high-level qualifier of the input data set name with the userid of the owner of the job and appending .ASM as the new low-level qualifier.
  - If the C source code is from a z/OS UNIX System Services file, the output file name is the input file name with the suffix changed to `.s`.

In the z/OS UNIX System Services environment only:

- The GENASM option is supported only by the xlc utility in the form of the -S flag. The -S flag for the xlc utility specifies that the output file produced by the compiler is in assembler source code format. The absence of the -S flag indicates that the output file produced is in object code format. When -S is specified, the **-o** flag can be used to specify the output file name for the generated HLASM source code. If no **-o** flag is provided, the compiler determines the output file name as follows:
  - If the C source code is from a host data set, the compiler determines the output data set name by replacing the high-level qualifier with the userid of the owner of the process. The low-level qualifier is determined by whether the low-level qualifier of the input data set is a recognized suffix (`csuffix_host` or `cxxsuffix_host`). If it is a recognized suffix, the low-level qualifier is replaced with the suffix for assembler data sets (`ssuffix_host`). Otherwise, the suffix for the output data set is appended as the new low-level qualifier.
  - If the C source is from a z/OS UNIX file, the output file name is the input file name with the suffix replaced by the suffix for assembler files (`ssuffix`).

  The following specifications have the same result:
  ```
  xlc -S -qmetal hello.c
  xlc -S -qmetal -o hello.s hello.c
  xlc -S -qmetal -c hello.c
  xlc -S -qmetal -c -o hello.s hello.c
  ```

Predefined macros

None.

Related information

For information on the METAL option, see “METAL | NOMETAL (C only)” on page 182.

GOFF | NOGOFF

Category

Object code control

Pragma equivalent

None.

Purpose

Instructs the compiler to produce an object file in the Generalized Object File Format (GOFF).

When the GOFF and OBJECT options are in effect, the compiler produces an object file in GOFF format.

When the NOGOFF and OBJECT options are in effect, the compiler produces an object file in XOBJ format.

Syntax

```
     ┌─NOGOFF─┐
►►───┴─GOFF───┴──────────────────────────────────────────────────►◄
```

Defaults

- NOGOFF
- When XPLINK or LP64 is used, the default is GOFF.

Usage

The GOFF format supersedes the IBM S/370™ Object Module and Extended Object Module formats. It removes various limitations of the previous format (for example, 16 MB section size) and provides a number of useful extensions, including native z/OS support for long names and attributes. GOFF incorporates some aspects of industry standards such as XCOFF and ELF.

When you specify the GOFF option, the compiler uses LONGNAME and CSECT() by default. You can override these default values by explicitly specifying the NOLONGNAME or the NOCSECT option.

When you specify the GOFF option, you must use the binder to bind the output object. You cannot use the prelinker to process GOFF objects.

Note: When using GOFF and source files with duplicate file names, the linker may emit an error and discard one of the code sections. In this case, turn off the CSECT option by specifying NOCSECT.

IPA effects

The GOFF option affects the regular object module if you request one by specifying the IPA(OBJECT) option. This option affects the IPA-optimized object module generated when you specify the IPA(OBJECT) option.

The IPA information in an IPA object file is always generated using the XOBJ format.

The IPA Link Step merges and optimizes the application code, and then divides it into sections for code generation. Each of these sections is a partition. The GOFF option affects the object format of the code and data generated for each partition.

Information from non-IPA input files is processed and transformed based on the original format. GOFF format information remains in GOFF format; all other formats (OBJ, XOBJ, load module) are passed in XOBJ format.

Predefined macros

__GOFF__ is predefined to 1 when the GOFF compiler option is in effect.

Related information

For more information on related compiler options, see:

- "LONGNAME | NOLONGNAME" on page 167
- "CSECT | NOCSECT" on page 88

GONUMBER | NOGONUMBER

Category

Error checking and debugging

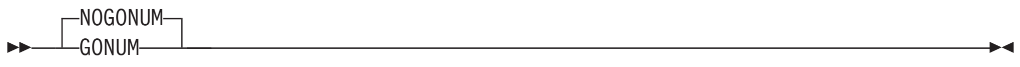
Pragma equivalent

#pragma options (gonumber) (C only), **#pragma options (nogonumber)** (C only)

Purpose

Generates line number tables that correspond to the input source file for Debug Tool and CEEDUMP processing.

Syntax



Defaults

- NOGONUMBER
- If DEBUG or TEST is specified, GONUMBER is the default.

Usage

The GONUMBER option is active by default when you use the DEBUG option. The DEBUG option will activate the GONUMBER option unless NOGONUMBER has been explicitly specified.

In the z/OS UNIX System Services environment, the GONUMBER option is enforced when you use the **-g** flag option using the **c89** or **xlc** commands. In another words, the **-g** flag option will always activate the GONUMBER option, regardless of other option specifications.

When you specify the GONUMBER option, a comment that indicates its use is generated in your object module to aid you in diagnosing your program.

Note: When the METAL option is specified, GONUMBER is not supported.

IPA effects

If you specify the GONUMBER option on the IPA Compile step, the compiler saves information about the source file line numbers in the IPA object file.

If you do not specify the GONUMBER option on the IPA Compile step, the object file produced contains the line number information for source files that contain function begin, function end, function call, and function return statements. This is the minimum line number information that the IPA Compile step produces. You can then use the TEST option on the IPA Link step to generate corresponding test hooks.

If you specify the GONUMBER option for the IPA Link step, the IPA Link step creates GONUMBER tables during code generation. The level of detail in these tables depends on the options that you used for the IPA Compile step. The GONUMBER table will only contain information for those compilation units that are compiled with the GONUMBER option.

If you do not specify the GONUMBER option on the IPA Link step, then the GONUMBER table will not be generated regardless of how the IPA object files are compiled.

Refer to “Interactions between compiler options and IPA suboptions” on page 43 and “LIST | NOLIST” on page 163 for more information.

Predefined macros

None.

Related information

For more information on related compiler options, see

- “TEST | NOTEST” on page 249
- “DEBUG | NODEBUG” on page 94

HALT(num)

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Stops compilation before producing any object, executable, or assembler source files if the maximum severity of compile-time messages equals or exceeds the severity specified for this option.

Syntax

```
►►──HALT──(num)──────────────────────────────────────────────────────►◄
```

Defaults

HALT(16)

Parameters

num
: Return code from the compiler. See *z/OS XL C/C++ Messages* for a list of return codes.

Usage

If the return code from compiling a particular member is greater than or equal to the value *num* specified in the HALT option, no more members are compiled. This option applies to the compilation of all members of a PDS or a z/OS UNIX System Services file system directory.

IPA effects
The HALT option affects the IPA Link step in a way similar to the way it affects the IPA Compile step, but the message severity levels may be different. Also, the severity levels for the IPA Link step and a C++ compilation include the ″unrecoverable″ level.

Predefined macros
None.

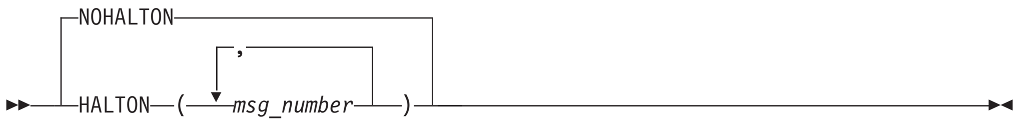
HALTONMSG | NOHALTONMSG

Category
Error checking and debugging

Pragma equivalent
None.

Purpose
Stops compilation before producing any object, executable, or assembler source files if a specified error message is generated.

Syntax



Defaults
NOHALTON

Parameters
msg_number
Message number.

Note: The HALTONMSG option allows you to specify more than one message number by separating the message numbers with commas.

Usage
When the HALTONMSG compiler option is in effect, the compiler stops after the compilation phase when it encounters the specified message number.

When the compilation stops as a result of the HALTONMSG option, the compiler return code is nonzero.

Predefined macros
None.

HGPR | NOHGPR

Category
Optimization and tuning

Pragma equivalent
None.

Purpose

Enables the compiler to exploit 64-bit General Purpose Registers (GPRs) in 32-bit programs targeting z/Architecture hardware.

When the HGPR compiler option is in effect, the compiler can exploit 64-bit GPRs in the generated code. The compiler will take the advantage of this permission when the code generation condition is appropriate.

When the NOHGPR and ILP32 compiler options are in effect, the compiler cannot exploit 64-bit GPRs in the generated code.

Syntax

```
     ┌─NOHGPR─┐
►►───┴─HGPR───┴──┬──────────────────────────────┬──────────────────────────────────►◄
                 │      ┌─NOPRESERVE─┐          │
                 └─(────┴─PRESERVE───┴────)─────┘
```

Defaults

NOHGPR

Parameters

PRESERVE
: Instructs the compiler to preserve the high halves of the 64-bit GPRs that a function is using, by saving them in the prolog for the function and restoring them in the epilog. The PRESERVE suboption is only necessary if the caller is not known to be z/OS XL C/C++ compiler-generated code.

NOPRESERVE
: Because of performance considerations, the default suboption for HGPR is NOPRESERVE.

Usage

HGPR means "High-half of 64-bit GPR", which refers to the use of native 64-bit instructions. In particular, if the application has the use of `long long` types, it should benefit from the native 64-bit instructions.

The HGPR compiler option requires ARCH(5) (and above).

IPA effects

The IPA Compile step generates information for the IPA Link step. If IPA(OBJECT) or IPA(OBJONLY) is specified, then the resulting object module will be compiled with the specified HGPR setting.

The IPA Link step will accept the HGPR option, but ignores it. The IPA Link step merges and optimizes the application code, and then divides it into sections for code generation. Each of these sections is a partition. The HGPR setting for a partition is determined by the first function that is imported into the partition. All other functions that are imported into the given partition must have the same HGPR option setting.

Predefined macros

None.

HOT | NOHOT

Category
Optimization and tuning

Pragma equivalent
None.

Purpose
Performs high-order loop analysis and transformations (HOT) during optimization.

Syntax

```
      ┌─NOHOT─┐
►►────┴─HOT───┴──────────────────────────────────────────────────────────────►◄
```

Defaults
NOHOT

In the z/OS UNIX System Services environment, if the xlc utility flag option -O, -O2, or -O3 is specified, the default is NOHOT. If -O4 or -O5 is specified, the default is HOT.

Usage

Note: You must specify either the LONGNAME compiler option or the **#pragma longname** preprocessor directive whenever the HOT option is specified. Otherwise, the compiler issues an unrecoverable error diagnostic message.

The HOT option is independent of the UNROLL command line option or pragma. The HOT option setting will be listed in the compiler listing, the IPA Link phase listing, and the end card of the output object file.

The HOT option can be specified with the debugging options TEST and DEBUG. The debugging information added to the output object code when HOT is used with TEST or DEBUG will be the same level of debugging information that is added when TEST or DEBUG is used with OPT(2).

If HOT is specified, the optimization level is forced to a level of at least 2. If a lower level is requested, it is increased to 2. If a higher level is requested, the requested value is used.

If **-g** is specified, then **-Wc,NOHOT** is forced.

Predefined macros
None.

Related information
For further information on the LONGNAME compiler option, see "LONGNAME | NOLONGNAME" on page 167.

IGNERRNO | NOIGNERRNO

Category
Optimization and tuning

Pragma equivalent

#pragma options (ignerrno) (C only), **#pragma options (noignerrno)** (C only)

Purpose

Allows the compiler to perform optimizations that assume `errno` is not modified by system calls.

When the IGNERRNO compiler option is in effect, the compiler is informed that your application is not using `errno`. Specifying this option allows the compiler to explore additional optimization opportunities for library functions in LIBANSI. The input to the library functions is assumed to be valid. Invalid input can lead to undefined behavior.

When the NOIGNERRNO compiler option is in effect, the compiler assumes that your application is using `errno`.

Syntax

For `NOOPT` and `OPT(2)`:

```
     ┌─NOIGNER─┐
►►───┴─IGNER───┴──────────────────────────────────────────────────►◄
```

For `OPT(3)`:

```
     ┌─IGNER───┐
►►───┴─NOIGNER─┴──────────────────────────────────────────────────►◄
```

Defaults

For NOOPT and OPT(2), the default option is NOIGNERRNO. For OPT(3), the default option is IGNERRNO.

Usage

ANSI library functions use `errno` to return the error condition. If your program does not use `errno`, the compiler has more freedom to explore optimization opportunities for some of these functions (for example, `sqrt()`). You can control this optimization by using the IGNERRNO option.

The IGNERRNO option is turned on by OPTIMIZE(3). Use NOIGNERRNO to turn it off if necessary. NOIGNERRNO must appear after OPTIMIZE(3), otherwise it is ignored.

IPA effects

The IPA Compile step generates information for the IPA Link step. The IGNERRNO option also affects the regular object module if you request one by specifying the IPA(OBJECT) option.

The IPA Link step accepts the IGNERRNO option, but ignores it. The IPA Link step merges and optimizes the application's code, and then divides it into sections for code generation. Each of these sections is a partition. The IPA Link step uses information from the IPA Compile step to determine if a subprogram can be placed in a particular partition. Only compatible subprograms are included in a given partition. Compatible subprograms have the same IGNERRNO option setting. For the purpose of this compatibility checking, objects produced by compilers prior to OS/390 Version 2 Release 9, where IGNERRNO is not supported, are considered NOIGNERRNO.

The value of the IGNERRNO option for a partition is set to the value of the first subprogram that is placed in the partition. The Partition Map sections of the IPA Link step listing and the object module display the value of the IGNERRNO option.

Predefined macros

None.

INFO | NOINFO

Category

Error checking and debugging

Pragma equivalent

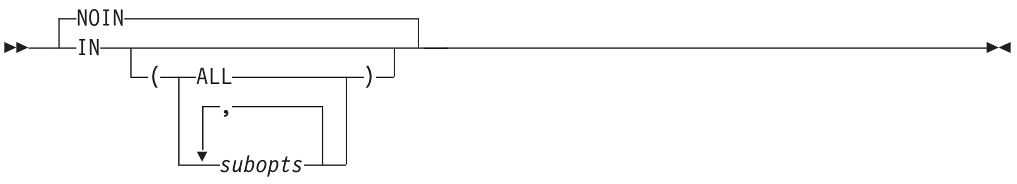
#pragma info (C++ only)

Purpose

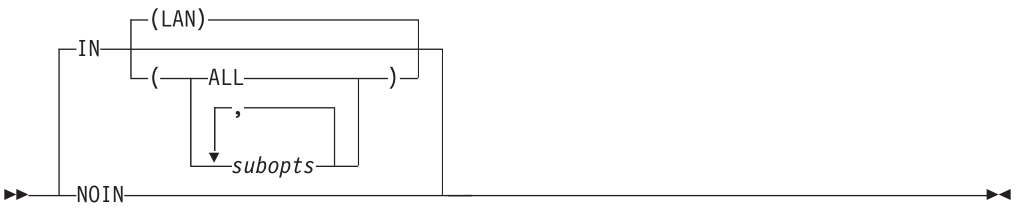
Produces groups of informational messages.

Syntax

For C:



For C++:



Defaults

For C++ in the z/OS UNIX System Services, the default is INFO(LAN) and NOINFO in batch. For C, the default option is NOINFO.

Parameters

subopts
: Use *subopts* if you want to specify the type of warning messages.

The following is a list of the applicable *subopts*:

CLS Emits class informational warning messages (C++ only).

CMP Emits conditional expression check messages.

CND	Emits messages on redundancies or problems in conditional expressions.
CNV	Emits messages about conversions.
CNS	Emits redundant operation on constants messages.
CPY	Emits warnings about copy constructors (C++ only).
EFF	Emits information about statements with no effect.
ENU	Emits information about ENUM checks.
EXT	Emits warnings about unused variables that have external declarations.
GNR	Emits information about the generation of temporary variables (C++ only).
GEN	Emits message if compiler generates temporaries.
LAN	Emits language level checks.
PAR	Emits warning messages on unused parameters.
POR	Emits warnings about non-portable constructs.
PPC	Emits messages on possible problems with using the preprocessor.
PPT	Emits trace of preprocessor actions.
REA	Emits warnings about unreached statements.
RET	Emits warnings about return statement consistency.
TRD	Emits warnings about possible truncation of data.
UND	Emits warnings about undefined classes (C++ only).
USE	Emits information about usage of variables.
VFT	Indicates where `vftable` is generated (C++ only).
ALL	Emits all of the above except a trace of preprocessor actions. Suboption PPT has to be turn on explicitly.

no suboptions
Same result as INFO(ALL).

Usage

Note: The INFO option may not produce the same diagnostic messages as the previous releases.

If you specify INFO with no suboptions, it is the same as specifying INFO(ALL).

The following information describes how to use INFO as a replacement for CHECKOUT and still retrieve the same messages:

Table 23. Migrating from CHECKOUT to INFO

CHECKOUT suboption	Equivalent INFO suboption
ACCURACY	TRD

Table 23. Migrating from CHECKOUT to INFO (continued)

CHECKOUT suboption	**Equivalent INFO suboption**
CAST	GEN **Note:** Use INFO(GEN) to provide the same messages as CHECKOUT(CAST) and the following: • some general messages • messages for appearance and usage of goto statements • messages for variables that are not explicitly initialized • messages for obsolete features • messages for ambiguous evaluation order
ENUM	ENU
EXTERN	EXT
GENERAL	Use one of the following: • INFO(CMP,CND,CNS,CNV,EFF,LAN, PRO,REA,RET,USE,GEN) **Note:** Use these options to provide the same messages as CHECKOUT(GENERAL) and messages for CHECKOUT(CAST,GOTO,INIT). • INFO(CMP,CND,CNS,CNV,EFF,LAN, PRO,REA,RET,USE) **Note:** Use these options to provide the same messages as CHECKOUT(GENERAL), but not the messages for obsolete features and ambiguous evaluation order.
GOTO	GEN **Note:** Use INFO(GEN) to provide the same messages as CHECKOUT(GOTO) and the following: • some general messages • messages for obsolete features • messages for ambiguous evaluation order • messages for CHECKOUT(CAST, INIT)
INIT	GEN **Note:** Use INFO(GEN) to provide the same messages as CHECKOUT(INIT) and the following: • some general messages • messages for obsolete features • messages for ambiguous evaluation order • messages for CHECKOUT(CAST, GOTO)
PARM	PAR
PORT	POR
PPCHECK	PPC

Table 23. Migrating from CHECKOUT to INFO (continued)

CHECKOUT suboption	Equivalent INFO suboption
PPTRACE	PPT
TRUNC	USE **Note:** INFO(USE) provides more messages than CHECKOUT(TRUNC).
ALL	ALL
NONE	NOINFO

IPA effects

The IPA Link step issues a diagnostic message if you specify the INFO option.

Predefined macros

None.

INITAUTO | NOINITAUTO

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Initializes automatic variables to a specific value for debugging purposes.

When the INITAUTO compiler option is in effect, the compiler instructs the compiler to generate extra code to initialize these variables with a user-defined value. This reduces the run-time performance of the program and should only be used for debugging.

When the NOINITAUTO compiler option is in effect, automatic variables without initializers are not implicitly initialized.

Syntax

```
      ┌─NOINITA──────────────────────────┐
►►────┴─INITA──(──nnnnnnnn──┬─────────┬──)─┴──────────────────────────────────►◄
                            └─, WORD──┘
```

Defaults

NOINITAUTO

Parameters

nnnnnnnn
: The hexadecimal value you specify for *nnnnnnnn* represents the initial value for automatic storage in bytes. It can be two to eight hexadecimal digits in length. There is no default for this value.

WORD
: The suboption WORD is optional, and can be abbreviated to W. If you specify WORD , *nnnnnnnnn* is a word initializer; otherwise it is a byte initializer. Only

one initializer can be in effect for the compilation. If you specify INITAUTO more than once, the compiler uses the last setting.

Note: The word initializer is useful in checking uninitialized pointers.

Usage

Automatic variables require storage only while the block in which they are declared is active. See *z/OS XL C/C++ Language Reference* for more information on automatic variables.

If you specify a byte initializer, and specify more than 2 digits for *nnnnnnnn*, the compiler uses the last 2 digits.

If you specify a word initializer, the compiler uses the last 2 digits to initialize a byte, and all digits to initialize a word.

IPA effects

The IPA Compile step generates information for the IPA Link step. The INITAUTO option also affects the regular object module if you request one by specifying the IPA(OBJECT) option.

If you do not specify the INITAUTO option in the IPA Link step, the setting in the IPA Compile step will be used. The IPA Link step merges and optimizes the application's code, and then divides it into sections for code generation. Each of these sections is a partition. The IPA Link step uses information from the IPA Compile step to determine if a subprogram can be placed in a particular partition. Only compatible subprograms are included in a given partition. Compatible subprograms have the same INITAUTO setting.

The IPA Link step sets the INITAUTO setting for a partition to the specification of the first subprogram that is placed in the partition. It places subprograms that follow in partitions that have the same INITAUTO setting.

You can override the setting of INITAUTO by specifying the option on the IPA Link step. If you do so, all partitions will use that value, and the Prolog section of the IPA Link step listing will display the value.

The Partition Map sections of the IPA Link step listing and the object module display the value of the INITAUTO option.

Predefined macros

- __INITAUTO__ is defined to the hexadecimal constant (0xnnU), including the parentheses, when the INITAUTO compiler option is in effect. Otherwise, it is not defined.
- __INITAUTO_W__ is defined to the hexadecimal constant (0xnnnnnnnnU), including the parentheses, when the INITAUTO compiler option is in effect. Otherwise, it is not defined.

INLINE | NOINLINE

Category

Optimization and tuning

Pragma equivalent

#pragma inline (C only), **#pragma noinline**

#pragma options (inline) (C only), **#pragma options (noinline)** (C only)

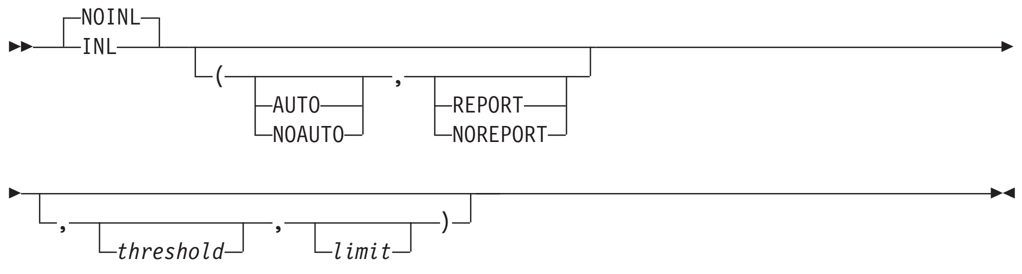
Purpose

Attempts to inline functions instead of generating calls to those functions, for improved performance.

When the INLINE compiler option is in effect, the compiler places the code for selected subprograms at the point of call; this is called *inlining.* It eliminates the linkage overhead and exposes the entire inlined subprogram for optimization by the global optimizer.

When the NOINLINE compiler option is in effect, the compiler generates calls to functions instead of inlining functions.

Syntax



Defaults

For a C/C++ compile:

- If NOOPT is in effect:

```
NOINLINE
  (AUTO,NOREPORT,
  100,1000)
```

- If OPT is in effect:

```
INLINE
  (AUTO,NOREPORT,
  100,1000)
```

NOOPT is the default for C/C++ compile.

For IPA Link:

- If NOOPT is in effect:

```
NOINLINE
  (AUTO,NOREPORT,
  1000,8000)
```

- If OPT is in effect:

```
INLINE
  (AUTO,NOREPORT,
  1000,8000)
```

OPT is the default for IPA Link.

For the c99, c89, cc, and c++ z/OS UNIX System Services utilities, the default is NOINLINE(AUTO,NOREPORT,,).

For the z/OS UNIX utilities, when NOOPT is specified, the default is NOINLINE(AUTO,NOREPORT,100,1000). When OPT is specified, the default is INLINE(AUTO,NOREPORT,100,1000).

In the z/OS UNIX environment, specifying -V, when using the **c89** or **cc** commands, will turn on the REPORT suboption of INLINE. The INLINE option itself is not touched (or changed) by -V.

Parameters

Note: You can specify INLINE without suboptions if you want to use the defaults. You must include a comma between each suboption even if you want to use the default for one of the suboptions. You must specify the suboptions in the following order:

AUTO | NOAUTO

The inliner runs in automatic mode and inlines subprograms within the *threshold* and *limit*.

For C only, if you specify NOAUTO the inliner only inlines those subprograms specified with the **#pragma inline** directive. The **#pragma inline** and **#pragma noinline** directives allow you to determine which subprograms are to be inlined and which are not when the INLINE option is specified. These `#pragma` directives have no effect if you specify NOINLINE. See *z/OS XL C/C++ Language Reference* for more information on `#pragma` directives.

The default is AUTO.

REPORT | NOREPORT

An inline report becomes part of the listing file. The inline report consists of the following:
- An inline summary
- A detailed call structure

You can obtain the same report if you use the INLRPT and OPT options. For more information on the inline report, see "Inline Report" on page 315, "Inline Report" on page 300, and "Inline Report for IPA inliner" on page 326.

The default is NOREPORT.

threshold

The maximum relative size of a subprogram to inline. For C/C++ compiles, the default for *threshold* is 100 Abstract Code Units (ACUs). For the IPA Link step, the default for *threshold* is 1000 ACUs. ACUs are proportional in size to the executable code in the subprogram; the z/OS XL C compiler translates your z/OS XL C code into ACUs. The maximum *threshold* is INT_MAX, as defined in the header file `limits.h`. Specifying a threshold of `0` is the same as specifying NOAUTO.

limit

The maximum relative size a subprogram can grow before auto-inlining stops. For C/C++ compiles, the default for *limit* is 1000 ACUs for a subprogram. For the IPA Link step, the default for *limit* is 8000 ACUs for that subprogram. The maximum for *limit* is INT_MAX, as defined in the header file `limits.h`. Specifying a limit of `0` is equivalent to specifying NOAUTO.

Usage

The INLINE compiler option has the following effects:
- The compiler invokes the compilation unit inliner to perform inlining of functions within the current compilation unit.

- If the compiler inlines all invocations of a static subprogram, it removes the non-inlined instance of the subprogram.
- If the compiler inlines all invocations of an externally visible subprogram, it does not remove the non-inlined instance of the subprogram. This allows callers who are outside of the current compilation unit to invoke the non-inlined instance.
- If you specify INLINE(,REPORT,,) or INLRPT, the compiler generates the Inline Report listing section.

For more information on optimization and the INLINE option, refer to the section about optimizing code in the *z/OS XL C/C++ Programming Guide*.

You can specify the INLINE | NOINLINE option on the invocation line and for C in the **#pragma options** preprocessor directive. When you use both methods at the same time, the compiler merges the suboptions. If a suboption on the invocation line conflicts with a suboption in the **#pragma options** directive, the one on the invocation line takes precedence. If the NOINLINE option is specified on the invocation line and the **#pragma options(inline)** directive is used, the compiler will behave as if the INLINE option is specified. If the INLINE option is specified on the invocation line and the **#pragma options(noinline)** directive is used, the compiler will behave as if the INLINE option is specified.

For example, because you typically do not want to inline your subprograms when you are developing a program, you can use the **#pragma options(noinline)** directive. When you want to inline your subprograms, you can override the **#pragma options(noinline)** by specifying INLINE on the invocation line rather than by editing your source program. The following example illustrates these rules.

Source file:
#pragma options(noinline(noauto,noreport,,2000))

Invocation line:
INLINE (AUTO,,,)

Result:
INLINE (AUTO,NOREPORT,100,2000)

Notes:

1. When you specify the INLINE compiler option, a comment, with the values of the suboptions, is generated in your object module to aid you in diagnosing your program.
2. If the compiler option OPT is specified, INLINE becomes the default.
3. Specify the INLRPT, LIST, or SOURCE compiler options to redirect the output from the INLINE(,REPORT,,) option.
4. If you specify INLINE and TEST:
   at OPT(0), INLINE is ignored
   at OPT, inlining is done
5. If you specify NOINLINE, no subprograms will be inlined even if you have **#pragma inline** directives in your code.
6. If you specify INLINE, subprograms may not be inlined or inline other subprograms when COMPACT is specified (either directly or via **#pragma option_override**). Generate and check the inline report to determine the final status of inlining. The inlining may not occur when OPT(0) is specified via the **#pragma option_override**.

IPA effects

The INLINE option generates inlined code for the regular compiler object; therefore, it affects the IPA Compile step only if you specify IPA(OBJECT). If you specify IPA(NOOBJECT), INLINE has no effect, and there is no reason to use it.

If you specify the INLINE option on the IPA Link step, it has the following effects:

- The IPA Link step invokes the IPA inliner, which inlines subprograms (functions and C++ methods) in the entire program.
- The IPA Link step uses **#pragma inline | noinline** directive information and `inline` subprogram specifier information from the IPA Compile step for source program inlining control. Specifying the INLINE option on the IPA Compile step has no effect on IPA Link step inlining processing.

  You can use the IPA Link control file `inline` and `noinline` directives to explicitly control the inlining of subprograms on the IPA Link step. These directives override IPA Compile step **#pragma inline | noinline** directives and `inline` subprogram specifiers.
- If the IPA Link step inlines all invocations of a subprogram, it removes the non-inlined instance of the subprogram, unless the subprogram entry point was exported using a **#pragma export** directive or the EXPORTALL compiler option, or was retained using the IPA Link control file `retain` directive. IPA Link processes static subprograms and externally visible subprograms in the same manner.

The IPA inliner has the inlining capabilities of the compilation unit inliner. In addition, the IPA inliner detects complex recursion, and may inline it. If you specify the INLRPT option, the IPA Link listing contains the IPA Inline Report section. This section is similar to the report that the compilation unit inliner generates. If you specify NOINLINE(,REPORT,,) or NOINLINE INLRPT, IPA generates an IPA Inline Report section that specifies that nothing was inlined.

Predefined macros

None.

INLRPT | NOINLRPT

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Generates a report on the status of inlined functions.

When the INLRPT compiler option is in effect, the compiler generates a report that provides the status of subprograms that were inlined, specifies whether they were inlined or not and displays the reasons for the action of the compiler.

When the NOINLRPT compiler option is in effect, the generation of a report on the status of inlined functions is suppressed.

Syntax

```
        ┌─NOINLR─┐
►►──────┴─INLR───┴──┬──────────────────────────────────────────────────┬──────────────►◄
                    └─(──┬─Sequential filename──────────────────┬─)─┘
                         ├─Partitioned data set─────────────────┤
                         ├─Partitioned data set (member)────────┤
                         ├─z/OS UNIX System Services filename───┤
                         └─z/OS UNIX System Services directory──┘
```

Defaults

NOINLRPT

In the z/OS UNIX System Services environment, the output of this option goes to `stdout`. This option is turned on by specifying -V.

Parameters

Sequential filename
Specifies the sequential data set file name for the inline report output file.

Partitioned data set
Specifies the partitioned data set for the inline report output file.

Partitioned data set (member)
Specifies the partitioned data set (member) for the inline report output file.

z/OS UNIX System Services filename
Specifies the z/OS UNIX System Services file name for the inline report output file.

z/OS UNIX System Services directory
Specifies the z/OS UNIX System Services directory for the inline report output file.

Usage

If you use the OPTIMIZE option, you can also use INLRPT to specify that the compiler generate a report as part of the compiler listing.

If you do not specify a file name, the compiler uses the SYSCPRT ddname if you allocated one. If you did not allocate SYSCPRT, the compiler uses the source file name to generate a file name.

The NOINLR option can optionally take a file name suboption. This file name then becomes the default. If you subsequently use the INLR option without a file name, the compiler uses the file name that you specified in the earlier specification or NOINLR. For example,

```
CXX HELLO (NOINLR(./hello.lis) INLR OPT
```

is the same as specifying:

```
CXX HELLO (INLR(./hello.lis) OPT
```

Note: If you specify a file name with any of the SOURCE, LIST, or INLRPT options, all the listing sections are combined into the last file name specified.

If you specify this multiple times, the compiler uses the last specified option with the last specified suboption. The following two specifications have the same result:

1. `CXX HELLO (NOINLR(./hello.lis) INLR(./n1.lis)  NOINLR(./test.lis) INLR`
2. `CXX HELLO (INLR(./test.lis)`

IPA effects

If you specify the INLRPT option on the IPA Link step, the IPA Link step listing contains an IPA Inline Report section. Refer to “INLINE | NOINLINE” on page 133 for more information about generating an IPA Inline Report section.

Predefined macros

None.

Related information

For more information on related compiler options, see:

- “OPTIMIZE | NOOPTIMIZE” on page 196
- “SOURCE | NOSOURCE” on page 226
- “LIST | NOLIST” on page 163

IPA | NOIPA

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Enables or customizes a class of optimizations known as interprocedural analysis (IPA).

Syntax

```
   ┌─NOIPA─┐
►►─┴─IPA───┴──────────────────────────────────────────────────────────────►

►─┬─────────────────────────────────────────────────────────────────┬─────►◄
  │  ┌─,─────────────────────────────────────────────────────────┐  │
  │  ▼                                                           │  │
  └──(─┬─NOLINK | LINK─────────────────────────────────────────┬─┴──┘
       ├─OBJ | NOOBJ | OBJONLY─────────────────────────────────┤
       ├─ATTR | NOATT──────────────────────────────────────────┤
       ├─COM|NOCOM─────────────────────────────────────────────┤
       ├─GONUM | NOGONUM───────────────────────────────────────┤
       ├─LIS | NOLIS───────────────────────────────────────────┤
       ├─OPT | NOPT────────────────────────────────────────────┤
       ├─XR | NOXR─────────────────────────────────────────────┤
       ├─LEVEL──┬─(0)─┬────────────────────────────────────────┤
       │        ├─(1)─┤                                        │
       │        └─(2)─┘                                        │
       ├─CONTROL | NOCONTROL──┬──────────────┬─────────────────┤
       │                      └─(─fileid─)───┘                 │
       ├─DUP | NODUP───────────────────────────────────────────┤
       ├─ER | NOER─────────────────────────────────────────────┤
       ├─MAP | NOMAP───────────────────────────────────────────┤
       ├─NCAL | NONCAL─────────────────────────────────────────┤
       ├─UPCASE | NOUPCASE─────────────────────────────────────┤
       ├─PDF1 | NOPDF1─────────────────────────────────────────┤
       ├─PDF2 | NOPDF2─────────────────────────────────────────┤
       └─PDFNAME | NOPDFNAME─(─┬─//data set name─────────────────┬─)─┘
                               └─z/OS UNIX System Services filename─┘
```

Defaults

NOIPA

The default for the c99, c89, cc, c++ z/OS UNIX System Services utilities is:

```
NOIPA(NOCONTROL(ipa.ctl),DUP,NOER,NOMAP,NOUPCASE,NONCAL)IPA(LINK,LEVEL(1)).
```

Parameters

NOLINK
IPA(NOLINK) invokes the IPA Compile step. NOLINK is the default suboption of the IPA option.

LINK
IPA(LINK) invokes the IPA Link step.

ATTRIBUTE | NOATTRIBUTE
Indicates whether the compiler saves information about symbols in the IPA object file. The IPA Link step uses this information if you specify the ATTR or XREF option on that step.

The difference between specifying IPA(ATTR) and specifying ATTR or XREF is that IPA(ATTR) does not generate a Cross Reference or Static Map listing sections after IPA Compile step source analysis is complete. It also does not generate a Storage Offset, Static Map, or External Symbol Cross Reference listing section during IPA Compile step code generation.

The default is IPA(NOATTRIBUTE). The abbreviations are IPA(ATTR|NOATTR). If you specify the ATTR or XREF option, it overrides the IPA(NOATTRIBUTE) option.

COMPRESS | NOCOMPRESS
Indicates that the IPA object information is compressed to significantly reduce the size of the IPA object file.

The default is IPA(COMPRESS). The abbreviations are IPA(COM|NOCOM).

GONUMBER | NOGONUMBER
Indicates whether the compiler saves information about source file line numbers in the IPA object file. The difference between specifying IPA(GONUMBER) and GONUMBER is that IPA(GONUMBER) does not cause GONUMBER tables to be built during IPA Compile step code generation. If the compiler does not build GONUMBER tables, the size of the object module is smaller.

Refer to “GONUMBER | NOGONUMBER” on page 123 for information about the effect of this suboption on the IPA Link step. Refer also to “Interactions between compiler options and IPA suboptions” on page 43.

The default is IPA(NOGONUMBER). The abbreviations are IPA(GONUM|NOGONUM). If you specify the GONUMBER or LIST option, it overrides the IPA(NOGONUMBER) option.

LIST | NOLIST
Indicates whether the compiler saves information about source line numbers in the IPA object file. The difference between specifying IPA(LIST) and LIST is that IPA(LIST) does not cause the IPA Compile step to generate a Pseudo Assembly listing.

Refer to “LIST | NOLIST” on page 163 for information about the effect of this suboption on the IPA Link step. Refer also to “Interactions between compiler options and IPA suboptions” on page 43.

The default is IPA(NOLIST). The abbreviations are IPA(LIS|NOLIS). If you specify the GONUMBER or LIST option, it overrides the IPA(NOLIST) option.

OBJECT | NOOBJECT | OBJONLY
Controls the content of the object file.

- OBJECT

The options IPA(NOLINK,OBJECT) result in an IPA Compile step.

The compiler performs IPA Compile-time optimizations and generates IPA object information for the resulting program information. In addition, the compiler generates non-IPA object code and data that is based on the original program information. Refer to *z/OS XL C/C++ Programming Guide* for a list of optimizations.

The object file may be used by an IPA Link step, a prelink/link, or a bind.

- NOOBJECT

  The options IPA(NOLINK,NOOBJECT) result in an IPA Compile step.

  The compiler performs IPA Compile-time optimizations and generates IPA object information for the resulting program information. No non-IPA object code or data is generated.

  The object file may be used by an IPA Link step only.

- OBJONLY

  The IPA(OBJONLY) compilation is an intermediate level of optimization. This results in a modified regular compile, not an IPA Compile step. Unlike the IPA Compile step, no IPA information is written to the object file.

  During compilation, this step performs the same IPA-specific compile-time optimizations as the IPA Compile step, performs the requested non-IPA optimizations, and then generates optimized object code and data.

  The object file may be used by an IPA Link step, a prelink/link, or a bind. If it is used as input to an IPA Link step, no IPA Link-time optimizations can be performed for this compilation unit because no IPA information is available.

  For all options in this mode, the *Effect on IPA Compile Step* and *Effect on IPA Link Step* considerations do not apply.

The default is IPA(OBJECT). The abbreviations are IPA(OBJ|NOOBJ|OBJO).

OPTIMIZE | NOOPTIMIZE

The default is IPA(OPTIMIZE) If you specify IPA(NOOPTIMIZE), the compiler issues an informational message and turns on IPA(OPTIMIZE). The abbreviations are IPA(OPT|NOOPT) .

IPA(OPTIMIZE) generates information (in the IPA object file) that will be needed by the OPT compiler option during IPA Link processing.

If you specify the IPA(OBJECT), the IPA(OPTIMIZE), and the NOOPTIMIZE option during the IPA Compile step, the compiler creates a non-optimized object module for debugging. If you specify the OPT(1) or OPT(2) option on a subsequent IPA Link step, you can create an optimized object module without first rerunning the IPA Compile step.

XREF | NOXREF

Indicates whether the compiler saves information about symbols in the IPA object file that will be used in the IPA Link step if you specify ATTR or XREF on that step.

The difference between specifying IPA(XREF) and specifying ATTR or XREF is that IPA(XREF) does not cause the compiler to generate a Cross Reference or Static Map listing sections after IPA Compile step source analysis is complete. It also does not cause the compiler to generate a Storage Offset, Static Map, or External Symbol Cross Reference listing section during IPA Compile step code generation.

Refer to “XREF | NOXREF” on page 264 for information about the effects of this suboption on the IPA Link step.

The default is IPA(NOXREF). The abbreviations are IPA(XR|NOXR). If you specify the ATTR or XREF option, it overrides the IPA(NOXREF) option.

IPA(LINK) invokes the IPA Link step. Only the following IPA suboptions affect the IPA Link step. If you specify other IPA suboptions, they do not affect the IPA Link step.

CONTROL[(fileid)] | NOCONTROL[(fileid)]
Specifies whether a file that contains IPA directives is available for processing. You can specify an optional `fileid`. If you specify both IPA(NOCONTROL(*fileid*)) and IPA(CONTROL), in that order, the IPA Link step resolves the option to IPA(CONTROL(*fileid*)).

The default `fileid` is `DD:IPACNTL` if you specify the IPA(CONTROL) option. The default is IPA(NOCONTROL).

DUP | NODUP
Indicates whether the IPA Link step writes a message and a list of duplicate symbols to the console.

The default is IPA(DUP).

ER | NOER
Indicates whether the IPA Link step writes a message and a list of unresolved symbols to the console.

The default is IPA(NOER).

LEVEL(0|1|2)
Indicates the level of IPA optimization that the IPA Link step should perform after it links the object files into the call graph.

If you specify LEVEL(0), IPA performs subprogram pruning and program partitioning only.

If you specify LEVEL(1), IPA performs all of the optimizations that it does at LEVEL(0), as well as subprogram inlining and global variable coalescing. IPA performs more precise alias analysis for pointer dereferences and subprogram calls.

Under IPA Level 1, many optimizations such as constant propagation and pointer analysis are performed at the intraprocedural (subprogram) level. If you specify LEVEL(2), IPA performs specific optimizations across the entire program, which can lead to significant improvement in the generated code.

The compiler option OPTIMIZE that you specify on the IPA Link step controls subsequent optimization for each partition during code generation. Regardless of the optimization level you specified during the IPA Compile step, you can request IPA optimization, regular code generation optimization, both, or neither, on the IPA Link step.

The default is IPA(LEVEL(1)).

MAP | NOMAP
Specifies that the IPA Link step will produce a listing. The listing contains a Prolog and the following sections:
- Object File Map
- Compiler Options Map
- Global Symbols Map (which may or may not appear, depending on how much global coalescence was done during optimization)
- Partition Map for each partition
- Source File Map

The default is IPA(NOMAP).

See “Using the IPA Link step listing” on page 317 for more information.

NCAL | NONCAL

Indicates whether the IPA Link step performs an automatic library search to resolve references in files that the IPA Compile step produces. Also indicates whether the IPA Link step performs library searches to locate an object file or files that satisfy unresolved symbol references within the current set of object information.

This suboption controls both explicit searches triggered by the LIBRARY IPA Link control statement, and the implicit SYSLIB search that occurs at the end of IPA Link input processing.

To help you remember the difference between NCAL and NONCAL, you may wish to think of NCAL as "nocall" and NONCAL as "no nocall", (or "call").

The default is IPA(NONCAL).

UPCASE | NOUPCASE

Determines whether the IPA Link step makes an additional automatic library call pass for SYSLIB if unresolved references remain at the end of standard IPA Link processing. Symbol matching is not case sensitive in this pass.

This suboption provides support for linking assembly language object routines, without forcing you to make source changes. The preferred approach is to add **#pragma map** definitions for these symbols, so that the correct symbols are found during normal IPA Link automatic library call processing.

The default is IPA(NOUPCASE). The abbreviations are IPA(UPC|NOUPC).

PDF1 | NOPDF1, PDF2 | NOPDF2, PDFNAME | NOPDFNAME

The default is IPA(NOPDF1, NOPDF2, NOPDFNAME).

Note: IPA(PDF) applies to both the IPA Link and IPA Compile steps. IPA(PDF) requires that the RENT (C only) compiler option is active. This requirement has no effect on C++ since C++ code behaves as if RENT was specified.

PDF (Profile-Directed Feedback) is a suboption of IPA that enables you to use the results from sample program execution to improve optimization near conditional branches and in frequently executed code sections. PDF allows you to gather information about the critical paths and the usage of various parts of the application. PDF passes this information to the compiler so that the optimizer can work to make these critical paths faster. This is a three stage process that involves:

1. Performing a full IPA build with the PDF1 and PDFNAME suboptions
2. Running the trial application with representative data
3. Performing another full IPA build with the PDF2 suboption (the file indicated by PDFNAME holds the profile generated when the code was run in step 2)

Note: The trial application built from the IPA(PDF1) compiler option can only be run on the current system.

The following list describes each of the IPA(PDF) suboptions:

PDF1

An IPA suboption specified during IPA Compile and Link steps. It tells IPA to prepare the application to collect profile information.

PDF2
An IPA suboption specified during IPA Compile and Link steps. This option tells IPA to use the profile information that is provided when optimizing the application.

PDFNAME
This IPA suboption should be used with PDF1 and PDF2 to provide the name of the file that will be used for the profile information.

PDFDIR
This environment variable can be used when using IPA(PDF) in the z/OS UNIX System Services shell. It is used to specify the directory for the profile file.

Usage

The following information describes how to use IPA(PDF).

Before you begin: IPA(PDF) requires that you compile the entire application twice and is intended to be used after other debugging and tuning is finished. IPA(PDF) compiles should be performed during one of the last steps before putting the application into production. The following is a list of restrictions that applies to the procedures that follow:

- You must compile the main program with PDF for profiling information to be collected at run time.
- Do not compile or run two different applications that use the same PDFDIR directory at the same time, unless you have used the PDFNAME(filename) option to distinguish the sets of profiling information.
- You must use the same set of compiler options at all compilation steps for a particular program otherwise PDF cannot optimize your program correctly and may even slow it down.
- You must ensure the profiling information that is provided to the compiler during the PDF2 step is for the application you are tuning.
  - If a non-qualified data set name is provided, the same userid that runs the application to collect the profiling information must perform the PDF2 step.
  - If PDFDIR is set for the PDF2 step, it must name a directory where the actual profiling information can be found.
- The profiling information is placed in the file specified by the PDFNAME(*filename*) suboption, where *filename* can be a z/OS UNIX System Services file name or a z/OS data set name (physical sequential data set or a member of a partitioned data set). For a data set name, it can be fully qualified such as `PDFNAME="//'HLQ.PDF'"` or non-qualified such as `PDFNAME=//PDF`, in which case the actual data set name will be `userid.PDF`, where the userid identifies the user who is executing the application built with PDF1 and building the application with PDF2. The key DCB attributes are `RECFM=U` and `LRECL=0`. In the z/OS UNIX environment, the profile is placed in the current working directory or in the directory that the PDFDIR environment variable names, if that variable is set. If PDFNAME(*filename*) is not specified, the default file name is `PDFNAME(@@PDF)`. This file is referred to as the PDF file.
- If PDFNAME is not provided, then you need to ensure that the same environment (z/OS UNIX System Services, batch/TSO, POSIX mode) is used to collect the profiling information and to perform the PDF1 or PDF2 steps. This is because PDFNAME will default to `@@PDF` and thus the actual location of the file is based on the environment. For example, when the POSIX(ON) run-time option is used, the PDF file will be `./@@PDF` and when the POSIX(OFF) run-time option is

used, the PDF file will be in data set `userid.@@PDF`. In order to eliminate unnecessary confusion, it is recommended that an explicit PDF file name always be provided.

- The compiler makes an attempt to delete the PDF file during PDF1 IPA(LINK) processing.
- If PDFNAME names a data set, it is strongly recommended that the data set physically exist and be allocated with sufficient space before step 2 in the process. Since the actual space required is based on the complexity of the application and the amount of the test data, you may run into a situation where the pre-allocated space is insufficient and you need to re-allocate the data set with larger space. The recommended attributes for the PDF data set are: `RECFM=U LRECL=0`.
- If you do compile a program with PDF1, it will generate profiling information when it runs, which involves some performance overhead. This overhead goes away when you recompile with PDF2 or with no PDF (NOPDF1 and NOPDF2).
- The CCNXPD1B, CCNPD1B, CCNPD1B, and CCNQPD1B PROCs have been created to help the batch user link with the libraries required for IPA(PDF1). Unlike the default link PROCs, these PROCs will statically bind the libraries to support correct operation of the information capture runs.
- Applications built with IPA(PDF1) should not be put into production because the application will be slower due to the instrumented code. The application will lose its natural reentrancy due to the sharing of the global data between the application and the statically bound PDF run-time code.

Perform the following steps to use IPA(PDF):

1. Compile some or all of the source files in a program with the IPA(PDF1) suboption on both the IPA Compile and IPA Link steps. You need to specify the OPTIMIZE(2) option, or preferably the OPTIMIZE(3) option, and the IPA(LEVEL(1|2)) option. Pay special attention to the compiler options that you use to compile the files, because you will need to use the same options later. In a large application, concentrate on those areas of the code that can benefit most from optimization, which are the paths that are executed most in a typical run of the program. Use sample data for profiling that reflects the typical runs that end users make and then the generated profiling information will show the most used paths, and the optimizer will be able to make these paths as fast as possible. You do not need to compile all of the application’s code with the PDF1 suboption but you need to compile the main function with the PDF1 suboption. Link the program using CCNPD1B, CCNXPD1B, or CCNQPD1B in batch, or the **-Wl,PDF** option in the z/OS UNIX System Services shell.
2. Run the program built from step 1 all the way through using a typical data set. The program records profiling information when it finishes. You can run the program multiple times with different input data sets, and the profiling information is accumulated to provide a count of how often branches are taken and blocks of code are executed, based on the input data sets used.

   Note: Use data that is representative of the data that will be used during a normal run of your finished program.
3. Re-build your program using the identical set of source files with the identical compiler options that you used in step 1, but change PDF1 to PDF2. This must be done with the same compiler you use in step 1. In this second stage, the accumulated profiling information is used to fine-tune the optimizations. The resulting program does not contain profiling overhead.

PDF1 at IPA Compile step causes IPA to place an indicator in the IPA object so the functions in the compilation unit are instrumented during the IPA Link step. PDF2 at IPA Compile step causes IPA to place an indicator in the IPA object so the functions in the compilation unit are optimized based on the profiling information.

PDF1 at IPA Link step causes IPA to insert instrumentation in the application code. PDF2 at IPA Link step causes IPA to optimize the application based on the profiling information collected in the file specified by PDFNAME.

IPA effects

Only the following IPA suboptions affect the IPA Compile step:

- ATTRIBUTE | NOATTRIBUTE
- COMPRESS | NOCOMPRESS
- GONUMBER | NOGONUMBER
- LIST | NOLIST
- OBJECT | NOOBJECT | OBJONLY
- OPTIMIZE | NOOPTIMIZE
- XREF | NOXREF
- CONTROL | NOCONTROL
- DUP | NODUP
- ER | NOER
- LEVEL
- MAP | NOMAP
- NCAL | NONCAL
- UPCASE | NOUPCASE
- PDF1 | NOPDF1 | PDF2 | NOPDF2 | PDFNAME | NOPDFNAME

You can specify other IPA suboptions, but they do not affect the IPA Compile step.

Predefined macros

None.

Related information

Refer to *z/OS XL C/C++ Programming Guide* for an overview, examples, and more details about Interprocedural Analysis.

KEYWORD | NOKEYWORD (C++ only)

Category

Language element control

Pragma equivalent

None.

Purpose

Controls whether the specified *name* is treated as a keyword or an identifier whenever it appears in your C++ source.

When the KEYWORD compiler option is in effect, the compiler treats the specified *name* as a keyword.

When the NOKEYWORD compiler option is in effect, the compiler treats the specified *name* as an identifier.

Syntax

```
                  ┌─,─────┐
                  ▼       │
      ┌─KEYWORD──(──name──┴─)─┐
►►────┴─NOKEYWORD─────────────┴──────────────────────────────────────────────►◄
```

Defaults

Recognizes all C++ keywords.

Parameters

name
    The name of a keyword. This suboption is case-sensitive.

Usage

By default, all the built-in keywords defined in the C++ standard are reserved as keywords. You cannot add keywords to the C++ language with this option. However, you can use it to enable built-in keywords that have been disabled using NOKEYWORD(string).

Predefined macros

- If you specify NOKEYWORD(bool), the __BOOL__ macro is undefined.
- __C99_RESTRICT is predefined to 1 when the KEYWORD(restrict) compiler option is in effect.
- __IBM__TYPEOF__ is predefined to 1 when the KEYWORD(typeof) compiler option is in effect.

LANGLVL

Category

Language element control

Pragma equivalent

#pragma langlvl (C only)

Purpose

Determines whether source code and compiler options should be checked for conformance to a specific language standard, or subset or superset of a standard.

Syntax

Category: Language element control for C

```
              ┌─EXTENDED─┐
►►──LANG──(───┼─COMMONC──┼──)──────────────────────────────────────────────►◄
              ├─ANSI─────┤
              ├─EXTC89───┤
              ├─EXTC99───┤
              ├─SAA──────┤
              ├─SAAL2────┤
              ├─STDC89───┤
              ├─STDC99───┤
              ├─LIBEXT───┤
              └─LONGLONG─┘
```

Category: Language element control for C++

```
          ┌─EXTENDED────────────────┐
►►──LANG──(─┼─EXTENDED0X──────────────┼─)──────────────────────────────────────►◄
          ├─COMPAT92────────────────┤
          ├─ANSI────────────────────┤
          ├─STRICT98────────────────┤
          ├─ANONSTRUCT──────────────┤
          ├─ANONUNION───────────────┤
          ├─ANSIFOR─────────────────┤
          ├─ANSISINIT───────────────┤
          ├─C99__FUNC__─────────────┤
          ├─C99VLA──────────────────┤
          ├─DBCS────────────────────┤
          ├─DEPENDENTBASELOOKUP─────┤
          ├─DOLLARINNAMES───────────┤
          ├─EMPTYSTRUCT─────────────┤
          ├─EXTENDEDFRIEND──────────┤
          ├─EXTERNTEMPLATE──────────┤
          ├─GNU_INCLUDE_NEXT────────┤
          ├─ILLPTOM─────────────────┤
          ├─IMPLICITINT─────────────┤
          ├─LIBEXT──────────────────┤
          ├─LONGLONG────────────────┤
          ├─NEWEXCP─────────────────┤
          ├─OFFSETNONPOD────────────┤
          ├─OLDDIGRAPH──────────────┤
          ├─OLDFRIEND───────────────┤
          ├─OLDMATH─────────────────┤
          ├─OLDSTR──────────────────┤
          ├─OLDTEMPACC──────────────┤
          ├─OLDTMPLALIGN────────────┤
          ├─OLDTMPLSPEC─────────────┤
          ├─TRAILENUM───────────────┤
          ├─TYPEDEFCLASS────────────┤
          ├─UCS─────────────────────┤
          └─ZEROEXTARRAY────────────┘
```

Defaults

LANGLVL(EXTENDED)

For the z/OS UNIX System Services utilities, the defaults are as follows:

- For the **c99** command:
  - LANGLVL(STDC99)
- For the **c89** command:
  - LANGLVL(ANSI)
- For the **cc** command:
  - LANGLVL(COMMONC)
- For the **c++** command:
  - LANGLVL(EXTENDED, NOLIBEXT, NOLONGLONG)

Parameters

The following suboptions are only available under z/OS XL C:

COMMONC
It indicates language constructs that are defined by XPG, many of which

LANGLVL(EXTENDED) already supports. LANGLVL(ANSI) and LANGLVL(EXTENDED) do not support the following, but LANGLVL(COMMONC) does:

- Unsignedness is preserved for standard integral promotions (that is, `unsigned char` is promoted to `unsigned int`)
- Trigraphs within literals are not processed
- `sizeof` operator is permitted on bit fields
- Bit fields other than `int` are tolerated, and a warning message is issued
- Macro parameters within quotation marks are expanded
- The empty comment in a subprogram-like macro is equivalent to the ANSI/ISO token concatenation operator

The macro __COMMONC__ is defined as 1 when you specify LANGLVL(COMMONC).

If you specify LANGLVL(COMMONC), the ANSIALIAS option is automatically turned off. If you want ANSIALIAS turned on, you must explicitly specify it.

Note: The option ANSIALIAS assumes code that supports ANSI. Using LANGLVL(COMMONC) and ANSIALIAS together may have undesirable effects on your code at a high optimization level. See "ANSIALIAS | NOANSIALIAS" on page 66 for more information.

EXTC89
Indicates language constructs that are defined by the ISO C89 standard, plus additional orthogonal language extensions that do not alter the behavior of this standard.

EXTC99
Indicates language constructs that are defined by the ISO C99 standard, plus additional orthogonal language extensions that do not alter the behavior of the standard.

SAA
Indicates language constructs that are defined by SAA.

SAAL2
Indicates language constructs that are defined by SAA Level 2.

STDC89
Indicates language constructs that are defined by the ISO C89 standard. This suboption is synonymous with LANGLVL(ANSI).

STDC99
Indicates language constructs that are defined by the ISO C99 standard.

The following suboptions are available under z/OS XL C and z/OS XL C++:

EXTENDED
It indicates all language constructs are available with z/OS XL C. It enables extensions to the ISO C standard. The macro __EXTENDED__ is defined as 1.

| **Note:** Under z/OS XL C, the unicode literals are enabled under the
| EXTENDED language level, and disabled under the strictly-conforming
| language levels. When the unicode literals are enabled, the macro
| __IBM_UTF_LITERAL is predefined to 1. Otherwise, this macro is not
| predefined.

ANSI
Use it if you are compiling new or ported code that is ISO C/C++ compliant. It indicates language constructs that are defined by ISO. Some non-ANSI stub

routines will exist even if you specify LANGLVL(ANSI), for compatibility with previous releases. The macro __ANSI__ is defined as 1 for C only. It is intended to ensure that the compilation conforms to the ISO C and C++ standards.

Note: When you specify LANGLVL(ANSI), the compiler can still read and analyze the `_Packed` keyword in z/OS XL C/C++. If you want to make your code purely ANSI, you should redefine `_Packed` in a header file as follows:

```
#ifdef __ANSI__
  #define _Packed
#endif
```

The compiler will now see the `_Packed` attribute as a blank when LANGLVL(ANSI) is specified at compile time, and the language level of the code will be ANSI.

LIBEXT | NOLIBEXT

Specifying this option affects the C/C++ run-time provided headers, which in turn control the availability of general ISO run-time extensions. In addition, it also defines the following macros and sets their values to 1:

- _MI_BUILTIN (this macro controls the availability of machine built-in instructions. Refer to the section on using built-in functions in *z/OS XL C/C++ Programming Guide*)
- _EXT (this macro controls the availability of general ISO run-time extensions)

The default for C is LANG(LIBEXT) and for C++ is LANG(NOLIBEXT). However, LANG(LIBEXT) is implicitly enabled in C by LANG(COMMONC | SAA | SAAL2 | EXTENDED | EXTC89 | EXTC99) and in C++ by LANG(EXTENDED | COMPAT92).

LONGLONG | NOLONGLONG

This option controls the availability of long long integer types for your compilation. The default for C is LANG(LONGLONG) and for C++ is LANG(NOLONGLONG).

| Four predefined option groups are provided for commonly used settings for C++.
| These groups are:

LANGLVL(COMPAT92)
Use this option group if your code compiles with z/OS V1R1 and you want to move to z/OS V1R2 with minimal changes. This group is the closest you can get to the old behavior of the previous compilers.

LANGLVL(STRICT98) or LANGLVL(ANSI)
These two option groups are identical. Use them if you are compiling new or ported code that is ISO C++ compliant. They indicate language constructs that are defined by ISO. Some non-ANSI stub routines will exist even if you specify LANGLVL(ANSI), for compatibility with previous releases.

LANGLVL(EXTENDED)
This option group indicates all language constructs available with z/OS XL C++. It enables extensions to the ISO C/C++ standard. The macro __EXTENDED__ is defined as 1.

LANGLVL(EXTENDED0X)
: This group option compiles code using all the C++ and currently supported C++0x features that are implemented in IBM z/OS V1R11 XL C/C++ compiler.

The options and settings included in the COMPAT92, STRICT98/ANSI, EXTENDED and EXTENDED0X groups are listed in Table 24. Except for TMPLPARSE, all settings have a value of either on (meaning the suboption or option is enabled) or off (meaning the suboption or option is not enabled).

Table 24. Compatibility options for z/OS XL C++ compiler

Options	**Group names**			
	compat92	**strict98/ ansi**	**extended**	**extended 0x**
KEYWORD(bool) | NOKEYWORD(bool)	Off	On	On	On
KEYWORD(explicit) | NOKEYWORD(explicit)	Off	On	On	On
KEYWORD(export) | NOKEYWORD(export)	Off	On	On	On
KEYWORD(false) | NOKEYWORD(false)	Off	On	On	On
KEYWORD(mutable) | NOKEYWORD(mutable)	Off	On	On	On
KEYWORD(namespace) | NOKEYWORD(namespace)	Off	On	On	On
KEYWORD(true) | NOKEYWORD(true)	Off	On	On	On
KEYWORD(typename) | NOKEYWORD(typename)	Off	On	On	On
KEYWORD(using) | NOKEYWORD(using)	Off	On	On	On
LANGLVL(ANONSTRUCT | NOANONSTRUCT)	Off	Off	On	On
LANGLVL(ANONUNION | NOANONUNION)	On	Off	On	On
LANGLVL(ANSIFOR | NOANSIFOR)	Off	On	On	On
LANGLVL(ANSISINIT | NOANSISINIT)	Off	On	On	On
LANGLVL(C99__FUNC__ | NOC99__FUNC__)	Off	Off	On	On
LANGLVL(C99VLA | NOC99VLA)	Off	Off	On	On
LANGLVL(EMPTYSTRUCT | NOEMPTYSTRUCT)	On	Off	On	On
LANGLVL(EXTENDEDFRIEND | NOEXTENDEDFRIEND)	Off	Off	Off	On
LANGLVL(EXTERNTEMPLATE | NOEXTERNTEMPLATE)	Off	Off	On	On
LANGLVL(GNU_INCLUDE_NEXT | NOGNU_INCLUDE_NEXT)	On	On	On	On

Table 24. Compatibility options for z/OS XL C++ compiler (continued)

Options	**Group names**			
	compat92	**strict98/ ansi**	**extended**	**extended 0x**
LANGLVL(ILLPTOM | NOILLPTOM)	On	Off	On	On
LANGLVL(IMPLICITINT | NOIMPLICITINT)	On	Off	On	On
LANGLVL(LIBEXT | NOLIBEXT)	On	Off	On	On
LANGLVL(LONGLONG | NOLONGLONG)	On	Off	On	Off
LANGLVL(OFFSETNONPOD | NOOFFSETNONPOD)	On	Off	On	On
LANGLVL(OLDDIGRAPH | NOOLDDIGRAPH)	Off	On	Off	Off
LANGLVL(OLDFRIEND | NOOLDFRIEND)	On	Off	On	On
LANGLVL(OLDMATH | NOOLDMATH)	On	Off	Off	Off
LANGLVL(OLDSTR | NOOLDSTR)	On	Off	Off	Off
LANGLVL(OLDTEMPACC | NOOLDTEMPACC)	On	Off	On	On
LANGLVL(OLDTMPLALIGN | NOOLDTMPLALIGN)	On	Off	Off	Off
LANGLVL(OLDTMPLSPEC | NOOLDTMPLSPEC)	On	Off	On	On
LANGLVL(TRAILENUM | NOTRAILENUM)	On	Off	On	On
LANGLVL(TYPEDEFCLASS | TYPEDEFCLASS)	On	Off	On	On
LANGLVL(ZEROEXTARRAY | NOZEROEXTARRAY)	Off	Off	On	On
RTTI | NORTTI	Off	On	On	On
TMPLPARSE(NO | ERROR | WARN)	NO	WARN	NO	NO

The following suboptions are only available under C++:

ANONSTRUCT | NOANONSTRUCT

This option controls whether anonymous structs and anonymous classes are allowed in your C++ source. When LANG(ANONSTRUCT) is specified, z/OS XL C++ allows anonymous structs. This is an extension to the C++ standard.

Example: Anonymous structs typically are used in unions, as in the following code example:

```
union U {
   struct {
      int i:16;
      int j:16;
   };
   int k;
} u;
// ...
u.j=3;
```

When LANG(ANONSTRUCT) is in effect, you receive a warning if your code declares an anonymous struct. You can suppress the warning with SUPPRESS(CCN5017). When you build with LANG(NOANONSTRUCT) an anonymous struct is flagged as an error. Specify LANG(NOANONSTRUCT) for compliance with ISO standard C++. The default is LANG(ANONSTRUCT).

ANONUNION | NOANONUNION

This option controls what members are allowed in anonymous unions. When LANG(ANONUNION) is in effect, anonymous unions can have members of all types that ISO standard C++ allows in non-anonymous unions. For example, non-data members, such as structs, typedefs, and enumerations are allowed. Member functions, virtual functions, or objects of classes that have non-trivial default constructors, copy constructors, or destructors cannot be members of a union, regardless of the setting of this option. When LANG(ANONUNION) is in effect, z/OS XL C++ allows non-data members in anonymous unions. This is an extension to ISO standard C++. When LANG(ANONUNION) is in effect, you receive a warning if your code uses the extension, unless you suppress the message with SUPPRESS(CCN6608). Specify LANG(NOANONUNION) for compliance with ISO standard C++. The default is LANG(ANONUNION).

ANSIFOR | NOANSIFOR

This option controls whether scope rules defined in the C++ standard apply to names declared in for-init statements. By default, ISO standard C++ rules are used.

Example: The following code causes a name lookup error:

```
{
   //...
   for (int i=1; i<5; i++) {
      cout << i * 2 << endl;
   }
   i = 10;  // error
 }
```

The reason for the error is that `i`, or any name declared within a for-init-statement, is visible only within the `for` statement. To correct the error, either declare `i` outside the loop or specify LANG(NOANSIFOR). Specify LANG(NOANSIFOR) to allow old language behavior. The default is LANG(ANSIFOR).

ANSISINIT | NOANSISINIT

This suboption can be used to select between the old (prior to z/OS V1R1) and the current (z/OS V1R2 or later) compiler behaviors. It is useful for building an application that includes an existing DLL originally built with a z/OS V1R1 or earlier version of the z/OS XL C/C++ compilers. Specifying the NOANSISINIT suboption, will cause the behavior of global (including static locals) objects with destructors in the newly-compiled objects to be compatible with objects built with earlier compilers.

If you specify the LP64 option and the LANGLVL(NOANSISINIT) option, the compiler issues a warning, ignores the LANGLVL(NOANSISINIT) option and turns on the LANGLVL(ANSISINIT) option.

The default setting is LANGLVL(ANSISINIT).

Note: LANGLVL(EXTENDED) and LANVLVL(ANSI) set LANGLVL(ANSISINIT). LANGLVL(COMPAT92) sets LANGLVL(NOANSISINIT).

C99__FUNC__ | NOC99__FUNC__
This option provides an alternative method for debugging programs by identifying the names of functions where the `__func__` identifier is used.

C99VLA | NOC99VLA
This option controls variable length arrays. The default is LANG(C99VLA).

DBCS | NODBCS
This option controls whether multi-byte characters are accepted in string literals and in comments. The default is LANG(NODBCS).

DEPENDENTBASELOOKUP | NODEPENDENTBASELOOKUP
This option controls whether to apply the name lookup rules for a template base class of dependent type, which is defined in the Technical Corrigendum 1 (TC1) of the C++ Standard. Specify LANG(NODEPENDENTBASELOOKUP) for compliance with TC1. When LANG(NODEPENDENTBASELOOKUP) is in effect, unqualified names in a template class will not be resolved in a base class if that base class is dependent on a template parameter. These names must be qualified with the base class name in order to be found by name lookup.

The following example shows code that compiles with LANG(NODEPENDENTBASELOOKUP):

```
struct base
{
          int baseName;
};

template <class B> struct derived : public B
{
   void func()
    {
    int i = baseName;     // this name will not be found in the base class
    };
};

int main(void)
{
   derived<base> x;
   x.func();
   return 0;
}
```

The following example produces the same compile result whether LANG(NODEPENDENTBASELOOKUP) is used or not:

```
struct base
{
          int baseName;
};

template <class B> struct derived : public B
{
   void func()
    {
    int i = B::baseName;   // qualified name will be found in the base class
    };
};

int main(void)
{
   derived<base> x;
   x.func();
   return 0;
}
```

| The default is LANG(DEPENDENTBASELOOKUP). When the default option is
| in effect, the behavior of previous XL C++ compilers remains.

DOLLARINNAMES | NODOLLARINNAMES

This option controls whether the dollar-sign character ($) is allowed in identifiers. If LANG(NODOLLARINNAMES) is in effect, dollar sign characters in identifiers are treated as syntax errors. The default is LANG(NODOLLARINNAMES).

Note: In the z/OS UNIX System Services environment, LANG(DOLLARINNAMES) is specified by using the **-qdollar** option with the **xlc** command.

EMPTYSTRUCT | NOEMPTYSTRUCT

This option instructs the compiler to tolerate empty member declarations in structs. ISO C++ does not permit empty member declaration in structs.

Example: When LANG(NOEMPTYSTRUCT) is in effect , the following example will be rejected by the compiler:

```
struct S {
      ;  // this line is ill-formed
};
```

The default is LANG(NOEMPTYSTRUCT).

| **EXTENDED0X**

| ▶ C++0x This group option is created to compile code using all the C++ and
| currently supported C++0x features that are implemented in IBM z/OS V1R11
| XL C/C++ compiler. Specifically, additional integer promotion rules for arithmetic
| conversions with `long long` data types are introduced in this release and the
| C++ preprocessor is enhanced to support C99 features.

| **Notes:**

1. | C++0x is a new version of the standard for the C++ programming language. This is a draft standard and has not been officially adopted in its entirety. Note that future levels of support for this standard are likely to change. The implementation of this language level is based on IBM's interpretation of the draft C++0x standard. and is subject to change at any time without notice. IBM will make no attempt to maintain compatibility with earlier releases, in source or binary, of the new C++0x LANGLVL suboptions (their names or their semantics) and therefore they should not be relied on as a stable programming interface. See *z/OS XL C/C++ Language Reference* for more information.
2. | Under z/OS XL C++, the unicode literals and character types are enabled under the EXTENDED and EXTENDED0X language levels, and disabled under the other language levels. When the unicode literals are enabled, the macros __IBM_UTF_LITERAL and __IBMCPP_UTF_LITERAL__ are predefined to 1. Otherwise, they are not predefined. Under the EXTENDED0X language level, the keywords char16_t and char32_t are enabled by default.

| **EXTENDEDFRIEND | NOEXTENDEDFRIEND**

| ▶ C++0x This option controls whether C++98 or C++0x friend declarations are
| used. With this option, you can name template parameters and `typedef` names
| as friends. Basic types can also be allowed to appear in friend declarations in
| the C++0x standard. The `class` keyword in the context for friend declarations is
| removed in the C++0x standard, which differs from the C++98 friend class

declaration syntax where the `class` keyword is necessary. This greatly improves the generality of templates and friend declarations. The default is LANG(NOEXTENDEDFRIEND).

Notes:

1. This feature can also be enabled by the group option LANGLVL(EXTENDED0X). If either LANGLVL(EXTENDED0X) or LANGLVL(EXTENDEDFRIEND) is in effect, the macro __IBMCPP_EXTENDED_FRIEND will be defined to 1; otherwise, it is undefined.
2. LANGLVL(EXTENDEDFRIEND) is implied in the group option LANGLVL(EXTENDED0X). If multiple LANGLVL suboptions are applied, the last one takes effect.

EXTERNTEMPLATE | NOEXTERNTEMPLATE

C++0x This option enables explicit instantiation declarations, which provide you with the ability to suppress the implicit instantiation of template specializations or members of the same. Compile time can be reduced through the use of explicit instantiation declarations. The default is LANGLVL(EXTERNTEMPLATE).

Notes:

1. When explicit instantiation declaration is enabled, the compiler defines the macro __IBMCPP_EXTERN_TEMPLATE as the preprocessing number 1; otherwise, the macro is undefined. In both cases, the macro is protected. When the macro is undefined or redefined, the compiler issues a warning.
2. LANGLVL(EXTERNTEMPLATE) is implied in the group options of LANGLVL(EXTENDED) and LANGLVL(EXTENDED0X). If multiple LANGLVL suboptions are specified, the last one takes effect.

GNU_INCLUDE_NEXT | NOGNU_INCLUDE_NEXT

This option is provided as a GNU C++ portability option to enable or disable support for the GNU **#include_next** preprocessor directive. The default is LANG(GNU_INCLUDE_NEXT).

ILLPTOM | NOILLPTOM

This controls what expressions can be used to form pointers to members. The compiler accepts some forms that are in common use, but do not conform to the C++ standard. When LANG(ILLPTOM) is in effect, the compiler allows these forms.

Example: The following code defines the pointer to a function member, `p`, and initializes the address of `C::foo`, in the old style:

```
struct C {
void foo(init);
};

void (C::*p) (int) = C::foo;
```

Specify LANG(NOILLPTOM) for compliance with the C++ standard.

Example: The example must be modified to use the & operator:

```
struct C {
void foo(int);
};

void (C::*p) (int) = &C::foo;
```

The default is LANG(ILLPTOM).

IMPLICITINT | NOIMPLICITINT

This option controls whether z/OS XL C++ will accept missing or partially specified types as implicitly specifying `int`. This is no longer accepted in the standard but may exist in legacy code. When LANG(NOIMPLICITINT) is specified, all types must be fully specified. Also, when LANG(IMPLICITINT) is specified, a function declaration at namespace scope or in a member list will implicitly be declared to return `int`. Also, any declaration specifier sequence that does not completely specify a type will implicitly specify an integer type. Note that the effect is as if the `int` specifier were present. This means that the specifier `const`, by itself, would specify a constant integer. The following specifiers do not completely specify a type:

- `auto`
- `const`
- `extern`
- `extern "<literal>"`
- `inline`
- `mutable`
- `friend`
- `register`
- `static`
- `typedef`
- `virtual`
- `volatile`
- platform specific types (for example, `_cdecl`, `__declspec`)

Note that any situation where a type is specified is affected by this option. This includes, for example, template and parameter types, exception specifications, types in expressions (eg, casts, `dynamic_cast`, `new`), and types for conversion functions. By default, LANG(EXTENDED) sets LANG(IMPLICITINT). This is an extension to the C++ standard.

Example: The return type of function `MyFunction` is `int` because it was omitted in the following code:

```
MyFunction()
{
   return 0;
}
```

Specify LANG(NOIMPLICITINT) for compliance with ISO standard C++.

Example: The function declaration must be modified to:

```
int MyFunction()
{
   return 0;
}
```

The default is LANG(IMPLICITINT).

NEWEXCP | NONEWEXCP

This option determines whether or not the C++ operator `new` throws an exception. When LANGLVL(NEWEXCP) is specified, the standard exception `std::bad_alloc` is thrown when the requested memory allocation fails. This option does not apply to the nothrow versions of the `new` operator. The default setting is NONEWEXCP. This option governs the behavior of the default versions of the standard `new` operators. This option does apply to the throw versions of the `new` operator except for class-specific `new` operators, user-defined `new` operators, and `new` operators with placement arguments.

OFFSETNONPOD | NOOFFSETNONPOD

This option controls whether the `offsetof` macro can be applied to classes that are not data-only. C++ programmers often casually call data-only classes ″Plain Old Data″ (POD) classes. By default, LANG(EXTENDED) allows `offsetof` to be used with non-POD classes. This is an extension to the C++ standard. When LANG(OFFSETNONPOD) is in effect , you receive a warning if your code uses the extension, unless you suppress the message with SUPPRESS(CCN6281). Specify LANG(NOOFFSETNONPOD) for compliance with ISO standard C++. Specify LANG(OFFSETNONPOD) if your code applies offsetof to a class that contains one of the following:

- User-declared constructors or destructors
- User-declared assignment operators
- Private or protected non-static data members
- Base classes
- Virtual functions
- Non-static data members of type pointer to member
- A `struct` or `union` that has non-data members
- References

The default is LANG(OFFSETNONPOD).

OLDDIGRAPH | NOOLDDIGRAPH

This option controls whether old-style digraphs are allowed in your C++ source. It applies only when DIGRAPH is also set. When LANG(NOOLDDIGRAPH) is specified, z/OS XL C++ supports only the digraphs specified in the C++ standard. Set LANG(OLDDIGRAPH) if your code contains at least one of following digraphs:

- `%%` digraph, which results in `#` (pound sign)
- `%%%%` digraph, which results in `##` (double pound sign, used as the preprocessor macro concatenation operator)

Specify LANG(NOOLDDIGRAPH) for compatibility with ISO standard C++ and the extended C++ language level. The default is LANG(NOOLDDIGRAPH).

OLDFRIEND | NOOLDFRIEND

This option controls whether friend declarations that name classes without elaborated class names are treated as C++ errors. When LANG(OLDFRIEND) is in effect, you can declare a friend class without elaborating the name of the class with the keyword `class`. This is an extension to the C++ standard. For example, this statement declares the class `IFont` to be a friend class and is valid when LANG(OLDFRIEND) is in effect:

```
friend IFont;
```

This example declaration causes a warning unless you modify it or suppress the message with the SUPPRESS(CCN5070) option. Specify LANG(NOOLDFRIEND) for compliance with ISO standard C++. Specifying this option will cause an error condition and message to be generated for the example declaration.

```
friend class IFont;
```

The default for batch and TSO is LANG(OLDFRIEND).

OLDMATH | NOOLDMATH

This option controls which math function declarations are introduced by the `math.h` header file. For conformance with the C++ standard, the `math.h` header file declares several new functions that were not declared by `math.h` in previous releases. These new function declarations may cause an existing program to become invalid and, therefore, to fail to compile. This occurs because the new function declarations introduce the possibility of ambiguities in function overload resolution. The OLDMATH option specifies that these new function declarations are not to be introduced by the `math.h` header file, thereby significantly reducing the possibility of ambiguous overload resolution. The default is LANG(NOOLDMATH).

OLDSTR | NOOLDSTR

This option provides backward compatibility with previous versions of z/OS XL C++ and predecessor products, by controlling which string function declarations are introduced by the `string.h` and `wchar.h` header files. For conformance with the current C++ standard, `string.h` and the `wchar.h` header files declare several C++ string functions differently for C++ source files than they were declared in previous releases. These new function declarations may cause an existing C++ program to become invalid and therefore fail to compile. The LANG(OLDSTR) option specifies that the new C++ string function declarations are not to be introduced by the `string.h` and `wchar.h` header files, thereby causing only the C versions of these functions to be declared, as in previous releases. Note that when a C source file is compiled, these declarations remain unchanged from previous releases.

A number of the string function signatures that are defined in the 1989 C International Standard and the C Amendment are not `const`-safe.

Example: Consider the following standard C signature:

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

The behavior of this function is specified as follows:

- The `strchr` function locates the first occurrence of `c` (converted to a char) in the string pointed to by `s`. The terminating null character is considered to be part of the string.
- The `strchr` function returns a pointer to the located character, or a null pointer if the character does not occur in the string.

Since the parameter `s` is of type `const char *s`, the string being searched by the `strchr` function is potentially composed of characters whose type is `const char`. The `strchr` function returns a pointer to one of the constituent characters of the string, but this pointer is of type `char *` even though the character that it points to is potentially of type `const char`. For this reason, `strchr` can be used to implicitly (and unintentionally) defeat the `const`-qualification of the string referenced by the pointer `s`.

Example: To correct this problem, the C++ standard replaces the signature from the C standard with the following overloaded signatures:

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

Both of these overloaded functions have the same behavior as the original C version of `strchr`.

In a similar manner, the signatures of several other standard C library routines are replaced in the C++ standard. The affected routines are:

- `strchr`
- `strpbrk`

- strrchr
- strstr
- memchr
- wcschr
- wcspbrk
- wcsrchr
- wcsstr
- wmemchr

Example: Because of the changes mandated by the C++ standard, the following unsafe example will not compile in C++:

```
        #include <string.h>

        const char s[] = "foobar";

        int main(void) {
                char * c = strchr(s, 'b');
        }
```

To preserve backward compatibility with previous releases (and thus enable this code example), specify LANGLVL(OLDSTR).

OLDTEMPACC | NOOLDTEMPACC

This option controls whether access to a copy constructor to create a temporary object is always checked, even if creation of the temporary object is avoided. When LANG(NOOLDTEMPACC) is in effect, z/OS XL C++ suppresses the access checking. This is an extension to the C++ standard. When LANG(OLDTEMPACC) is in effect, you receive a warning if your code uses the extension, unless you disable the message. Disable the message by building with SUPPRESS(CCN5306) when the copy constructor is a private member, and SUPPRESS(CCN5307) when the copy constructor is a protected member. Specify LANG(NOOLDTEMPACC) for compliance with ISO standard C++.

Example: The throw statement in the following code causes an error because the copy constructor is a protected member of class C:

```
class C {
public:
   C(char *);
protected:
   C(const C&);
};

C foo() {return C("test");}  // returns a copy of a C object

void f()
{
// catch and throw both make implicit copies of the thrown object
   throw C("error");         // throws a copy of a C object
   const C& r = foo();       // uses the copy of a C object created by foo()
}
```

This example code contains three ill formed uses of the copy constructor C(const C&). The default is LANG(OLDTEMPACC).

OLDTMPLALIGN | NOOLDTMPLALIGN

This option specifies the alignment rules implemented by the compiler for nested templates. Previous versions of the compiler ignored alignment rules specified for nested templates. By default, LANG(EXTENDED) sets LANG(NOOLDTMPLALIGN) so the alignment rules are not ignored. The default for is LANG(NOOLDTMPLALIGN).

OLDTMPLSPEC | NOOLDTMPLSPEC

This option controls whether template specializations that do not conform to the C++ standard are allowed. When LANG(NOOLDTMPLSPEC) is in effect, z/OS XL C++ allows these old specializations. This is an extension to ISO standard C++. When LANGLVL(OLDTMPLSPEC) is set, you receive a warning if your code uses the extension, unless you suppress the message with SUPPRESS(CCN5080).

Example: You can explicitly specialize the template class `ribbon` for type `char` with the following lines:

```
template<classT> class ribbon { /*...*/};
class ribbon<char> { /*...*/};
```

Specify LANG(NOOLDTMPLSPEC) for compliance with standard C++. In this example, the template specialization must be modified to:

```
template<class T> class ribbon { /*...*/};
template<> class ribbon<char> { /*...*/};
```

The default is LANG(OLDTMPLSPEC).

TRAILENUM | NOTRAILENUM

This option controls whether trailing commas are allowed in `enum` declarations. When LANG(TRAILENUM) is in effect, z/OS XL C++ allows one or more trailing commas at the end of the enumerator list. This is an extension to the C++ standard. The following `enum` declaration uses this extension:

```
enum grain { wheat, barley, rye,, };
```

Specify LANG(NOTRAILENUM) for compliance with the ISO C and C++ standards. The default is LANG(TRAILENUM).

TYPEDEFCLASS | NOTYPEDEFCLASS

This option provides backwards compatibility with previous versions of z/OS XL C++ and predecessor products. The current C++ standard does not allow a typedef name to be specified where a class name is expected. This option relaxes that restriction. Specify LANG(TYPEDEFCLASS) to allow the use of typedef names in base specifiers and constructor initializer lists. When LANG(NOTYPEDEFCLASS) is in effect, a typedef name cannot be specified where a class name is expected. The default is LANG(TYPEDEFCLASS).

UCS | NOUCS

This option controls whether Unicode characters are allowed in identifiers, string literals and character literals in C++ sources. The Unicode character set is supported by the C++ standard. This character set contains the full set of letters, digits and other characters used by a wide range of languages, including all North American and Western European languages. Unicode characters can be 16 or 32 bits. The ASCII one-byte characters are a subset of the Unicode character set. When LANG(UCS) is in effect, you can insert Unicode characters in your source files either directly or using a notation that is similar to escape sequences. Because many Unicode characters cannot be displayed on the screen or entered from the keyboard, the latter approach is usually preferred. Notation forms for Unicode characters are `\uhhhh` for 16-bit characters, or `\Uhhhhhhhh` for 32-bit characters, where `h` represents a hexadecimal digit. Short identifiers of characters are specified by ISO/IEC 10646. The default is LANG(NOUCS).

Note: The C99 specification has added the use of universal character names within identifiers and the set of restrictions differs from C++. LANG(UCS) supports the union of valid universal character name ranges from the current C++ specification and the new C99 specification.

ZEROEXTARRAY | NOZEROEXTARRAY

This option controls whether zero-extent arrays are allowed as the last non-static data member in a class definition. When LANG(ZEROEXTARRAY) is in effect, z/OS XL C++ allows arrays with zero elements. This is an extension to the C++ standard.

Example: These example declarations define dimensionless arrays `a` and `b`:

```
struct S1 { char a[0]; };
struct S2 { char b[]; };
```

Specify LANG(NOZEROEXTARRAY) for compliance with the ISO C++ standard. When LANG(ZEROEXTARRAY) is set, you receive informational messages about zero-extent arrays in your code, unless you suppress the message with SUPPRESS(CCN6607). The default is LANG(ZEROEXTARRAY).

Usage

The LANGLVL option defines a macro that specifies a language level. You must then include this macro in your code to force conditional compilation; for example, with the use of `#ifdef` directives. You can write portable code if you correctly code the different parts of your program according to the language level. You use the macro in preprocessor directives in header files.

Note: The following list shows ISO C99 language constructs unavailable with LANGLVL(EXTENDED) or LANGLVL(EXTC89):

- `inline` keyword
- `restrict` keyword
- C++ style comments

Unsuffixed integer literals are handled differently under ISO C99 than they are for LANGLVL(EXTENDED) or LANGLVL(EXTC89). Unsuffixed integer literals with values greater than INT_MAX, have a `long long` type under ISO C99 and an `unsigned int` type under LANGVL(EXTENDED) or LANGLVL(EXTC89).

You can control individual language features in the z/OS V1R2 C++ compiler by using the LANGLVL and KEYWORD suboptions listed in Table 24 on page 151. In order to conform to the ISO C++ standard, you may need to make a number of changes to your existing source code. These suboptions can help by breaking up the changes into smaller steps.

Note: The group options override the individual suboptions so if you want to specify a suboption it should be after a group option. For example, if you specify LANG(ANSIFOR,COMPAT92) you will get LANG(NOANSIFOR) because the LANG(COMPAT92) specifies NOANSIFOR. Thus you should specify LANG(COMPAT92,ANSIFOR) to get ANSIFOR.

Predefined macros

For the list of predefined macros related to language levels, see "Macros related to language levels" in *z/OS XL C/C++ Language Reference*.

LIBANSI | NOLIBANSI

Category

Optimization and tuning

Pragma equivalent

#pragma options (libansi) (C only), **#pragma options (nolibansi)** (C only)

Purpose

Indicates whether or not functions with the name of an ANSI C library function are in fact ANSI C library functions and behave as described in the ANSI standard.

When LIBANSI is in effect, the optimizer can generate better code because it will know about the behavior of a given function, such as whether or not it has any side effects.

Syntax

```
       ┌─NOLIB─┐
►►─────┴─LIB───┴──────────────────────────────────────────────────────►◄
```

Defaults

NOLIBANSI

IPA effects

The LIBANSI option has the same effect on the IPA compile step as it does for normal compilation.

The LIBANSI option will be in effect for the IPA Link step unless the NOLIBANSI option is specified.

The LIBANSI option that you specify on the IPA Link step will override the LIBANSI option that you specify on the IPA Compile step. The LIBANSI option that you specify on the IPA Link step is shown in the IPA Link listing Compile Option Map for reference.

Predefined macros

__LIBANSI__ is defined to 1 when LIBANSI is specified in C++; otherwise, it is not defined.

LIST | NOLIST

Category

Listings, messages and compiler information

Pragma equivalent

None.

Purpose

Produces a compiler listing file that includes a pseudo assembly listing.

Syntax

```
       ┌─NOLIS─┐
►►─────┴─LIS───┴──┬──────────────────────────────────────────────┬────────►◄
                  └─(──┬─Sequential filename─────────────────┬─)─┘
                       ├─Partitioned data set────────────────┤
                       ├─Partitioned data set (member)───────┤
                       ├─z/OS UNIX System Services filename──┤
                       └─z/OS UNIX System Services directory─┘
```

Defaults

NOLIST

In the z/OS UNIX System Services environment, this option is turned on by specifying -V when using the **c89**, **cc** or **c++** commands. -V produces all reports for the compiler, and binder, or prelinker, and directs them to `stdout`. To produce only the listing (and no other reports), and write the listing to a user-specified file, use the following command:

```
-Wc,"LIST(filename)"
```

Parameters

Sequential filename
: Specifies the sequential data set file name for the compiler listing.

Partitioned data set
: Specifies the partitioned data set for the compiler listing.

Partitioned data set (member)
: Specifies the partitioned data set (member) for the compiler listing.

z/OS UNIX System Services filename
: Specifies the z/OS UNIX System Services file name for the compiler listing.

z/OS UNIX System Services directory
: Specifies the z/OS UNIX System Services directory for the compiler listing.

Usage

When the LIST compiler option is in effect, the compiler is instructed to generate a listing of the machine instructions in the object module (in a format similar to assembler language instructions) in the compiler listing.

LIST(*filename*) places the compiler listing in the specified file. If you do not specify a file name for the LIST option, the compiler uses the SYSCPRT ddname if you allocated one. Otherwise, the compiler generates a file name as follows:

- If you are compiling a data set, the compiler uses the source file name to form the name of the listing data set. The high-level qualifier is replaced with the userid under which the compiler is running, and `.LIST` is appended as the low-level qualifier.
- If you are compiling a z/OS UNIX file, the compiler stores the listing in a file that has the name of the source file with a `.lst` extension.

The NOLIST option optionally takes a file name suboption. This file name then becomes the default. If you subsequently use the LIST option without a file name suboption, the compiler uses the file name that you specified in the earlier NOLIST. For example, the following specifications have the same effect:

```
c89 -Wc,"NOLIST(hello.list)" LIST
c89 -Wc,"LIST(hello.list)"
```

If you specify data set names in a C or C++ program, with the SOURCE, LIST or INLRPT options, all the listing sections are combined into the last data set name specified.

Notes:

1. Usage of information such as registers, pointers, data areas, and control blocks that are shown in the object listing are not programming interface information.
2. If you use the following form of the command in a JES3 batch environment where xxx is an unallocated data set, you may get undefined results.

```
LIST(xxx)
```

3. Statement line numbers exceeding 99999 will wrap back to 00000 for the generated assembly listing for the C/C++ source file. This may occur when the compiler LIST option is used.

IPA effects

If you specify the LIST option on the IPA Compile step, the compiler saves information about the source file and line numbers in the IPA object file. This information is available during the IPA Link step for use by the LIST or GONUMBER options.

If you do not specify the GONUMBER option on the IPA Compile step, the object file produced contains the line number information for source files that contain function begin, function end, function call, and function return statements. This is the minimum line number information that the IPA Compile step produces. You can then use the TEST option on the IPA Link step to generate corresponding test hooks.

Refer to “Interactions between compiler options and IPA suboptions” on page 43 and “GONUMBER | NOGONUMBER” on page 123 for more information.

If you specify the LIST option, the IPA Link listing contains a Pseudo Assembly section for each partition that contains executable code. Data-only partitions do not generate a Pseudo Assembly listing section.

The source file and line number shown for each object code statement depend on the amount of detail the IPA Compile step saves in the IPA object file, as follows:

- If you specified the GONUMBER, LIST, IPA(GONUMBER), or IPA(LIST) option for the IPA Compile step, the IPA Link step accurately shows the source file and line number information.
- If you did not specify any of these options on the IPA Compile step, the source file and line number information in the IPA Link listing or GONUMBER tables consists only of the following:
  - function entry, function exit, function call, and function call return source lines. This is the minimum line number information that the IPA Compile step produces.
  - All other object code statements have the file and line number of the function entry, function exit, function call, and function call return that was last encountered. This is similar to the situation of encountering source statements within a macro.

Predefined macros

None.

Related information

Refer to “Interactions between compiler options and IPA suboptions” on page 43 and “GONUMBER | NOGONUMBER” on page 123 for more information.

LOCALE | NOLOCALE

Category

Object code control

Pragma equivalent

None.

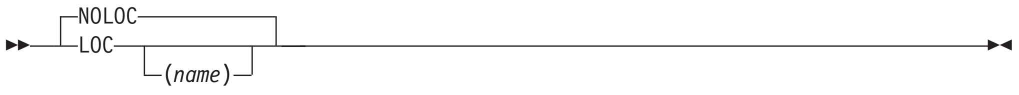
Purpose

Specifies the locale to be used by the compiler as the current locale throughout the compilation unit.

When the LOCALE compiler option is in effect, you can specify the locale.

When the NOLOCALE compiler option is in effect, the compiler uses the default code page, which is IBM-1047.

Syntax



Defaults

NOLOCALE

For the z/OS UNIX System Services utilities, the default for a regular compile is LOCALE(POSIX). The utilities pick up the locale value of the environment using `setlocale(LC_ALL,NULL))`. Because the compiler runs with the POSIX(OFF) option, categories that are set to C are changed to POSIX.

Parameters

name
: Indicates the name of the locale to be used by the compiler at compile time. If you omit *name*, the compiler uses the current default locale in the environment. If *name* does not represent a valid locale name, a warning message is emitted and NOLOCALE is used.

Usage

You can specify LOCALE on the command line or in the PARMS list in the JCL.

If you specify the LOCALE option, the locale name and the associated code set appear in the header of the listing. A locale name is also generated in the object module.

The LC_TIME category of the current locale controls the format of the time and the date in the compiler-generated listing file. The identifiers that appear in the tables in the listing file are sorted as specified by the LC_COLLATE category of the locale specified in the option.

Note: The formats of the predefined macros __DATE__, __TIME__, and __TIMESTAMP__ are not locale-sensitive.

IPA effects

The LOCALE option controls processing only for the IPA step for which you specify it.

During the IPA Compile step, the compiler converts source code using the code page that is associated with the locale specified by the LOCALE compile-time option. As with non-IPA compilations, the conversion applies to identifiers, literals, and listings. The locale that you specify on the IPA Compile step is recorded in the IPA object file.

You should use the same code page for IPA Compile step processing for all of your program source files. This code page should match the code page of the run-time environment. Otherwise, your application may not run correctly.

The locale that you specify on the IPA Compile step does not determine the locale that the IPA Link step uses. The LOCALE option that you specify on the IPA Link step is used for the following:
- The encoding of the message text and the listing text.
- Date and time formatting in the Source File Map section of the listing and in the text in the object comment string that records the date and time of IPA Link step processing.
- Sorting of identifiers in listings. The IPA Link step uses the sort order associated with the locale for the lists of symbols in the Inline Report (Summary), Global Symbols Map, and Partition Map listing sections.

If the code page you used for a compilation unit for the IPA Compile step does not match the code page you used for the IPA Link step, the IPA Link step issues an informational message.

If you specify the IPA(MAP) option, the IPA Link step displays information about the LOCALE option, as follows:
- The Prolog section of the listing displays the LOCALE or NOLOCALE option. If you specified the LOCALE option, the Prolog displays the locale and code set that are in effect.
- The Compiler Options Map listing section displays the LOCALE option active on the IPA Compile step for each IPA object. If you specified conflicting code sets between the IPA Compile and IPA Link steps, the listing includes a warning message after each Compiler Options Map entry that displays a conflict.
- The Partition Map listing section shows the current LOCALE option.

Predefined macros
- __CODESET__ is defined to the name of the compile-time code set. The compiler uses the run-time function `nl_langinfo(CODESET)` to determine the name of the compile-time code set. If you do not use the LOCALE compile option, the macro is undefined.
- __LOCALE__ is defined to the name of the compile-time locale. If you specified LOCALE(string literal), the compiler uses the run-time function `setlocale(LC_ALL,"string literal")` to determine the name of the compile-time locale. If you do not use the LOCALE compile option, the macro is undefined.

Related information
For more information on locales, refer to *z/OS XL C/C++ Programming Guide*.

LONGNAME | NOLONGNAME

Category
Object code control

Pragma equivalent
#pragma longname, **#pragma nolongname**

Purpose
Provides support for external names of mixed case and up to 1024 characters long.

When the LONGNAME compiler option is in effect, the compiler generates untruncated and mixed case external names in the object module produced by the compiler for functions with non-C++ linkage.

When the NOLONGNAME compiler option is in effect:

- The compiler generates truncated and uppercase names in the object module.
- Only those functions that do not have C++ linkage are given truncated and uppercase names.
- The XL C compiler truncates all the external names to 8 characters whereas the XL C++ compiler only truncates the external functions to 8 characters.

Syntax

For C:

```
      ┌─NOLO─┐
►►────┴─LO───┴──────────────────────────────────────────────────────►◄
```

For C++:

```
      ┌─LO───┐
►►────┴─NOLO─┴──────────────────────────────────────────────────────►◄
```

Defaults

For C, the default option is NOLONGNAME. For C++, the default option is LONGNAME.

For the z/OS UNIX System Services utilities, the default for a regular compile is LONGNAME.

Usage

Functions with C++ linkage are always untruncated and mixed-case external names.

The system binder recognizes the format of long external names in object modules, but the system linkage editor does not.

For z/OS XL C, if you specify the ALIAS option with LONGNAME, the compiler generates a NAME control statement, but no ALIAS control statements.

If you use **#pragma map** to associate an external name with an identifier, the compiler generates the external name in the object module. That is, **#pragma map** has the same behavior for the LONGNAME and NOLONGNAME compiler options. Also, **#pragma csect** has the same behavior for the LONGNAME and NOLONGNAME compiler options.

A comment that indicates the setting of the LONGNAME option will be generated in your object module to aid you in diagnosing your program.

Note: For C only, you must specify either the LONGNAME compiler option or the **#pragma longname** preprocessor directive whenever the HOT option is specified. Otherwise, the compiler issues an unrecoverable error diagnostic message.

IPA effects

For C only, you must specify either the LONGNAME compiler option or the **#pragma longname** preprocessor directive for the IPA Compile step. Otherwise, the compiler issues an unrecoverable error diagnostic message.

The IPA Link step ignores this option if you specify it, and uses the LONGNAME option for all partitions it generates.

Predefined macros

For C, __LONGNAME__ is predefined to 1 when the LONGNAME compiler option is in effect.

For C++, __LONGNAME__ is predefined to 1 when the LONGNAME or NOLONGNAME compiler option is in effect.

For C++, __IBMCPP_LONGNAME__ is defined as 1 if the LONGNAME compiler option is in effect, and it is undefined if the NOLONGNAME compiler option is on.

Note: __IBMCPP_LONGNAME__ is only available starting with z/OS V1R10. Please check your XL C++ compiler level by using __IBMCPP__ when using __IBMCPP_LONGNAME__. For more information, see the *z/OS XL C/C++ Language Reference*.

LP64 | ILP32

Category

Object code control

Pragma equivalent

None.

Purpose

Selects either AMODE 64 or AMODE 31 mode.

When the LP64 compiler option is in effect, the compiler generates AMODE 64 code utilizing the z/Architecture 64-bit instructions.

When the ILP32 compiler option is in effect, the compiler generates AMODE 31 code. This is the default and is the same mode as in previous releases of the compiler.

Note: AMODE is the addressing mode of the program code generated by the compiler. In AMODE 64 and AMODE 31, 64 and 31 refer to the range of addresses that can be accessed (in other words 64-bits and 31-bits are used to form the address respectively). When there is no ambiguity, we will refer to these as 64-bit mode and 31-bit mode. Refer to the information that follows for further information on the data model.

Syntax

```
   ┌─ILP32─┐
►►─┴─LP64──┴──────────────────────────────────────────────────────►◄
```

Defaults

ILP32

Usage

LP64 and ILP32 are mutually exclusive. If they are specified multiple times, the compiler will take the last one.

LP64 and ILP32 refer to the data model used by the language. "I" is an abbreviation that represents int type, "L" represents long type, and "P" represents the pointer type. 64 and 32 refer to the size of the data types. When the ILP32 option is used, `int`, `long` and pointers are 32-bit in size. When LP64 is used, `long` and pointer are 64-bit in size; `int` remains 32-bit. The addressing mode used by LP64 is AMODE 64, and by ILP32 is AMODE 31. In the latter case, only 31 bits within the pointer are taken to form the address. For the sake of conciseness, the terms 31-bit mode and ILP32, will be used interchangeably in this document when there is no ambiguity. The same applies to 64-bit mode and LP64.

The LP64 option requires the XPLINK and GOFF compiler options. It also requires architecture level 5 or above (ARCH(5) or higher). ARCH(5), XPLINK, and GOFF are the default settings for LP64 if you don't explicitly override them. If you explicitly specify NOXPLINK, or NOGOFF, or specify an architecture level lower than 5, the compiler will issue a warning message, ignore NOXPLINK or NOGOFF, and raise the architecture level to 5.

Notes:

1. The maximum size of a GOFF object is 1 gigabyte.
2. ARCH(5) specifies the 2064 hardware models.

The prelinker cannot be used with 64-bit object modules.

Note: The Language Environment element does not support mixing 64-bit and 31-bit object files in the same application. If one compilation unit is compiled with LP64, all compilation units within the program must be compiled with LP64. The binder will issue a message if it encounters mixed addressing modes during external name resolution.

In 31-bit mode, the size of `long` and pointers is 4 bytes and the size of `wchar_t` is 2 bytes. Under LP64, the size of `long` and pointer is 8 bytes and the size of `wchar_t` is 4 bytes. The size of other intrinsic datatypes remain the same between 31-bit mode and LP64. Under LP64, the type definition for `size_t` changes to `long`, and the type definition for `ptrdiff_t` changes to `unsigned long`. The following tables give the size of the intrinsic types:

Table 25. Size of intrinsic types in 64–bit mode

Type	**Size (in bits)**
char, unsigned char, signed char	8
short, short int, unsigned short, unsigned short int, signed short, signed short int	16
int, unsigned int, signed int	32
long, long int, unsigned long, unsigned long int, signed long, signed long int	64
long long, long long int, unsigned long long, unsigned long long int, signed long long, signed long long int	64
pointer	64

Table 26. Size of intrinsic types in 31–bit mode

Type	Size (in bits)
char, unsigned char, signed char	8
short, short int, unsigned short, unsigned short int, signed short, signed short int	16
int, unsigned int, signed int	32
long, long int, unsigned long, unsigned long int, signed long, signed long int	32
long long, long long int, unsigned long long, unsigned long long int, signed long long, signed long long int	64
pointer	32

The `__ptr32` pointer qualifier is intended to make the process of porting applications from ILP32 to LP64 easier. Use this qualifier in structure members to minimize the changes in the overall size of structures. Note that these pointers cannot refer to objects above the 31-bit address line (also known as "the bar"). In general, the program has no control over the address of a variable; the address is assigned by the implementation. It is up to the programmer to make sure that the use of `__ptr32` is appropriate within the context of the program's logic. For more information on the `__ptr32` pointer qualifer, refer to *z/OS XL C/C++ Language Reference*.

Notes:

1. The `long` and `wchar_t` data types also change in size.
2. LP64 only supports OBJECTMODEL(IBM).

IPA effects

The IPA Compile step generates information for the IPA Link step for ILP32. The LP64 option affects the regular object module if you request one by specifying the IPA(OBJECT) option, in which case, the object module generated will be in 64-bit.

The IPA Link step accepts the LP64 option, but ignores it. The DLL side deck generated by the binder has been enhanced. The side deck contains attribute flags to mark symbols exported from 64-bit DLLs; the flags are CODE64 and DATA64 for code and data respectively. IPA recognizes these flags.

The IPA Link step will check that all objects have a consistent data model, either ILP32 or LP64. It checks both IPA object modules and non-IPA object modules. If the IPA Link step finds a mixture of addressing modes among the object files, the compiler issues a diagnostic message and ends the compilation.

Predefined macros

_LP64 is defined to 1 when the LP64 compiler option is in effect; otherwise, the macro _ILP32 is predefined to 1.

LSEARCH | NOLSEARCH

Category

Compiler input

Pragma equivalent

None.

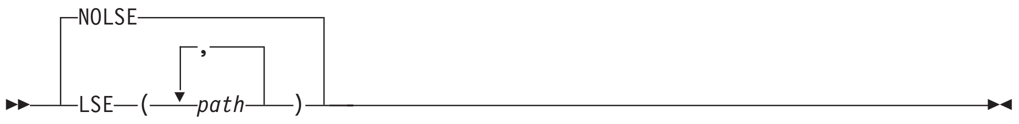
Purpose

Specifies the directories or data sets to be searched for user include files.

When the LSEARCH compiler option is in effect, the preprocessor looks for the user include files in the specified directories or data sets.

When the NOLSEARCH compiler option is in effect, the preprocessor only searches those data sets that are specified in the USERLIB DD statement. A NOLSEARCH option cancels all previous LSEARCH specifications, and the compiler uses any LSEARCH options that follow it.

Syntax



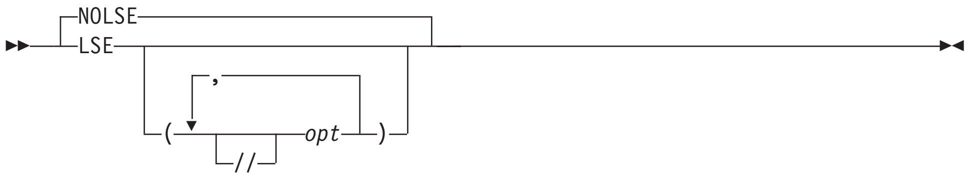
Defaults

NOLSEARCH

Parameters

path
: Specifies any of the following:

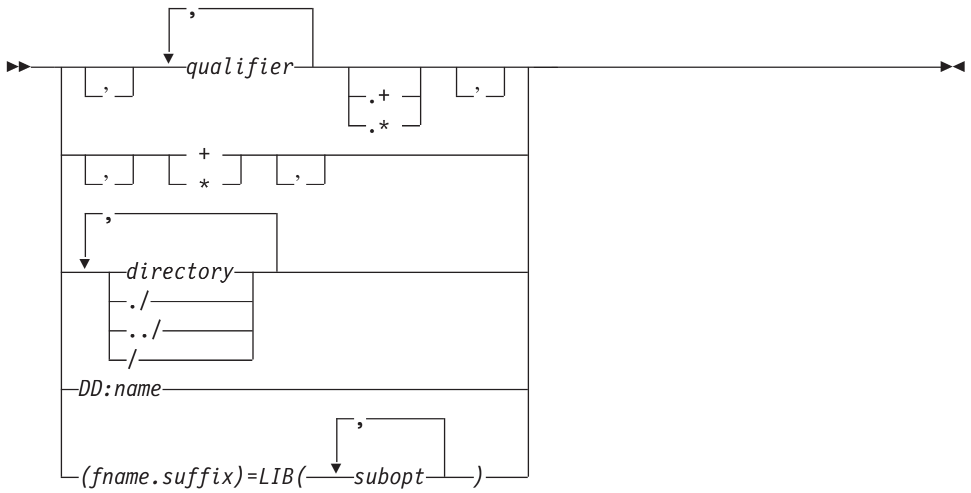
- The name of a partitioned or sequential data set that contains user include files.
- A z/OS UNIX System Services file system path that contains user include files.
- A search path that is more complex:



You must use the double slashes (//) to specify data set library searches when you specify the OE compiler option. (You may use them regardless of the OE option).

The USERLIB ddname is considered the last suboption for LSEARCH, so that specifying LSEARCH `(X)` is equivalent to specifying LSEARCH `(X,DD:USERLIB)`.

Parts of the `#include` *`filename`* are appended to each LSEARCH *opt* to search for the include file. *opt* has the format:



In this syntax diagram, *opt* specifies one of the following:

- The name of a partitioned or sequential data set that contains user include files
- A z/OS UNIX file system path name that should be searched for the include file. You can also use ./ to specify the current directory and ../ to specify the parent directory for your z/OS UNIX file.
- A DD statement for a sequential data set or a partitioned data set. When you specify a ddname in the search and the include file has a member name, the member name of the include file is used as the name for the `DD:` *name* search suboption, for example:

  ```
  LSEARCH(DD:NEWLIB)
  #include "a.b(c)"
  ```

  The resulting file name is `DD:NEWLIB(C)`.
- A specification of the form `(`*`fname.suffix`*`) = (`*`subopt,subopt,...`*`)` where:
  - *fname* is the name of the include file, or *
  - *suffix* is the suffix of the include file, or *
  - *subopt* indicates a subpath to be used in the search for the include files that match the pattern of *fname.suffix*. There should be at least one *subopt*. The possible values are:
    - LIB(*[pds,...]*) where each *pds* is a partitioned data set name. They are searched in the same order as they are specified.

      There is no effect on the search path if no *pds* is specified, but a warning is issued.
    - LIBs are cumulative; for example, LIB(A),LIB(B) is equivalent to LIB(A, B).
    - NOLIB specifies that all LIB(...) previously specified for this pattern should be ignored at this point.

When the `#include` *`filename`* matches the pattern of *fname.suffix*, the search continues according to the subopts in the order specified. An asterisk (*) in *fname* or *suffix* matches anything. If the compiler does not find the file, it attempts other searches according to the remaining options in LSEARCH.

Usage

When you specify more than one LSEARCH option, the compiler uses all the directories or data sets in these LSEARCH options to find the user include files.

The `#include "`*`filename`*`"` format of the `#include` C/C++ preprocessor directive indicates user include files. See “Using include files” on page 360 for a description of the `#include` preprocessor directive.

Note: If the *filename* in the `#include` directive is in absolute form, the compiler does not perform a search. See “Determining whether the file name is in absolute form” on page 365 for more details on absolute `#include` *`filename`*.

For further information on search sequences, see “Search sequences for include files” on page 369.

When specifying z/OS UNIX library searches, do not put double slashes at the beginning of the LSEARCH *opt* . Use *pathnames* separated by slashes (/) in the LSEARCH *opt* for a z/OS UNIX library. When the LSEARCH *opt* does not start with double slashes, any single slash in the name indicates a z/OS UNIX library. If you do not have path separators (/), then setting the OE compile option on indicates that this is a z/OS UNIX library; otherwise the library is interpreted as a data set. See “Using SEARCH and LSEARCH” on page 367 for additional information on z/OS UNIX files.

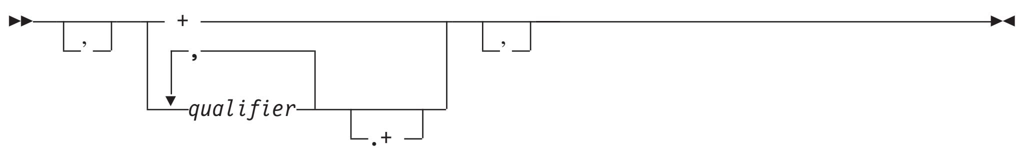
Example: The *opt* specified for LSEARCH is combined with the *filename* in `#include` to form the include file name:

```
LSEARCH(/u/mike/myfiles)
#include "new/headers.h"
```

The resulting z/OS UNIX file name is `/u/mike/myfiles/new/headers.h`.

Use an asterisk (*) or a plus sign (+) in the LSEARCH *opt* to specify whether the library is a sequential or partitioned data set.

When you want to specify a set of PDSs as the search path, you add a period followed by a plus sign (.+) at the end of the last qualifier in the *opt*. If you do not have any qualifier, specify a single plus sign (+) as the *opt*. The *opt* has the following syntax for specifying partitioned data set:



where *qualifier* is a data set qualifier.

Start and end the *opt* with single quotation marks (') to indicate that this is an absolute data set specification. Single quotation marks around a single plus sign (+) indicate that the *filename* that is specified in `#include` is an absolute partitioned data set.

When you do not specify a member name with the `#include` directive, for example, `#include "PR1.MIKE.H"`, the PDS name for the search is formed by replacing the plus sign with the following parts of the *filename* of the `#include` directive:
- For the PDS file name:
  1. All the *path*s and slashes (slashes are replaced by periods)
  2. All the periods and *qualifier*s after the left-most *qualifier*
- For the PDS member name, the left-most *qualifier* is used as the member name

See the first example in Table 27 on page 176.

However, if you specified a member name in the *filename* of the `#include` directive, for example, `#include "PR1.MIKE.H(M1)"`, the PDS name for the search is formed by replacing the plus sign with the qualified name of the PDS. See the second example in Table 27 on page 176.

See "Forming data set names with LSEARCH | SEARCH options" on page 362 for more information on forming PDS names.

Note: To specify a single PDS as the *opt*, do not specify a trailing asterisk (*) or plus sign (+). The library is then treated as a PDS but the PDS name is formed by just using the leftmost *qualifier* of the `#include` *filename* as the member name. For example:

```
LSEARCH(AAAA.BBBB)
#include "sys/ff.gg.hh"

Resulting PDS name is
userid.AAAA.BBBB(FF)
```

Also see the third example in Table 27 on page 176.

Predefined macros

None.

Examples

To search for PDS or PDSE files when you have coded your include files as follows:

```
#include "sub/fred.h"
#include "fred.inl"
```

You specified LSEARCH as follows:

```
LSEARCH(USER.+,'USERID.GENERAL.+')
```

The compiler uses the following search sequence to look for your include files:

1. First, the compiler looks for `sub/fred.h` in this data set:
   `USERID.USER.SUB.H(FRED)`
2. If that PDS member does not exist, the compiler looks in the data set:
   `USERID.GENERAL.SUB.H(FRED)`
3. If that PDS member does not exist, the compiler looks in DD:USERLIB, and then checks the system header files.
4. Next, the compiler looks for `fred.inl` in the data set:
   `USERID.USER.INL(FRED)`
5. If that PDS member does not exist, the compiler will look in the data set:
   `USERID.GENERAL.INL(FRED)`
6. If that PDS member does not exist, the compiler looks in DD:USERLIB, and then checks the system header files.

The compiler forms the search path for z/OS UNIX files by appending the path and name of the `#include` file to the path that you specified in the LSEARCH option.

Example 1

You code `#include "sub/fred.h"` and specify:

```
LSEARCH(/u/mike)
```

The compiler looks for the include file `/u/mike/sub/fred.h` .

Example 2

You specify your header file as `#include "fred.h"` , and your LSEARCH option as:

```
LSEARCH(/u/mike, ./sub)
```

The compiler uses the following search sequence to look for your include files:

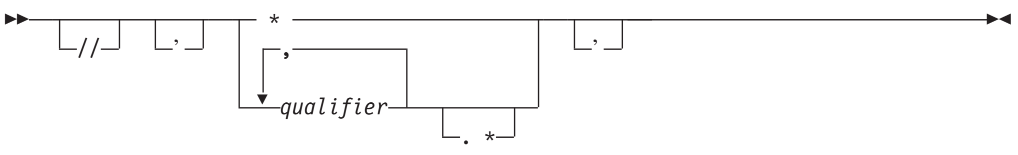
1. The compiler looks for `fred.h` in:
   `/u/mike/fred.h`
2. If that z/OS UNIX file does not exist, the compiler looks in:
   `./sub/fred.h`
3. If that z/OS UNIX file does not exist, the compiler looks in the libraries specified on the USERLIB DD statement.
4. If USERLIB DD is not allocated, the compiler follows the search order for system include files.

The following example shows you how to specify a PDS search path:

Table 27. Partitioned data set examples

include Directive	LSEARCH option	Result
#include ″PR1.MIKE.H″	LSEARCH(’CC.+’)	’CC.MIKE.H(PR1)’
#include ″PR.KE.H(M1)″	LSEARCH(’CC.+’)	’CC.PR.KE.H(M1)’
#include ″A.B″	LSEARCH(CC)	*userid*.CC(A)
#include ″A.B.D″	LSEARCH(CC.+)	*userid*.CC.B.D(A)
#include ″a/b/dd.h″	LSEARCH(’CC.+’)	’CC.A.B.H(DD)’
#include ″a/dd.ee.h″	LSEARCH(’CC.+’)	’CC.A.EE.H(DD)’
#include ″a/b/dd.h″	LSEARCH(’+’)	’A.B.H(DD)’
#include ″a/b/dd.h″	LSEARCH(+)	*userid*.A.B.H(DD)
#include ″A.B(C)″	LSEARCH(’D.+’)	’D.A.B(C)’

When you want to specify a set of sequential data sets as the search path, you add a period followed by an asterisk (.*) at the end of the last qualifier in the *opt*. If you do not have any qualifiers, specify one asterisk (*) as the *opt*. The *opt* has the following syntax for specifying a sequential data set:



where *qualifier* is a data set qualifier.

Start and end the *opt* with single quotation marks (') to indicate that this is an absolute data set specification. Single quotation marks (') around a single asterisk (*) means that the file name that is specified in `#include` is an absolute sequential data set.

The asterisk is replaced by all of the qualifiers and periods in the `#include` *filename* to form the complete name for the search (as shown in the following table).

The following example shows you how to specify a search path for a sequential data set:

Table 28. Sequential data set examples

include Directive	LSEARCH option	Result
#include ″A.B″	LSEARCH(CC.*)	*userid*.CC.A.B
#include ″a/b/dd.h″	LSEARCH(’CC.*’)	’CC.DD.H’
#include ″a/b/dd.h″	LSEARCH(’*’)	’DD.H’
#include ″a/b/dd.h″	LSEARCH(*)	*userid*.DD.H

Note: If the trailing asterisk is not used in the LSEARCH *opt*, then the specified library is a PDS:

```
#include "A.B"
LSEARCH('CC')
```

Result is `'CC(A)'` which is a PDS.

MAKEDEP

Category

Compiler output

Pragma equivalent

None.

Purpose

Source files read during compilation are displayed in an output dependency file, which is consumable by the make utility

Note: This option is only supported using **-q** syntax. **-M** is the equivalent of specifying **-qmakedep**. For detail information about **-M**, see Chapter 24, “xlc — Compiler invocation using a customizable configuration file,” on page 577.

Syntax

```
       ┌─NOMAKEDEP─┐
►►─────┴─MAKEDEP───┴──────────────────────────────────────────────►◄
```

Defaults

NOMAKEDEP

Usage

For each C/C++ source file specified on the command line, an output file is generated with the same name as the object file and the suffix replaced with the suffix for make dependency files. The default suffix for make dependency files is .u. It can be customized using the usuffix attribute in the xlc utility configuration file.

The option only applies to C/C++ sources in z/OS UNIX files, because MVS data sets do not have a time stamp required for make utility processing.

If the **-o** option is used to rename the object file, the output file uses the name you specified on the **-o** option.

The description file contains a line for the input file and specifies a separate rule for each of the main source file's dependencies in no particular order. It has the general form:

```
file_name.o: include_file_name
file_name.o: file_name.suffix
```

Include files are listed according to the search order rules for the `#include` preprocessor directive. If the include file is not found, it is not added to the .u file. Files with no include statements produce output files containing one line that lists only the input file name.

IPA effects

None.

Predefined macros

None.

Examples

To compile `foo_src.c` and create an output file named `foo_src.u`, enter:

```
xlc -c -qmakedep foo_src.c
```

To compile `foo_src.c` and create an output file named `my.d`, enter:

```
xlc -c -qmakedep foo_src.c -MF my.d
```

To compile `foo_src.c` and create an object file named `foo_obj.o` and an output file named `foo_obj.u`, enter:

```
xlc -c -qmakedep foo_src.c -o foo_obj.o
```

MARGINS | NOMARGINS

Category

Compiler input

Pragma equivalent

#pragma margins, **#pragma nomargins**

Purpose

Specifies, inclusively, the range of source column numbers that will be compiled.

When the MARGINS option is in effect, you can specify the columns in the input record that are to be scanned for input to the compiler. The compiler ignores text in the source input that does not fall within the range that is specified in the MARGINS option.

When the NOMARGINS options is in effect, the entire input source record will be scanned for input to the compiler.

Syntax

For C++ and C (variable record format):

```
        ┌─NOMAR──────────┐
►►──────┴─MAR──┬───────┬─┴──────────────────────────────────────────────►◄
               └─(m,n)─┘
```

For C (fixed record format):

```
        ┌─MAR──(m,n)─┐
►►──────┴─NOMAR──────┴──────────────────────────────────────────────────►◄
```

Defaults

- For C++ and C (variable record format), the default option is NOMARGINS.
- For C (fixed record format), the default option is MARGINS(`1,72`).
- For the z/OS UNIX System Services utilities, the default for a regular compile is NOMARGINS.

Parameters

m Specifies the first column of the source input that contains valid z/OS XL C/C++ code. The value of *m* must be greater than `0` and less than `32761`.

n Specifies the last column of the source input that contains valid z/OS XL C/C++ code. The value of *n* must be greater than *m* and less than `32761`. An asterisk (*) can be assigned to *n* to indicate the last column of the input record. If you specify MARGINS (`9,*`), the compiler scans from column 9 to the end of the record for input source statements.

Usage

You can use the MARGINS and SEQUENCE compiler options together. The MARGINS option is applied first to determine which columns are to be scanned. The SEQUENCE option is then applied to determine which of these columns are not to be scanned. If the SEQUENCE settings do not fall within the MARGINS settings, the SEQUENCE option has no effect.

When a source (or include) file is opened, it initially gets the margins and sequence specified on the command line (or the defaults if none was specified). You can reset these settings by using **#pragma margins** or **#pragma sequence** at any point in the file. When an `#include` file returns, the previous file keeps the settings it had when it encountered the `#include` directive.

If the MARGINS option is specified along with the SOURCE option in a C or C++ program, only the range specified on the MARGINS option is shown in the compiler source listing.

Notes:

1. The MARGINS option does not reformat listings.
2. If your program uses the `#include` preprocessor directive to include z/OS XL C library header files *and* you want to use the MARGINS option, you must ensure that the specifications on the MARGINS option does not exclude columns `20` through `50`. That is, the value of *m* must be less than `20`, and the value of *n* must be greater than `50`. If your program does not include any z/OS XL C library header files, you can specify any setting you want on the MARGINS option when the setting is consistent with your own include files.

Predefined macros
None.

Related information
For more information on related compiler options, see
- “SEQUENCE | NOSEQUENCE” on page 221
- “SOURCE | NOSOURCE” on page 226

MAXMEM | NOMAXMEM

Category
Optimization and tuning

Pragma equivalent
#pragma options (maxmem) (C only), **#pragma options (nomaxmem)** (C only)

Purpose
Limits the amount of memory used for local tables, and that the compiler allocates while performing specific, memory-intensive optimizations, to the specified number of kilobytes.

Syntax

```
     ┌MAXM──(──size──)┐
►►───┴NOMAXM──────────┴──────────────────────────────────────────────────►◄
```

Defaults
MAXMEM(*)

Parameters
size
: The valid range for *size* is 0 to 2097152. You can use asterisk as a value for *size* , MAXMEM(*), to indicate the highest possible value, which is also the default. NOMAXMEM, MAXMEM(0), and MAXMEM(*) are equivalent. Use the MAXMEM *size* sub-option if you want to specify a memory size of less value than the default.

Usage
If the memory specified by the MAXMEM option is insufficient for a particular optimization, the compilation is completed in such a way that the quality of the optimization is reduced, and a warning message is issued.

When a large *size* is specified for MAXMEM, compilation may be aborted because of insufficient virtual storage, depending on the source file being compiled, the size of the subprogram in the source, and the virtual storage available for the compilation.

The advantage of using the MAXMEM option is that, for large and complex applications, the compiler produces a slightly less-optimized object module and generates a warning message, instead of terminating the compilation with an error message of “insufficient virtual storage”.

Notes:

1. The limit that is set by MAXMEM is the amount of memory for specific optimizations, and not for the compiler as a whole. Tables that are required during the entire compilation process are not affected by or included in this limit.
2. Setting a large limit has no negative effect on the compilation of source files when the compiler needs less memory.
3. Limiting the scope of optimization does not necessarily mean that the resulting program will be slower, only that the compiler may finish before finding all opportunities to increase performance.
4. Increasing the limit does not necessarily mean that the resulting program will be faster, only that the compiler may be able to find opportunities to increase performance.
5. At OPT(3), the default for MAXMEM is set to (*).

IPA effects

If you specify the MAXMEM option for any compilation unit in the IPA Compile step, the compiler generates information for the IPA Link step. This option also affects the regular object module if you request one by specifying the IPA(OBJECT) option.

The option value you specify on the IPA Compile step for each IPA object file appears in the IPA Link step Compiler Options Map listing section.

If you specify the MAXMEM option on the IPA Link step, the value of the option is used. The IPA Link step Prolog and Partition Map listing sections display the value of the option.

If you do not specify the option on the IPA Link step, the value that it uses for a partition is the maximum MAXMEM value you specified for the IPA Compile step for any compilation unit that provided code for that partition. The IPA Link Step Prolog listing section does not display the value of the MAXMEM option, but the Partition Map listing section does.

Predefined macros

None.

MEMORY | NOMEMORY

Category

Compiler customization

Pragma equivalent

None.

Purpose

Improves compile-time performance by using a memory file in place of a temporary work file, if possible.

Syntax

```
    ┌─MEM───┐
►►──┴─NOMEM─┴──────────────────────────────────────►◄
```

Defaults

MEMORY

Usage

This option generally increases compilation speed, but you may require additional memory to use it. If you use this option and the compilation fails because of a storage error, you must increase your storage size or recompile your program using the NOMEMORY option. For information on how to increase storage size, see “Setting the region size for z/OS XL C/C++ applications” on page 455.

IPA effects

The MEMORY option has the same effect on the IPA Link step as it does on a regular compilation. If the IPA Link step fails due to an out-of-memory condition, provide additional virtual storage. If additional storage is unavailable, specify the NOMEMORY option.

Predefined macros

None.

Related information

See the *z/OS XL C/C++ Programming Guide* for more information on memory files.

METAL | NOMETAL (C only)

Category

Object code control

Pragma equivalent

None.

Purpose

Generates HLASM code that has no Language Environment run-time dependencies and follows the MVS linkage conventions for passing parameters, returning values, and setting up function save areas.

Syntax

```
      ┌─NOMETAL─┐
►►────┴─METAL───┴──────────────────────────────────────────────────────►◄
```

Defaults

NOMETAL

Usage

The z/OS XL C compiler-generated object code relies on Language Environment support. In addition to depending on Language Environment C run-time library functions, the XL C generated object code also depends on the establishment of an overall execution environment, which includes automatic storage. This Language Environment dependency prohibits you from using the XL C compiler to generate code to run in an environment where the Language Environment services do not exist.

The METAL-generated object code does not depend on any run-time environment. Any system services that the program needs can be obtained directly by using the system macros supplied by the operating system.

The compiler-generated code is in HLASM source code (as specified by the GENASM compiler option) so that your code can use system macros to acquire operating system services and so that your coded system macros can be properly expanded by the additional assembly process.

To save you the effort of always providing your own libraries, we provide a subset of the C library functions, which covers the commonly used basic functions. These functions include memory management functions, such as `malloc`. You can either use these METAL-supplied functions or supply your own run-time functions by including the appropriate header files.

For further information on programming with Metal C and the C run time that is provided, see *z/OS Metal C Programming Guide and Reference*.

You may need to switch addressing mode (AMODE) between programs. The default AMODE assigned by the XL C compiler is based on the LP64 compiler option or the ILP32 compiler option. AMODE 64 is assigned when LP64 is specified and AMODE 31 is assigned when ILP32 is specified. The METAL option enables the XL C compiler to generate code for calling an external function with an AMODE that is different from the default AMODE. This capability supports the creation of METAL C programs that require AMODE switching across functions. The resulting compiler generated code follows the linkage conventions expected by the called function, particularly in the areas of save area format and the parameter list width. You can use the `amode31` function attribute to mark an AMODE 31 function or the `amode64` function attribute to mark an AMODE 64 function in your source files. The `__ptr64` qualifier can be used when the METAL option is specified so that a 64-bit pointer can be handled by an AMODE 31 function without dereferencing it. For more information on the `amode31` function attribute, `amode64` function attribute, and the `__ptr64` qualifier, see *z/OS XL C/C++ Language Reference*. *z/OS Metal C Programming Guide and Reference* describes the impact of AMODE switching across functions on the save area chain in the user-supplied prolog or epilog code and the restrictions that apply to AMODE switching across functions.

Note: The METAL compiler option implies that the GENASM compiler option is specified, which means that METAL can only generate HLASM source code.

The METAL option disables support for packed-decimal and decimal floating-point data types.

METAL also disables:
- DLL
- RENT
- XPLINK
- IPA
- HOT
- DFP

METAL sets the following as defaults:
- ARCH(5)
- TUNE(5)
- CSECT
- HGPR(PRESERVE)
- FLOAT(IEEE)
- NOLONGNAME
- NODEBUG(FORMAT(DWARF),NOHOOK,SYMBOL)

METAL ignores the following:

- TARGET
- INLRPT
- GOFF
- INLINE when OPTIMIZE(0) is in effect
- All suboptions of INLINE

METAL ignores the following:

- **#pragma variable**
- **#pragma linkage**

In the z/OS UNIX System Services environment:

- The METAL option is specified by using the **-qmetal** option with the **xlc** command.

  Note: It is possible to use the METAL option with the c89 utility by specifying `-Wc,metal` and `-Wc,genasm`, but this method is not recommended.
- With the **-qmetal** option, the -S flag must be explicitly specified. Otherwise, the compiler issues an error message.
- The **as** command can be used to produce the desired object code from the compiler-generated HLASM source code.

Notes:

1. If the METAL option is specified but the DEBUG option is not specified, then the generated assembler source will not contain any debug type information. If this assembler source is fed into the as utility or CDAHLASM, the resulting .dbg file will only contain information at the assembler source level (in other words, it will not make any references to the original C source.)
2. The compiler-generated code does not establish code base registers.
3. Due to the flat name space and the case insensitivity required by HLASM, the XL C compiler prepends extra qualifiers to user names to maintain the uniqueness of each name seen by HLASM. This is referred to as *name-encoding*. External symbols are not subject to the name-encoding scheme as they need to be referenced by the exact symbol names.
4. The length of identifiers in the user code is constrained by the length supported by HLASM. It is your responsibility to ensure all external symbols are acceptable to HLASM. For local symbols, the compiler will apply truncation so that all symbols are acceptable to HLASM.
5. You need to provide the C library functions for your code. The inlined code generated for the selected C library functions is still available when the appropriate header file is included. IBM provides a subset of the C library functions that support the METAL-generated code.
6. It is your responsibility to ensure the correctness of your assembly code, including prolog and epilog code, and inlined assembly code.
7. When binding or linking, you may need to specify the ENTRY name.
8. No ASCII version of the Metal C runtime libraries is available, even though the ASCII compiler option is supported.
9. The HLASM GOFF option is required to assemble the compiler-generated code when the compiler LONGNAME option is specified.

The METAL compiler option also sets the default linkage convention to be the MVS system linkage described in the *MVS Programming: Assembler Services Guide*. For further information on MVS system linkage, see *z/OS Metal C Programming Guide and Reference*.

Predefined macros

- __IBM_METAL__ is predefined to 1 when METAL is in effect; otherwise it is undefined.
- __IBM_FAR_IS SUPPORTED__ is predefined to 1 when METAL is in effect; otherwise it is undefined.

Examples

For examples that describe how to use the METAL compiler option, see *z/OS Metal C Programming Guide and Reference*.

Related information

For more information on related compiler options, the **as** command, and the CDAHLASM utility, see:

- “ARMODE | NOARMODE (C only)” on page 73
- “ASMDATASIZE (C only)” on page 75
- “RESERVED_REG (C only)” on page 211
- “GENASM | NOGENASM (C only)” on page 120
- “PROLOG (C only)” on page 206
- “EPILOG (C only)” on page 106
- Chapter 20, “as — Use the HLASM assembler to produce object files,” on page 531
- Chapter 17, “CDAHLASM — Use the HLASM assembler to create DWARF debug information (C only),” on page 523

NAMEMANGLING (C++ only)

Category

Portability and migration

Pragma equivalent

#pragma namemangling (C++ only)

Purpose

Specifies the name mangling scheme for external symbol names which have C++ linkage.

Syntax

```
                    ┌─zOSV1R2──────────┐
►►──NAMEMANGLING──(─┼─ANSI─────────────┼─)──────────────────────────────────►◄
                    ├─zOSV1R11_ANSI────┤
                    ├─zOSV1R10_ANSI────┤
                    ├─zOSV1R9_ANSI─────┤
                    ├─zOSV1R8_ANSI─────┤
                    ├─zOSV1R7_ANSI─────┤
                    ├─zOSV1R5_ANSI─────┤
                    ├─zOSV1R5_DEFAULT──┤
                    ├─OSV2R10──────────┤
                    └─COMPAT───────────┘
```

Defaults

By default, the NAMEMANGLING option is set as follows:

- ZOSV1R2 — If LANGLVL is set to ANSI, EXTENDED or EXTENDED0X.
- COMPAT — If LANGLVL is set to COMPAT92.
- ANSI — If LP64 is set; the effect of LP64 takes precedence over the effect of LANGLVL.

Parameters

The NAMEMANGLING compiler option enables you to choose between the following name mangling schemes:

ANSI
: This scheme complies with the most recent C++ language features and is equivalent to zOSV1R11_ANSI.

zOSV1R11_ANSI
: This scheme is compatible with z/OS XL C++ V1R11 link modules that were created with NAMEMANGLING(ANSI) or #pragma namemangling(ansi).

zOSV1R10_ANSI
: This scheme is compatible with z/OS XL C++ V1R10 link modules that were created with NAMEMANGLING(ANSI) or #pragma namemangling(ansi).

zOSV1R9_ANSI
: This scheme is compatible with z/OS XL C++ V1R9 link modules that were created with NAMEMANGLING(ANSI) or #pragma namemangling(ansi).

zOSV1R8_ANSI
: This scheme is compatible with z/OS XL C++ V1R8 link modules that were created with NAMEMANGLING(ANSI) or #pragma namemangling(ansi).

zOSV1R7_ANSI
: This scheme is compatible with z/OS XL C++ V1R7 link modules that were created with NAMEMANGLING(ANSI) or #pragma namemangling(ansi).

zOSV1R5_ANSI
: This scheme is compatible with z/OS XL C++ V1R5 link modules that were created with NAMEMANGLING(ANSI) or #pragma namemangling(ansi).

zOSV1R5_DEFAULT
: This scheme ensures backwards compatibility with previous z/OS XL C++ versions and is equivalent to ZOSV1R2.

zOSV1R2
: This scheme is compatible with z/OS XL C++ V1R2 link modules that were created with NAMEMANGLING(ANSI) or #pragma namemangling(ansi).

OSV2R10
: This scheme is compatible with the link modules created by OS/390 C++ V2R10 or previous versions, or with link modules that were created with NAMEMANGLING(COMPAT) or #pragma namemangling(compat).

COMPAT
: This scheme is equivalent to OSV2R10.

Usage

Name mangling is the encoding of variable names into unique names so that linkers can separate common names in the language. With respect to the C++ language, name mangling is commonly used to facilitate the overloading feature and visibility within different scopes.

Note: If the NAMEMANGLING compiler option is not specified, LANGLVL(EXTENDED) and LANGLVL(ANSI) set NAMEMANGLING to zOSV1R2. LANGLVL(COMPAT92) sets NAMEMANGLING to COMPAT.

The NAMEMANGLING compiler option takes precedence over the LP64 compiler option. The LP64 compiler option takes precedence over the LANGLVL compiler option. When the NAMEMANGLING and LANGLVL compiler options are specified, the last specified option takes precedence. This is to preserve the V1R2 behavior so that existing code is not broken.

Predefined macros

None.

Examples

The following table shows some examples of the NAMEMANGLING options that are in effect when certain compiler options are specified:

Table 29. Examples of NAMEMANGLING in effect

Compiler option(s) specified	NAMEMANGLING in effect
NAMEMANGLING(zOSV1R2)	zOSV1R2
LANGLVL(COMPAT92)	COMPAT
LP64	ANSI
NAMEMANGLING(zOSV1R2) LANGLVL(COMPAT92)	COMPAT
LANGLVL(COMPAT92) NAMEMANGLING(zOSV1R2)	zOSV1R2
NAMEMANGLING(zOSV1R2) LP64	zOSV1R2
LP64 NAMEMANGLING(zOSV1R2)	zOSV1R2
LANGLVL(COMPAT92) LP64	ANSI
LP64 LANGLVL(COMPAT92)	ANSI
NAMEMANGLING(zOSV1R2) LANGLVL(COMPAT92) LP64	COMPAT
NAMEMANGLING(zOSV1R2) LP64 LANGLVL(COMPAT92)	COMPAT
LP64 NAMEMANGLING(zOSV1R2) LANGLVL(COMPAT92)	COMPAT
LP64 LANGLVL(COMPAT92) NAMEMANGLING(zOSV1R2)	zOSV1R2
LANGLVL(COMPAT92) LP64 NAMEMANGLING(zOSV1R2)	zOSV1R2
LANGLVL(COMPAT92) NAMEMANGLING(zOSV1R2) LP64	zOSV1R2

Related information

- For information on the **#pragma namemanglingrule** directive, see *z/OS XL C/C++ Language Reference*.
- “LANGLVL” on page 147
- “LP64 | ILP32” on page 169

NESTINC | NONESTINC

Category

Compiler input

Pragma equivalent

None.

Purpose

Specifies the number of nested include files to be allowed in your source program.

When the NESTINC compiler option is in effect, you can specify the maximum limit of nested include files.

When the NONESTINC compiler option is in effect, you are specifying NESTINC(255).

Syntax

```
      ┌─NEST──(──num──)─┐
►►────┴─NONEST──────────┴──────────────────────────────────────────►◄
```

Defaults

NESTINC(255)

Parameters

num
   You can specify a limit of any integer from 0 to `SHRT_MAX`, which indicates the maximum limit, as defined in the header file `LIMITS.H`. To specify the maximum limit, use an asterisk (*). If you specify an invalid value, the compiler issues a warning message, and uses the default limit, which is 255.

Usage

If you use heavily nested include files, your program requires more storage to compile.

Predefined macros

None.

OBJECT | NOOBJECT

Category

Compiler output

Pragma equivalent

#pragma options (object) (C only), **#pragma options (noobject)** (C only)

Purpose

Produces an object module, and stores it in the file that you specify, or in the data set associated with SYSLIN.

Syntax

```
       ┌─OBJ───┐
►►─────┴─NOOBJ─┴─┬──────────────────────────────────────────────────┬─────────────►◄
                 └─(──┬─Sequential filename──────────────────┬─)─┘
                      ├─Partitioned data set─────────────────┤
                      ├─Partitioned data set (member)────────┤
                      ├─z/OS UNIX System Services filename───┤
                      └─z/OS UNIX System Services directory──┘
```

Defaults

OBJECT

Parameters

Sequential filename
Specifies the sequential data set file name for the object module.

Partitioned data set
Specifies the partitioned data set for the object module.

Partitioned data set (member)
Specifies the partitioned data set (member) for the object module.

z/OS UNIX System Services filename
Specifies the z/OS UNIX System Services file name for the object module.

z/OS UNIX System Services directory
Specifies the z/OS UNIX System Services directory for the object module.

Usage

The GOFF compiler option specifies the object format that will be used to encode the object information.

You can specify OBJECT(*filename*) to place the object module in that file. If you do not specify a file name for the OBJECT option, the compiler uses the SYSLIN ddname if you allocated it. Otherwise, the compiler generates a file name as follows:

- If you are compiling a data set, the compiler uses the source file name to form the name of the object module data set. The high-level qualifier is replaced with the userid under which the compiler is running, and `.OBJ` is appended as the low-level qualifier.
- If you are compiling a z/OS UNIX System Services file, the compiler stores the object module in a file that has the name of the source file with an `.o` extension.

The NOOBJECT option can optionally take a file name suboption. This file name then becomes the default. If you subsequently use the OBJECT option without a file name suboption, the compiler uses the file name that you specified in the earlier NOOBJECT. For example, the following specifications have the same result:

```
CXX HELLO (NOOBJ(./hello.obj) OBJ
CXX HELLO (OBJ(./hello.obj)
```

If you specify OBJ and NOOBJ multiple times, the compiler uses the last specified option with the last specified suboption. For example, the following specifications have the same result:

```
CXX HELLO (NOOBJ(./hello.obj) OBJ(./n1.obj)  NOOBJ(./test.obj) OBJ
CXX HELLO (OBJ(./test.obj)
```

If you request a listing by using the SOURCE, INLRPT, or LIST option, and you also specify OBJECT, the name of the object module is printed in the listing prolog.

In the z/OS UNIX System Services environment, you can specify the object location by using the **-c -o objectname** options when using the **c89**, **cc**, **c++**, **cxx**, **xlc**, **xlC**, or **xlc++** commands. In the z/OS UNIX System Services environment, the **-o** flag option is used to specify the name of the object file.

Note: If you use the following form of the command in a JES3 batch environment where xxx is an unallocated data set, you may get undefined results.

```
OBJECT(xxx)
```

IPA effects

IPA Compile uses the same rules as the regular compile to determine the file name or data set name of the object module it generates. If you specify NOOBJECT, the IPA Compile step suppresses object output, but performs all analysis and code generation processing (other than writing object records).

Note: You should not confuse the OBJECT compiler option with the IPA(OBJECT) suboption. The OBJECT option controls file destination. The IPA(OBJECT) suboption controls file content. Refer to "IPA | NOIPA" on page 139 for information about the IPA(OBJECT) suboption.

When you use the c89 utility for IPA Link invocation, the object is assigned to //DD:SYSLIPA and should not be changed by specifying the OBJECT compiler option.

c89 does not normally keep the object file output from the IPA Link step, as the output is an intermediate file in the link-edit phase processing. To find out how to make the object file permanent, refer to the *prefix*_TMPS environment variable information in the **c89** section of *z/OS UNIX System Services Command Reference*.

Note: The OBJECT compiler option is not the same as the OBJECT suboption of the IPA option. Refer to "IPA | NOIPA" on page 139 for information about the IPA(OBJECT) option.

Predefined macros

None.

Related information

For more information on related compiler options and the c89 utility, see:

- "GOFF | NOGOFF" on page 121
- "SOURCE | NOSOURCE" on page 226
- "INLRPT | NOINLRPT" on page 137
- "LIST | NOLIST" on page 163
- "IPA | NOIPA" on page 139
- Chapter 21, "c89 — Compiler invocation using host environment variables," on page 535

OBJECTMODEL (C++ only)

Category

Object code control

Pragma equivalent

#pragma object_model (C++ only)

Purpose

Sets the object model to be used for structures, unions, and classes.

Syntax

```
                 ┌─COMPAT─┐
►►──OBJECTMODEL──┴─IBM────┴──────────────────────────────────────────────►◄
```

Defaults

OBJECTMODEL(COMPAT)

Parameters

COMPAT

COMPAT is compatible with name mangling and the virtual function table that was available with the previous releases of the C++ compiler.

IBM

Select IBM if you want improved performance. This is especially true for class hierarchies with many virtual base classes. The size of the derived class is considerably smaller and access to the virtual function table is faster.

Notes:

1. When you compile with the OBJECTMODEL(IBM) option, and the `dynamic_cast` operator is used in a constructor, a destructor, or in functions called from a constructor or destructor, the `dynamic_cast` operator has the following behavior:
   - Does not return a pointer or a reference to the derived object from the class for the constructor or destructor.
   - Returns NULL.
2. When you compile with the LP64 compiler option, the OBJECTMODEL(IBM) compiler option is specified along with XPLINK.
3. In order to use the OBJECTMODEL(IBM) option, the XPLINK option must be specified. If XPLINK is not specified, the compiler will issue a warning and use the default OBJECTMODEL(COMPAT) setting.

Usage

z/OS XL C++ includes two ways to compile your programs using different object models. The two object models, COMPAT and IBM, differ in the following areas:

- Layout for the virtual function table
- Name mangling scheme

IPA effects

The IPA Link step does not accept the OBJECTMODEL option. The compiler issues a warning message if you specify this option in the IPA Link step.

Predefined macros

- __OBJECT_MODEL_COMPAT__ is predefined to a value of 1 when the OBJECTMODEL(COMPAT) compiler option is in effect; otherwise it is undefined.
- __OBJECT_MODEL_IBM__ is predefined to a value of 1 when the OBJECTMODEL(IBM) compiler option is in effect; otherwise it is undefined.

Related information

- For more information on the XPLINK compiler option, see "XPLINK | NOXPLINK" on page 260.

OE | NOOE

Category

Compiler input

Pragma equivalent

None.

Purpose

Specifies the rules used when searching for files specified with `#include` directives.

Syntax

```
     ┌─NOOE─┐
►►───┴─OE───┴──┬──────────────────┬──────────────────────────────────────────►◄
               └─(──filename──)───┘
```

Defaults

NOOE

When compiling in the z/OS UNIX System Services Shell environment, the default is OE.

Parameters

filename
: Specifies the path that is used when searching for files specified with `#include` directives.

Note: Diagnostics and listing information will refer to the file name that is specified for the OE option (in addition to the search information).

Usage

When the OE compiler option is in effect, the compiler uses the POSIX.2 standard rules when searching for files specified with `#include` directives. These rules state that the path of the file currently being processed is the path used as the starting point for searches of include files contained in that file.

The NOOE option can optionally take a *filename* suboption. This *filename* then becomes the default. If you subsequently use the OE option without a *filename* suboption, the compiler uses the *filename* that you specified in the earlier NOOE.

Example: The following specifications have the same result:

```
xlc hello.c -qnooe=./hello.c -qoe
```

```
xlc hello.c -qoe=./hello.c
```

If you specify OE and NOOE multiple times, the compiler uses the last specified option with the last specified suboption.

Example: The following specifications have the same result:

```
xlc hello.c -qnooe=./hello.c -qoe=./n1.c -qnooe=./test.c -qoe
```

```
xlc hello.c -qoe=./test.c
```

When the OE option is in effect and the main input file is a z/OS UNIX file, the path of *filename* is used instead of the path of the main input file name. If the file names indicated in other options appear ambiguous between z/OS and the z/OS UNIX file system, the presence of the OE option tells the compiler to interpret the ambiguous names as z/OS UNIX file names. User include files that are specified in the main input file are searched starting from the path of *filename*. If the main input file is not a z/OS UNIX file, *filename* is ignored.

For example, if the compiler is invoked to compile a z/OS UNIX file `/a/b/hello.c` it searches directory `/a/b/` for include files specified in `/a/b/hello.c`, in accordance with POSIX.2 rules . If the compiler is invoked with the `OE(/c/d/hello.c)` option for the same source file, the directory specified as the suboption for the OE option, `/c/d/`, is used to locate include files specified in `/a/b/hello.c`.

IPA effects
On the IPA Link step, the OE option controls the display of file names.

Predefined macros
None.

OFFSET | NOOFFSET

Category
Listings, messages and compiler information

Pragma equivalent
None.

Purpose
Lists offset addresses relative to entry points of functions.

When the OFFSET compiler option is in effect, the compiler displays the offset addresses relative to the entry point or start of each function in the pseudo assembly listing generated by the LIST option. The OFFSET compiler option also prints the CSECT Offset field in the pseudo assembly listing for a function, which shows the offset of the function in the CSECT.

When the NOOFFSET compiler option is in effect, the compiler displays the offset addresses relative to the beginning of the generated code in the pseudo assembly listing generated by the LIST option and does not display the entry point.

Syntax

```
      ┌─NOOF─┐
►►────┴─OF───┴──────────────────────────────────────────────────►◄
```

Defaults
NOOFFSET

In the z/OS UNIX System Services environment, this option is turned on by specifying -V when using the **c89**, **cc** or **c++** commands.

Usage
If you use the OFFSET option, you must also specify the LIST option to generate the pseudo assembly listing. If you specify the OFFSET option but omit the LIST option, the compiler generates a warning message, and does not produce a pseudo assembly listing.

IPA effects
If you specify the IPA(OBJECT) option (that is, if you request code generation), the OFFSET option has the same effect on the IPA Compile step as it does on a regular compilation.

If you specify the LIST option during IPA Link, the IPA Link listing will be affected (in the same way as a regular compilation) by the OFFSET option setting in effect at that time.

The OFFSET option that you specified on the IPA Compile step has no effect on the IPA Link step.

Predefined macros
None.

Related information
For more information on related compiler options, see
- "LIST | NOLIST" on page 163
- "IPA | NOIPA" on page 139

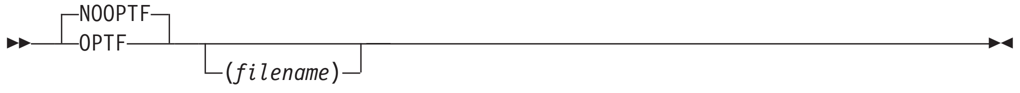
OPTFILE | NOOPTFILE

Category
Compiler customization

Pragma equivalent
None.

Purpose
Specifies where the compiler should look for additional compiler options.

Syntax



Defaults
NOOPTFILE

Parameters
filename
: Specifies an alternative file where the compiler should look for compiler options.

  You can specify any valid *filename*, including a DD name such as (`DD:MYOPTS`). The DD name may refer to instream data in your JCL. If you do not specify *filename*, the compiler uses `DD:SYSOPTF`.

Usage
The NOOPTF option can optionally take a *filename* suboption. This *filename* then becomes the default. If you subsequently use the OPTF option without a *filename* suboption, the compiler uses the *filename* that you specified in the earlier NOOPTF.

Example: The following specifications have the same result:
```
CXX HELLO (NOOPTF(./hello.opt) OPTF
CXX HELLO (OPTF(./hello.opt)
```

The options are specified in a free format with the same syntax as they would have on the command line or in JCL. The code points for the special characters `\f`, `\v`, and `\t` are whitespace characters. Everything that is specified in the file is taken to be part of a compiler option (except for the continuation character), and unrecognized entries are flagged. Nothing on a line is ignored.

If the record format of the options file is fixed and the record length is greater than 72, columns 73 to the end-of-line are treated as sequence numbers and are ignored.

Notes:

1. Comments are supported in an option file used in the OPTFILE option. When a line begins with the # character, the entire line is ignored, including any continuation character. The option files are encoded in the IBM-1047 codepage.
2. You cannot nest the OPTFILE option. If the OPTFILE option is also used in the file that is specified by another OPTFILE option, it is ignored.
3. If you specify NOOPTFILE after a valid OPTFILE, it does not undo the effect of the previous OPTFILE. This is because the compiler has already processed the options in the options file that you specified with OPTFILE. The only reason to use NOOPTFILE is to specify an option file name that a later specification of OPTFILE can use.
4. If the file cannot be opened or cannot be read, a warning message is issued and the OPTFILE option is ignored.
5. The options file can be an empty file.
6. **Example:** You can use an option file only once in a compilation. If you use the following options:

   ```
   OPTFILE(DD:OF)     OPTFILE
   ```

   the compiler processes the option `OPTFILE(DD:OF)`, but the second option `OPTFILE` is not processed. A diagnostic message is produced, because the second specification of `OPTFILE` uses the same option file as the first.

   Example: You can specify OPTFILE more than once in a compilation, if you use a different options file with each specification:

   ```
   OPTFILE(DD:OF)   OPTFILE(DD:OF1)
   ```

IPA effects

The OPTFILE option has the same effect on the IPA Link step as it does on a regular compilation.

Predefined macros

None.

Examples

1. Suppose that you use the following JCL:

   ```
   //   CPARM='SO OPTFILE(PROJ1OPT) EXPORTALL'
   ```

   If the file `PROJ1OPT` contains `OBJECT LONGNAME`, the effect on the compiler is the same as if you specified the following:

   ```
   //   CPARM='SO OBJECT LONGNAME EXPORTALL'
   ```
2. Suppose that you include the following in the JCL:

   ```
   //   CPARM='OBJECT OPTFILE(PROJ1OPT) LONGNAME OPTFILE(PROJ2OPT) LIST'
   ```

If the file PROJ1OPT contains SO LIST and the file PROJ2OPT contains GONUM, the net effect to the compiler is the same as if you specified the following:

```
//   CPARM='OBJECT SO LIST LONGNAME GONUM LIST'
```

3. If an F80 format options file looks like this:

```
|  ...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8
                            LIST                                      00000010
                                                              INLRPT      00000020
MARGINS                                                                   00000030
   OPT                                                                    00000040
   XREF                                                                   00000050
```

The compile has the same effect as if you specified the following options on the command line or in a PARMS= statement in your JCL:

```
  LIST INLRPT MARGINS OPT XREF
```

4. The following example shows how to use the options file as an instream file in JCL:

```
//COMP  EXEC CBCC,
//      INFILE='<userid>.USER.CXX(LNKLST)',
//      OUTFILE='<userid>.USER.OBJ(LNKLST),DISP=SHR ',
//      CPARM='OPTFILE(DD:OPTION)'
//OPTION DD DATA,DLM=@@
                            LIST
                                                              INLRPT
MARGINS
   OPT
   XREF
@@
```

OPTIMIZE | NOOPTIMIZE

Category

Optimization and tuning

Pragma equivalent

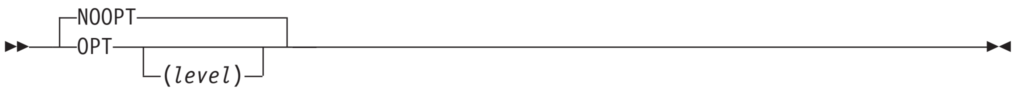
#pragma options (optimize) (C only), **#pragma options (nooptimize)** (C only)

#pragma option_override(subprogram_name, ″OPT(LEVEL,n)″)

Purpose

Specifies whether to optimize code during compilation and, if so, at which level.

Syntax



Defaults

For C and C++, the default option is NOOPTIMIZE. For IPA Link, the default option is OPTIMIZE(2).

For the z/OS UNIX System Services utilities, the default for a regular compile is OPTIMIZE(0) for NOIPA.

Parameters

level

level can have the following values:

0 Indicates that no optimization is to be done; this is equivalent to NOOPTIMIZE. You should use this option in the early stages of your application development since the compilation is efficient but the execution is not. This option also allows you to take full advantage of the debugger.

1 OPTIMIZE(1) is an obsolete artifact of the OS/390 Version 2 Release 4 compiler. We suggest that you use OPTIMIZE(2), which may help avoid compatibility issues.

2 Indicates that global optimizations are to be performed. You should be aware that the size of your functions, the complexity of your code, the coding style, and support of the ISO standard may affect the global optimization of your program. You may need significant additional memory to compile at this optimization level.

3 Performs additional optimizations to those performed with OPTIMIZE(2). OPTIMIZE(3) is recommended when the desire for run-time improvement outweighs the concern for minimizing compilation resources. Increasing the level of optimization may or may not result in additional performance improvements, depending on whether additional analysis detects further opportunities for optimization. Compilation may require more time and machine resources.

Use the STRICT option with OPTIMIZE(3) to turn off the aggressive optimizations that might change the semantics of a program. STRICT combined with OPTIMIZE(3) invokes all the optimizations performed at OPTIMIZE(2) as well as further loop optimizations. The STRICT compiler option must appear after the OPTIMIZE(3) option, otherwise it is ignored.

The aggressive optimizations performed when you specify OPTIMIZE(3) are:

- Aggressive code motion, and scheduling on computations that have the potential to raise an exception, are allowed.
- Conformance to IEEE rules are relaxed. With OPTIMIZE(2), certain optimizations are not performed because they may produce an incorrect sign in cases with a zero result, and because they remove an arithmetic operation that may cause some type of floating-point exception. For example, `X + 0.0` is not folded to `X` because, under IEEE rules, `-0.0 + 0.0 = 0.0`, which is `-X`. In some other cases, some optimizations may perform optimizations that yield a zero result with the wrong sign. For example, `X - Y * Z` may result in a `-0.0` where the original computation would produce `0.0`. In most cases, the difference in the results is not important to an application and OPTIMIZE(3) allows these optimizations.
- Floating-point expressions may be rewritten. Computations such as `a*b*c` may be rewritten as `a*c*b` if, for example, an opportunity exits to get a common subexpressions by such rearrangement. Replacing a divide with a multiply by the reciprocal is another example of reassociating floating-point computations.

no level
OPTIMIZE specified with no level defaults, depending on the compilation environment and IPA mode.

Usage

When the OPTIMIZE compiler option is in effect, the compiler is instructed to optimize the generated machine instructions to produce a faster running object module. This type of optimization can also reduce the amount of main storage that is required for the generated object module.

Note: When the compiler is invoked using the **c89**, **cc**, **c++**, **xlc** or **xlC** commands under z/OS UNIX System Services, the optimization level is specified by the compiler flag **-O** (the letter O). The OPTIMIZE option has no effect on these commands.

In the z/OS UNIX System Services environment, **-g** implies NOOPTIMIZE.

Using OPTIMIZE will increase compile time over NOOPTIMIZE and may have greater storage requirements. During optimization, the compiler may move code to increase run-time efficiency; as a result, statement numbers in the program listing may not correspond to the statement numbers used in run-time messages.

The OPTIMIZE option will control the overall optimization value. Any subprogram-specific optimization levels specified at compile time by **#pragma option_override(subprogram_name, ″OPT(LEVEL,n)″)** directives will be retained. Subprograms with an OPT(LEVEL,0) value will receive minimal code generation optimization. Subprograms may not be inlined or inline other subprograms. Generate and check the inline report to determine the final status of inlining.

Inlining of functions in conjunction with other optimizations provides optimal run-time performance. See “INLINE | NOINLINE” on page 133 for more information about the INLINE option and the optimization information in *z/OS XL C/C++ Programming Guide*.

If you specify OPTIMIZE with TEST or DEBUG, you can only set breakpoints at function call, function entry, function exit, and function return points.

The option INLINE is automatically turned on when you specify OPTIMIZE, unless you have explicitly specified the NOINLINE option.

A comment that notes the level of optimization will be generated in your object module to aid you in diagnosing your program.

Effect of ANSIALIAS: When the ANSIALIAS option is specified, the optimizer assumes that pointers can point only to objects of the same type, and performs more aggressive optimization. However, if this assumption is not true and ANSIALIAS is specified, wrong program code could be generated. If you are not sure, use NOANSIALIAS.

IPA effects

During a compilation with IPA Compile-time optimizations active, any subprogram-specific optimization levels specified by **#pragma option_override(subprogram_name, ″OPT(LEVEL,n)″)** directives will be retained. Subprograms with an OPT(LEVEL,0) value will receive minimal IPA and code generation optimization. Subprograms may not be inlined or inline other subprograms. Generate and check the inline report to determine the final status of inlining.

On the IPA Compile step, all values (except for (0)) of the OPTIMIZE compiler option and the OPT suboption of the IPA option have an equivalent effect.

Refer to the descriptions of the OPTIMIZE and LEVEL suboptions of the IPA option in "IPA | NOIPA" on page 139 for information about using the OPTIMIZE option under IPA.

OPTIMIZE(2) is the default for the IPA Link step, but you can specify any level of optimization. The IPA Link step Prolog listing section will display the value of this option.

This optimization level will control the overall optimization value. Any subprogram-specific optimization levels specified at IPA Compile time by **#pragma option_override(subprogram_name, ″OPT(LEVEL,n)″)** directives will be retained. Subprograms with an OPT(LEVEL,0)) value will receive minimal IPA and code generation optimization, and will not participate in IPA Inlining.

The IPA Link step merges and optimizes your application code, and then divides it into sections for code generation. Each of these sections is a partition. The IPA Link step uses information from the IPA Compile step to determine if a subprogram can be placed in a particular partition. Only compatible subprograms are included in a given partition. Compatible subprograms have the same OPTIMIZE setting.

The OPTIMIZE setting for a partition is set to that of the first subprogram that is placed in the partition. Subprograms that follow are placed in partitions that have the same OPTIMIZE setting. An OPTIMIZE(0) mode is placed in an OPTIMIZE(0) partition, and an OPTIMIZE(2) is placed in an OPTIMIZE(2) partition.

The option value that you specified for each IPA object file on the IPA Compile step appears in the IPA Link step Compiler Options Map listing section.

The Partition Map sections of the IPA Link step listing and the object module END information section display the value of the OPTIMIZE option. The Partition Map also displays any subprogram-specific OPTIMIZE values.

If you specify OPTIMIZE(2) for the IPA Link step, but only OPTIMIZE(0) for the IPA Compile step, your program may be slower or larger than if you specified OPTIMIZE(2) for the IPA Compile step. This situation occurs because the IPA Compile step does not perform as many optimizations if you specify OPTIMIZE(0).

Refer to the descriptions for the OPTIMIZE and LEVEL suboptions of the IPA option in "IPA | NOIPA" on page 139 for information about using the OPTIMIZE option under IPA.

Predefined macros

__OPTIMIZE__ is defined to the value specified by the OPTIMIZE compiler option; it is undefined if NOOPTIMIZE is used.

Related information

For more information on related compiler options, see:

- "TEST | NOTEST" on page 249
- "DEBUG | NODEBUG" on page 94

- “ANSIALIAS | NOANSIALIAS” on page 66

PHASEID | NOPHASEID

Category

Listings, messages and compiler information

Pragma equivalent

None.

Purpose

Causes each compiler component (phase) to issue an informational message as each phase begins execution, which assists you with determining the maintenance level of each compiler component (phase). This message identifies the compiler phase module name, product identification, and build level.

Syntax

```
     ┌─NOPHASEID─┐
►►───┴─PHASEID───┴───────────────────────────────────────────────►◄
```

Defaults

NOPHASEID

Usage

The compiler issues a separate CCN0000(I) message each time compiler execution causes a given compiler component (phase) to be entered. This could happen many times for a given compilation.

The FLAG option has no effect on the PHASEID informational message.

In the z/OS UNIX System Services environment, **-qphsinfo** is synonymous with the PHASEID compiler option.

Note: The compiler saves phase ID information for all active compiler phases in an executable using the Saved Option String feature even if you don’t specify the PHASEID compiler option. For more information on Saved Option String, see *z/OS XL C/C++ Programming Guide*.

Predefined macros

None.

PLIST

Category

Object code control

Pragma equivalent

None.

Purpose

Specifies that the original operating system parameter list should be available.

Syntax

```
               ┌─HOST─┐
►►──PLIST──(──┴─OS───┴──)──────────────────────────────────────────►◄
```

Defaults

PLIST(HOST)

Parameters

HOST
: If you specify PLIST(HOST), the parameters are presented to `main()` as an argument list (`argc, argv`).

OS
: If you specify PLIST(OS), the parameters are passed without restructuring, and the standard calling conventions of the operating system are used. See *z/OS Language Environment Programming Guide* for details on how to access these parameters.

Usage

When compiling `main()` programs, use the PLIST option to direct how the parameters from the caller are passed to `main()`.

If you are compiling a `main()` program to run under IMS, you must specify the PLIST(OS) and TARGET(IMS) options together.

IPA effects

If you specified PLIST for any compilation unit in the IPA Compile step, it generates information for the IPA Link step. This option also affects the regular object module if you request one by specifying the IPA(OBJECT) option.

If you specify PLIST for the IPA Compile step, you do not need to specify it again on the IPA Link step. The IPA Link step uses the information generated for the compilation unit that contains the `main()` function, or for the first compilation unit it finds if it cannot find a compilation unit containing `main()`.

If you specify this option on both the IPA Compile and the IPA Link steps, the setting on the IPA Link step overrides the setting on the IPA Compile step. This situation occurs whether you use PLIST as a compiler option or specify it using the **#pragma runopts** directive (on the IPA Compile step).

Predefined macros

None.

Related information

For more information on the TARGET compiler option, see “TARGET” on page 240.

PORT | NOPORT (C++ only)

Category

Portability and migration

Pragma equivalent

None.

Purpose

Adjusts the error recovery action that the compiler takes when it encounters an ill-formed **#pragma pack** directive.

When the PORT compiler option is in effect, the compiler uses the specified error recovery mode.

When the NOPORT compiler option is in effect, the compiler uses the default error recovery mode.

Syntax

```
   ┌─NOPORT─┐
►►─┴─PORT───┴──┬───────────────────────┬──────────────────────────────────────►◄
               └─(──┬─PPS───┬──)───────┘
                    └─NOPPS─┘
```

Defaults

NOPORT(NOPPS)

Parameters

PPS
  When you specify PORT(PPS), the compiler uses the strict error recovery mode.

NOPPS
  When you specify PORT(NOPPS), the compiler uses the default error recovery mode.

Usage

When you specify PORT without a suboption, the suboption setting is inherited from the default setting or from previous PORT specifications.

When the default error recovery mode is active, the compiler recovers from errors in the **#pragma pack** directive as follows:

- **#pragma pack (***first_value*
  - If *first_value* is a valid suboption for **#pragma pack**, packing is done as specified by *first_value*. The compiler detects the missing closing parentheses and issues a warning message.
  - If *first_value* is not a valid suboption for **#pragma pack**, no packing changes are made. The compiler ignores the **#pragma pack** directive and issues a warning message.
- **#pragma pack (***first_value bad_tokens***)**
  - If *first_value* is a valid suboption for **#pragma pack**, packing is done as specified by *first_value*. If *bad_tokens* is invalid, the compiler detects it and issues a warning message.
  - If *first_value* is not a valid suboption for **#pragma pack**, no packing changes will be performed. The compiler will ignore the **#pragma pack** directive and issue a warning message.
- **#pragma pack (***valid_value***)** *extra_trailing_tokens*

  The compiler ignores the extra text and issues an information message.

To use the strict error recovery mode of the compiler, you must explicitly request it by specifying PORT(PPS).

When the strict error recovery mode is active, and the compiler detects errors in the **#pragma pack** directive, it ignores the pragma and does not make any packing changes.

Example: For example, the compiler detects errors for any of the following specifications of the **#pragma pack** directive:

```
#pragma pack(first_value

#pragma pack(first_value bad_tokens

#pragma pack(valid_value) extra_trailing_tokens
```

See *z/OS XL C/C++ Language Reference* for more information on **#pragma pack**.

IPA effects

The IPA Link step issues a diagnostic message if you specify the PORT option for that step.

Predefined macros

None.

PPONLY | NOPPONLY

Category

Compiler output

Pragma equivalent

None.

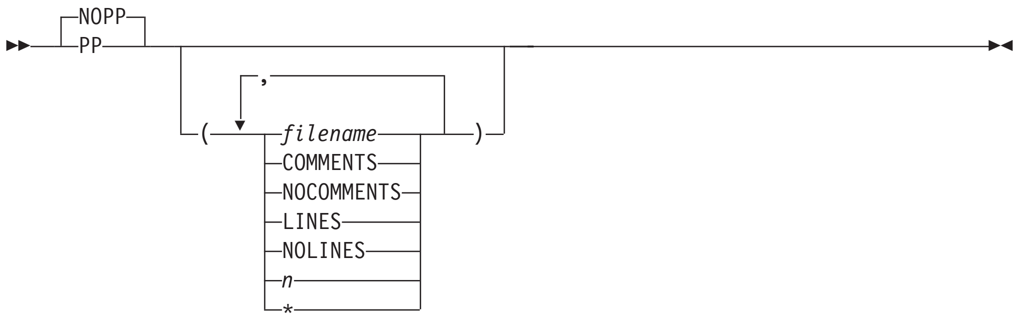
Purpose

Specifies that only the preprocessor is to be run and not the compiler.

When the PPONLY compiler option is in effect, the output of the preprocessor consists of the original source file with all the macros expanded and all the include files inserted. It is in a format that can be compiled.

When the NOPPONLY compiler option is in effect, both the preprocessor and the compiler are used to compile the source file.

Syntax



Defaults

NOPPONLY

For the z/OS UNIX System Services utilities, the default for a regular compile is NOPPONLY(NOCOMMENTS, NOLINES, /dev/fd1, 2048).

In the z/OS UNIX System Services environment, this option is turned on by specifying the -E flag option when using the c89 utility to invoke the compiler. When using the xlc utility, this option can be turned on by specifying the -E or -P flag options, or by specifying the **-qpponly** compiler option in a manner similar to specifying the PPONLY option in JCL or TSO compiler invocations.

Parameters

COMMENTS | NOCOMMENTS
The COMMENTS suboption preserves comments in the preprocessed output. The default is NOCOMMENTS.

LINES | NOLINES
The LINES suboption issues `#line` directives at include file boundaries, block boundaries and where there are more than 3 blank lines. The default is NOLINES.

filename
The name for the preprocessed output file. The *filename* may be a data set or a z/OS UNIX file. If you do not specify a file name for the PPONLY option, the `SYSUT10` ddname is used if it has been allocated. If `SYSUT10` has not been allocated, the file name is generated as follows:

- If a data set is being compiled, the name of the preprocessed output data set is formed using the source file name. The high-level qualifier is replaced with the userid under which the compiler is running, and `.EXPAND` is appended as the low-level qualifier.
- If the source file is a z/OS UNIX file, the preprocessed output is written to a z/OS UNIX file that has the source file name with `.i` extension.

Note: If you are using the xlc utility and you do not specify the file name, the preprocessed output goes to `stdout`. If -E or -P is also specified, the output file is determined by the -E option. The -E flag option maps to `PP(stdout)`. -P maps to PP(default_name). default_name is constructed using the source file name as the base and the suffix is replaced with the appropriate suffix, as defined by the `isuffix`, `isuffix_host`, `ixxsuffix`, and `ixxsuffix_host` configuration file attributes. See Chapter 24, "xlc — Compiler invocation using a customizable configuration file," on page 577 for further information on the xlc utility.

n If a parameter *n*, which is an integer between 2 and 32752 inclusive, is specified, all lines are folded at column *n*. The default for *n* is 72.

Note: If the PPONLY output is directed into an existing file, and *n* is larger than the maximum record length of the file, then all lines are folded to fit into the output file, based on the record length of the output file.

* If an asterisk (*) is specified, the lines are folded at the maximum record length of 32752. Otherwise, all lines are folded to fit into the output file, based on the record length of the output file.

Usage

PPONLY output is typically requested when reporting a compiler problem to IBM using a Problem Management Record (PMR), so your build process should be able to produce a PPONLY file on request.

Note: For further information on the PMR process, refer to techsupport.services.ibm.com/guides/handbook.html.

PPONLY also removes conditional compilation constructs like `#if`, and `#ifdef`.

Note: If the PPONLY output is directed into an existing file, the record length of the file will be used to override the value of *n* if that value is bigger than the maximum record length.

The PPONLY suboptions are cumulative. If you specify suboptions in multiple instances of PPONLY and NOPPONLY, all the suboptions are combined and used for the last occurrence of the option.

Example: The following three specifications have the same result:

```
CXX HELLO (NOPPONLY(./aa.exp) PPONLY(LINES) PPONLY(NOLINES)
```

```
CXX HELLO (PPONLY(./aa.exp,LINES,NOLINES)
```

```
CXX HELLO (PPONLY(./aa.exp,NOLINES)
```

All `#line` and **#pragma** preprocessor directives (except for `margins` and `sequence` directives) remain. When you specify PPONLY(*), `#line` directives are generated to keep the line numbers generated for the output file from the preprocessor similar to the line numbers generated for the source file. All consecutive blank lines are suppressed.

If you specify the PPONLY option, the compiler turns on the TERMINAL option. If you specify the SHOWINC, XREF, AGGREGATE, or EXPMAC options with the PPONLY option, the compiler issues a warning, and ignores the options.

If you specify the PPONLY and LOCALE options, all the **#pragma filetag** directives in the source file are suppressed. The compiler generates its **#pragma filetag** directive at the first line in the preprocessed output file in the following format:

```
??=pragma filetag ("locale code page")
```

In this example, `??=` is a trigraph representation of the `#` character.

The code page in the pragma is the code set that is specified in the LOCALE option. For more information on locales, refer to *z/OS XL C/C++ Programming Guide*.

If you specify both PPONLY and NOPPONLY, the last one that is specified is used.

In the z/OS UNIX environment, the COMMENTS suboption can be requested by specifying the -C flag option. When using the c89 utility to invoke the compiler, the PPONLY compiler option cannot be specified. A combination of -E and -C flag options must be used instead. The c89 utility also provides the *prefix*_ELINES environment variable to control the LINES suboption (for further information on *prefix*_ELINES, refer to “Environment variables” on page 550). The output always goes to `stdout` when using the c89 utility because the PPONLY option can only be turned on by specifying the -E flag option. These limitations do not exist when using the xlc utility because the PPONLY option can be specified in addition to the -E, -P and -C flag options (for example, `-qpponly=foo.pp:comments:nolines:65`).

Note: **-Wc,PPONLY** syntax is not supported.

Predefined macros

None.

Related information

For more information on related compiler options, see:

- "TERMINAL | NOTERMINAL" on page 248
- "SHOWINC | NOSHOWINC" on page 223
- "XREF | NOXREF" on page 264
- "AGGREGATE | NOAGGREGATE (C only)" on page 64
- "EXPMAC | NOEXPMAC" on page 111

PREFETCH | NOPREFETCH

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Inserts prefetch instructions automatically where there are opportunities to improve code performance.

When PREFETCH is in effect, the compiler may insert prefetch instructions in compiled code. When NOPREFETCH is in effect, prefetch instructions are not inserted in compiled code.

Syntax

```
     ┌─PREFETCH───┐
►►───┴─NOPREFETCH─┴──────────────────────────────────────────────────────►◄
```

Defaults

PREFETCH

Usage

The compiler will attempt to generate prefetch instructions for ARCH(8) or above. The compiler will not issue a message if PREFETCH is active and the ARCH level is below 8.

Predefined macros

None.

PROLOG (C only)

Category

Object code control

Pragma equivalent

#pragma prolog (C only)

Purpose

Enables you to provide your own function entry code for all functions that have `extern` scope, or for all extern and static functions.

Syntax

```
►►──PROLOG──(──┬─"text-string"─────────────────────┬─)──────────────────────────►◄
               └─┬─EXTERN─┬──(──"text-string"──)─┘
                 └─ALL────┘
```

Defaults

The compiler generates default prolog code for the functions that do not have user-supplied prolog code.

Parameters

text-string

text-string is a C string, which must contain valid HLASM statements.

If the *text-string* consists of white-space characters only, or if the *text-string* is not provided, then the compiler ignores the option specification. If the *text-string* does not contain any white-space characters, then the compiler will insert leading spaces in front. Otherwise, the compiler will insert the *text-string* into the function prolog location of the generated assembler source. The compiler does not understand or validate the contents of the *text-string*. In order to satisfy the assembly step later, the given *text-string* must form valid HLASM code with the surrounding code generated by the compiler.

Note: Special characters like `newline` and `quote` are shell (or command line) meta characters, and may be preprocessed before reaching the compiler. It is advisable to avoid using them. The intended use of this option is to specify an assembler macro as the function prolog.

For information on valid HLASM statements, see **#pragma prolog**.

EXTERN

If the PROLOG option is specified with this suboption or without any suboption, the prolog applies to all functions that have external linkage in the compilation unit.

ALL

If the PROLOG option is specified with this suboption, the prolog also applies to static functions defined in the compilation unit.

Usage

For more information on METAL C default prolog code, see *z/OS Metal C Programming Guide and Reference*.

Notes:

1. The PROLOG option is only valid when the METAL option is specified.
2. When the PROLOG option is specified multiple times with the same suboption **all** or **extern**, only the function entry code of the last suboption specified will be displayed.
3. The PROLOG option with the suboption **all** overwrites the one with **extern** suboption, or the one without any suboption.

Predefined macros

None.

Related information

For more information on the METAL compiler option, see “METAL | NOMETAL (C only)” on page 182.

See “EPILOG (C only)” on page 106 for information on providing function exit code for system development.

REDIR | NOREDIR

Category

Object code control

Pragma equivalent

None.

Purpose

Allows redirection of `stderr`, `stdin`, and `stdout` from the command line.

Syntax

```
     ┌─RED───┐
►►───┴─NORED─┴──────────────────────────────────────────────►◄
```

Defaults

REDIR

Usage

When the REDIR compiler option is in effect, the compiler creates an object module that, when linked and run, allows you to redirect `stdin`, `stdout`, and `stderr` for your program from the command line when invoked from TSO or batch.

REDIR does not apply to programs invoked by the `exec` or `spawn` family of functions (in other words, redirection does not apply to programs invoked from the z/OS UNIX System Services shell).

IPA effects

If you specify the REDIR option for any compilation unit in the IPA Compile step, the compiler generates information for the IPA Link step. This option also affects the regular object module if you request one by specifying the IPA(OBJECT) option.

If you specify the REDIR option for the IPA Compile step, you do not need to specify it again on the IPA Link step. The IPA Link step uses the information generated for the compilation unit that contains the `main()` function, or for the first compilation unit it finds if it cannot find a compilation unit containing `main()`.

If you specify this option on both the IPA Compile and the IPA Link steps, the setting on the IPA Link step overrides the setting on the IPA Compile step. This situation occurs whether you use REDIR and NOREDIR as compiler options or specify them using the **#pragma runopts** directive (on the IPA Compile step).

Predefined macros

None.

RENT | NORENT (C only)

Category

Object code control

Pragma equivalent

#pragma options (rent) (C only), **#pragma options (norent)** (C only)

#pragma variable(rent), **#pragma variable(norent)**

Purpose

Generates reentrant code.

When the RENT compiler option is in effect, the compiler takes code that is not naturally reentrant and make it reentrant. Refer to *z/OS Language Environment Programming Guide* for a detailed description of reentrancy.

When the NORENT compiler option is in effect, the compiler does not generate reentrant code from non-reentrant code. Any naturally reentrant code remains reentrant.

Syntax

```
      ┌─NORENT─┐
►►────┴─RENT───┴──────────────────────────────────────────────────────►◄
```

Defaults

NORENT for C and RENT for C++.

For the z/OS UNIX System Services utilities, the default for a regular compile is RENT.

Usage

If you use the RENT option, the linkage editor cannot directly process the object module that is produced. You must use either the binder, which is described in Chapter 9, “Binding z/OS XL C/C++ programs,” on page 401, or the prelinker, which is described in Appendix A, “Prelinking and linking z/OS XL C/C++ programs,” on page 601.

Notes:

1. Whenever you specify the RENT compiler option, a comment that indicates its use is generated in your object module to aid you in diagnosing your program.
2. z/OS XL C++ code always uses constructed reentrancy so the RENT option is always in effect for C++. For C++, you cannot specify NORENT.
3. RENT variables reside in the modifiable writable static area for both z/OS XL C and z/OS XL C++ programs.
4. NORENT variables reside in the code area (which may be write protected) for z/OS XL C programs.

Note: The RENT compiler option has implications on how the binder processes objects. See *z/OS MVS Program Management: User’s Guide and Reference* for further information.

IPA effects
If you specify RENT or use **#pragma strings(readonly)** or **#pragma variable(rent | norent)** during the IPA Compile step, the information in the IPA object file reflects the state of each symbol.

If you specify the RENT option on the IPA Link step, it ignores the option. The reentrant/nonreentrant state of each symbol is maintained during IPA optimization and code generation. If any symbols within a partition are reentrant, the option section of the Partition Map displays the RENT compiler option.

If you generate an IPA Link listing by using the LIST or IPA(MAP) compiler option, the IPA Link step generates a Partition Map listing section for each partition. If any symbols within a partition are reentrant, the options section of the Partition Map displays the RENT compiler option.

Predefined macros
None.

Related information
For more information on related compiler options, see:
- “LIST | NOLIST” on page 163
- “IPA | NOIPA” on page 139

REPORT | NOREPORT

Category
Listings, messages, and compiler information

Pragma equivalent
None.

Purpose
Produces pseudo-C code listing files that show how sections of code have been optimized in both compile and link steps, when IPA is active. You can use this information to understand your application code and to tune your code for better performance.

Syntax

```
      ┌─NOREPORT─┐
►►────┴─REPORT───┴──────────────────────────────────────────────────────►◄
```

Defaults
NOREPORT

Usage
For REPORT to generate a pseudo-C code listing, you need to specify the LIST option. In addition, you must also specify one of the following options on the command line:
- **HOT**
- **IPA**

When SPLITLIST is specified, the pseudo-C listing will precede the pseudo-assembly listing in the same listing file.

The pseudo-C code listing is not intended to be compilable. Do not include any of the pseudo-C code in your program, and do not explicitly call any of the internal routines whose names may appear in the pseudo-C code listing.

Predefined macros

None.

Examples

The following example generates a pseudo-C code listing at IPA compile step:

```
xlc -qipa -qlist -qreport -c hello.c
```

The following example generates a pseudo-C code listing at IPA link step:

```
xlc -qipa -qlist -qreport -o hello.o
```

Related information

For more information on related compiler options, see:

- “LIST | NOLIST” on page 163
- “HOT | NOHOT” on page 127
- “IPA | NOIPA” on page 139

RESERVED_REG (C only)

Category

Object code control

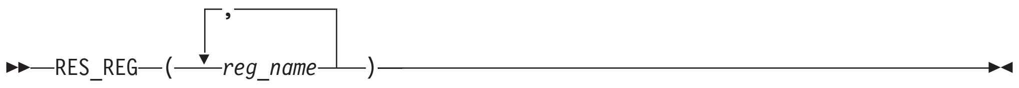
Pragma equivalent

None.

Purpose

Instructs the compiler not to use the specified general purpose register (GPR) during the compilation.

Syntax



Defaults

Not specified.

Parameters

reg_name
: Only the general purpose registers 0-15 (written as r0, r1, ..., r15 or R0, R1, ...,R15) can be specified for the RESERVED_REG option. Any other name is rejected with a warning message. Some general purpose registers have designated roles in the compiler for generating program code, and reserving these registers may prevent the compiler from generating the correct code. See Table 30 on page 212 for further information on z/OS general purpose registers that have designated roles for the XL C compiler.

Usage

A global register variable declaration reserves the register for the declared variable in the compilation unit where the declaration appears. The register is not reserved in other compilation units unless the global register declaration is placed in a common header file.

Notes:

1. Duplicate register names are ignored silently.
2. The RESERVED_REG option is cumulative, which means that, for example:

   ```
   -qreserved_reg=r14 -qreserved_reg=r15
   ```

   is equivalent to:

   ```
   -qreserved_reg=r14:r15
   ```

Table 30. General purpose registers that have designated roles for the z/OS XL C compiler

Register	Designated role
r0	volatile
r1	parameter list pointer
r3	designated by the compiler
r10	used by the C generated code for addressing data
r11	used by the C generated code for addressing data
r13	savearea pointer (C: stack pointer)
r14	function return address
r15	function entry point on entry, return code on exit. (C: integral type return value)

Predefined macros

None.

ROCONST | NOROCONST

Category

Object code control

Pragma equivalent

#pragma variable(*var_name***, NORENT)**

Purpose

Specifies the storage location for constant values.

When the ROCONST compiler option is in effect, the compiler places constants in read-only storage, even if the RENT option is in effect. Placing constant values in read-only memory can improve run-time performance, save storage, and provide shared access.

When the NOROCONST compiler option is in effect, constant values are placed in read/write storage.

Syntax

For C:

```
     ┌─NOROC─┐
►►───┴─ROC───┴──────────────────────────────────────────────────►◄
```

For C++:

```
     ┌─ROC───┐
►►───┴─NOROC─┴──────────────────────────────────────────────────►◄
```

Defaults

For C, the default option is NOROCONST. For C++, the default option is ROCONST.

Usage

The ROCONST option informs the compiler that the `const` qualifier is respected by the program. Variables defined with the `const` keyword will not be overridden by a casting operation.

Note that these `const` variables cannot be exported.

If the specification for a `const` variable in a **#pragma variable** directive is in conflict with the option, the **#pragma variable** takes precedence. The compiler issues an informational message.

If you set the ROCONST option, and if there is a **#pragma export** for a `const` variable, the pragma directive takes precedence. The compiler issues an informational message. The variable will still be exported and the variable will be reentrant.

IPA effects

If you specify the ROCONST option during the IPA Compile step, the information in the IPA object file reflects the state of each symbol.

If you specify the ROCONST option on the IPA Link step, it ignores the option. The reentrant or non-reentrant and `const` or non-`const` state of each symbol is maintained during IPA optimization and code generation.

The IPA Link step merges and optimizes your application code, and then divides it into sections for code generation. Each of these sections is a partition. The IPA Link step uses information from the IPA Compile step to determine if a subprogram can be placed in a particular partition. Only compatible subprograms are included in a given partition. Compatible subprograms have the same ROCONST setting.

The ROCONST setting for a partition is set to the specification of the first subprogram that is placed in the partition.

The option value that you specified for each IPA object file on the IPA Compile step appears in the IPA Link step Compiler Options Map listing section.

The RENT, ROCONST, and ROSTRING options all contribute to the re-entrant or non-reentrant state for each symbol.

The Partition Map sections of the IPA Link step listing and the object module END information section display the value of the ROCONST option.

Predefined macros

None.

Related information

For more information on related compiler options, see:

- “RENT | NORENT (C only)” on page 209
- “ROSTRING | NOROSTRING”

ROSTRING | NOROSTRING

Category

Object code control

Pragma equivalent

#pragma strings(readonly)

Purpose

Specifies the storage type for string literals.

When the ROSTRING compiler option is in effect, the compiler places string literals in read-only storage. Placing string literals in read-only memory can improve run-time performance and save storage.

When the NOROSTRING compiler option is in effect, string literals are placed in read/write storage.

Syntax

```
     ┌─ROS───┐
►►───┴─NOROS─┴──────────────────────────────────────────────►◄
```

Defaults

ROSTRING

IPA effects

If you specify the ROSTRING option during the IPA Compile step, the information in the IPA object file reflects the state of each symbol.

If you specify the ROSTRING option on the IPA Link step, it ignores the option. The reentrant or nonreentrant state of each symbol is maintained during IPA optimization and code generation.

The Partition Map section of the IPA Link step listing and the object module do not display information about the ROSTRING option for that partition. The RENT, ROCONST, and ROSTRING options all contribute to the reentrant or nonreentrant state for each symbol. If any symbols within a partition are reentrant, the option section of the Partition Map displays the RENT compiler option.

Predefined macros

None.

Related information

For more information on related compiler options, see:

- “RENT | NORENT (C only)” on page 209
- “ROCONST | NOROCONST” on page 212

ROUND

Category

Floating-point and integer control

Pragma equivalent

None.

Purpose

Specifies the rounding mode for the compiler to use when evaluating constant floating-point expressions at compile time.

Syntax

The syntax depends on whether the ROUND option is used with a base 2 IEEE-754 binary format (specified by the FLOAT(IEEE) compiler option), base 16 z/Architecture hexadecimal format (specified by the FLOAT(HEX) compiler option), or base 10 decimal floating-point format (specified by the DFP compiler option).

When FLOAT(IEEE) is specified:

```
             ┌─N─┐
►►──ROUND──(─┼─M─┼──)──────────────────────────────────────────────────►◄
             ├─P─┤
             └─Z─┘
```

When FLOAT(HEX) is specified:

```
►►──ROUND──(──Z──)─────────────────────────────────────────────────────►◄
```

When DFP is specified:

```
             ┌─DN──┐
►►──ROUND──(─┼─DI──┼──)────────────────────────────────────────────────►◄
             ├─DM──┤
             ├─DNA─┤
             ├─DNZ─┤
             ├─DP──┤
             └─DZ──┘
```

Defaults

- For FLOAT(IEEE), the default option is ROUND(N).
- For FLOAT(HEX), the default option is ROUND(Z).
- For DFP, the default is ROUND(DN).

Parameters

The rounding *mode* depends on whether the ROUND option is used with the DFP compiler option.

If FLOAT(IEEE) is in effect but DFP is not in effect, the following modes are valid:

N round to the nearest representable number (ties to even)

Note: A *tie* occurs when the number to be rounded is at the exact midpoint between two values towards which it can be rounded. For example, if we are rounding to the nearest representable whole number, and we are given the value 1.5, we are at the exact midpoint between the two nearest whole numbers (2 and 1). This is considered a tie. In this example, and using ties to even, we would round the value 1.5 to the value 2, as 2 is an even number.

M round towards minus infinity

P round towards positive infinity

Z round towards zero

Note: ROUND() is the same as ROUND(N).

If the DFP compiler option is in effect, the following modes are valid:

DI round towards infinity (away from zero)

DM
round towards minus infinity

DN
round to the nearest representable number (ties to even)

DNA
round to the nearest representable number (ties away from zero)

Note: The value will round to the nearest representable number, but when there is a tie, it will round towards the larger magnitude (or away from zero).

DNZ
round to the nearest representable number (ties towards zero)

Note: The value will round to the nearest representable number, but when there is a tie, it will round towards the smaller magnitude (or towards zero).

DP
round towards positive infinity

DZ
round towards zero

Usage

You can specify a rounding *mode* only when you use IEEE floating-point mode. In hexadecimal mode, the rounding is always towards zero.

You must ensure that you are in the same rounding mode at compile time (specified by the ROUND(*mode*) option), as at run time. Entire compilation units will be compiled with the same rounding mode throughout the compilation. For further information on the DFP header files and functions, see *z/OS XL C/C++ Run-Time Library Reference*. If you switch run-time rounding modes inside a function, your results may vary depending upon the optimization level used and other characteristics of your code; use caution if you switch rounding mode inside functions.

If you specify ROUND(*mode*) in hexadecimal floating-point mode, where mode is not Z, the compiler ignores ROUND(*mode*) and issues a warning.

IPA effects

The IPA Compile step generates information for the IPA Link step. The ROUND option also affects the regular object module if you request one by specifying the IPA(OBJECT) option.

The IPA Link step merges and optimizes the application code, and then divides it into sections for code generation. Each of these section is a partition. The IPA Link step uses information from the IPA Compile step to ensure that an object is included in a compatible partition. Refer to the “FLOAT” on page 114 for further information.

Predefined macros

None.

Related information

For information on related compiler options, see:

- “FLOAT” on page 114
- “DFP | NODFP” on page 100

RTCHECK | NORTCHECK

Category

Error checking and debugging

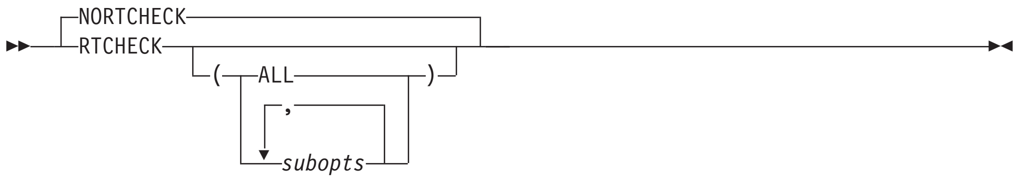
Pragma equivalent

None.

Purpose

Generates compare-and-trap instructions which perform certain types of runtime checking. The messages can help you to debug your C and C++ programs.

Syntax



Defaults

NORTCHECK

Parameters

suboption is one of the suboptions that are shown in Table 31 on page 218.

The following table lists the RTCHECK suboptions and the messages they generate.

Note: Default RTCHECK suboptions are underlined.

Table 31. RTCHECK suboptions and descriptions

RTCHECK Suboption	Description
ALL	Automatically generates compare-and-trap instructions for all possible runtime checks. This suboption is equivalent to **RTCHECK**.
BOUNDS | NOBOUNDS	Performs runtime checking of addresses when subscripting within an object of known size.
DIVZERO | NODIVZERO	Performs runtime checking of integer division. A trap will occur if an attempt is made to divide by zero.
NULLPTR | NONULLPTR	Performs runtime checking of addresses contained in pointer variables used to reference storage.

Usage

You can specify the RTCHECK option more than once. The suboption settings are accumulated, but the later suboptions override the earlier ones.

You can use the **all** suboption along with the **no...** form of one or more of the other options as a filter. For example, using:

```
xlc -qrtcheck=all:nonullptr
```

provides checking for everything except for addresses contained in pointer variables used to reference storage. If you use **all** with the **no...** form of the suboptions, **all** should be the first suboption.

Note: The RTCHECK option is only valid for architecture level 8 or above, and for Language Environment V1.10 and up.

Predefined macros

None.

RTTI | NORTTI (C++ only)

Category

Object code control

Pragma equivalent

None.

Purpose

Generates runtime type identification (RTTI) information for exception handling and for use by the `typeid` and `dynamic_cast` operators.

Syntax

```
      ┌─NORTTI─┐
►►────┴─RTTI───┴──┬──────────────────────────┬──────────────────────────►◄
                  └─(──┬─ALL─────────┬──)────┘
                       └─DYNAMICCAST─┘
```

Defaults
NORTTI

Parameters
ALL
: The compiler generates the information needed for the RTTI `typeid` and `dynamic_cast` operators. If you specify just RTTI, this is the default suboption.

DYNAMICCAST
: The compiler generates the information needed for the RTTI `dynamic_cast` operator, but the information needed for `typeid` operator is not generated.

Usage
For improved runtime performance, suppress RTTI information generation with the NORTTI setting.

Note: Even though the default is NORTTI, if you specify LANGLVL(EXTENDED) or LANGLVL(ANSI), you will also implicitly select RTTI.

IPA effects
The IPA Link step does not accept the RTTI option. The compiler issues a warning message if you specify this option in the IPA Link step.

Predefined macros
- __RTTI_DYNAMIC_CAST__ is predefined to a value of 1 when the RTTI, RTTI(ALL), or RTTI(DYNAMICCAST) compiler options are in effect; otherwise, it is not defined.
- __RTTI_ALL__ is predefined to a value of 1 when the RTTI or RTTI(ALL) compiler options are in effect; otherwise, it is not defined.
- __NO_RTTI__ is predefined to a value of 1 when the NORTTI compiler option is in effect; otherwise, it is not defined.

Related information
For more information on the LANGLVL(EXTENDED) compiler option, see "LANGLVL" on page 147.

SEARCH | NOSEARCH

Category
Compiler input

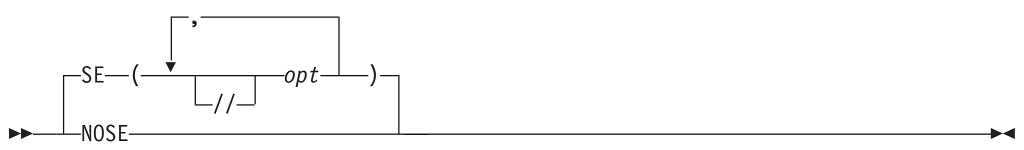
Pragma equivalent
None.

Purpose
Specifies the directories or data sets to be searched for system include files.

When the SEARCH compiler option is in effect, the preprocessor looks for system include files in the specified directories or data sets. System include files are those files that are associated with the `#include <filename>` form of the `#include` preprocessor directive. See "Using include files" on page 360 for a description of the `#include` preprocessor directive.

When the NOSEARCH compiler option is in effect, the preprocessor searches only those data sets that are specified in the SYSLIB statement.

Syntax



Defaults

For C++, the default option is SE(//'CEE.SCEEH.+', //'CBC.SCLBH.+'). For C, the default option is SE(//'CEE.SCEEH.+').

Note: The c99, c89, cc, and c++ utilities explicitly specify this option in the z/OS UNIX System Services shell. The suboptions are determined by the following:

- Additional include search directories identified by the **c89 -I** options. Refer to Chapter 21, "c89 — Compiler invocation using host environment variables," on page 535 for more information.
- z/OS UNIX System Services environment variable settings: *prefix*_INCDIRS, *prefix*_INCLIBS, and *prefix*_CSYSLIB. They are normally set during compiler installation to reflect the compiler and run-time include libraries. Refer to "Environment variables" on page 550 for more information.

This option is specified as NOSEARCH, SEARCH by the c89 utility, so it resets the SEARCH parameters you specify. While the c89 utility forces NOSEARCH so that any defaults that are set by the customizable defaults module CCNEDFLT are cleared, the xlc utility relies on the entry in the configuration file for that purpose. If you do not specify **-qnosearch** in the configuration file, **xlc** will append the search libraries specified via the -I flags to the libraries set by the CCNEDFLT customizable defaults module. This essentially allows xlc users to take advantage of the customization module, which is not the case with the c89 utility.

Parameters

The suboptions for the SEARCH option are identical to those for the LSEARCH option. For information on the LSEARCH option, see "LSEARCH | NOLSEARCH" on page 171.

Usage

The SYSLIB ddname is considered the last suboption for SEARCH, so that specifying SEARCH `(X)` is equivalent to specifying SEARCH`(X,DD:SYSLIB)`.

Any NOSEARCH option cancels all previous SEARCH specifications, and any new SEARCH options that follow it are used. When more than one SEARCH compile option is specified, all directories or data sets in the SEARCH options are used to find the system include files.

Notes:

1. SEARCH allows the compiler to distinguish between header files that have the same name but reside in different data sets. If NOSEARCH is in effect, the compiler searches for header files only in the data sets concatenated under the SYSLIB DD statement. As the compiler includes the header files, it uses the first file it finds, which may not be the correct one. Thus the build may encounter unpredictable errors in the subsequent link-edit or bind, or may result in a malfunctioning application.

2. If the *filename* in the `#include` directive is in absolute form, searching is not performed. See “Determining whether the file name is in absolute form” on page 365 for more details on absolute `#include` *filename*.

IPA effects

The SEARCH option is used for source code searching, and has the same effect on an IPA Compile step as it does on a regular compilation.

The IPA Link step accepts the SEARCH option, but ignores it.

Predefined macros

None.

Related information

For further information on library search sequences, see “Search sequences for include files” on page 369.

SEQUENCE | NOSEQUENCE

Category

Compiler input

Pragma equivalent

#pragma sequence, **#pragma nosequence**

Purpose

Specifies the columns used for sequence numbers.

Syntax

For C++ (fixed record format, variable record format, and the z/OS UNIX System Services file system):

```
   ┌─NOSEQ──────────────┐
►►─┴─SEQ──┬─────────┬───┴────────────────────────────────────────────►◄
          └─(m,n)───┘
```

For C (fixed record format, variable record format, and the z/OS UNIX file system):

```
   ┌─SEQ──(──m,n──)─┐
►►─┴─NOSEQ──────────┴────────────────────────────────────────────────►◄
```

Defaults

- For C++ fixed record format, variable record format, and the z/OS UNIX file system, the default is NOSEQUENCE.
- For C variable record format and the z/OS UNIX file system, the default is NOSEQUENCE.
- For C fixed record format, the default is SEQUENCE.
- The default values for C++ SEQUENCE are columns 73 to 80.

Parameters

m Specifies the column number of the left-hand margin. The value of *m* must be greater than `0` and less than `32767`.

n Specifies the column number of the right-hand margin. The value of *n* must be greater than *m* and less than 32767. An asterisk (*) can be assigned to *n* to indicate the last column of the input record. Thus, SEQUENCE (74,*) shows that sequence numbers are between column 74 and the end of the input record.

Usage

When the SEQUENCE compiler option is in effect, it defines the section of the input record that is to contain sequence numbers. No attempt is made to sort the input lines or records into the specified sequence or to report records out of sequence.

You can use the MARGINS and SEQUENCE options together. The MARGINS option is applied first to determine which columns are to be scanned. The SEQUENCE option is then applied to determine which of these columns are not to be scanned. If the SEQUENCE settings do not fall within the MARGINS settings, the SEQUENCE option has no effect.

Note: If your program uses the `#include` preprocessor directive to include z/OS XL C library header files *and* you want to use the SEQUENCE option, you must ensure that the specifications on the SEQUENCE option do not include any columns from 20 through 50. That is, both *m* and *n* must be less than 20, or both must be greater than 50. If your program does not include any z/OS XL C/C++ library header files, you can specify any setting you want on the SEQUENCE option when the setting is consistent with your own include files.

Predefined macros

None.

Related information

For further information on the MARGINS compiler option, see “MARGINS | NOMARGINS” on page 178.

SERVICE | NOSERVICE

Category

Error checking and debugging

Pragma equivalent

#pragma options(service) (C only), **#pragma options(noservice)** (C only)

Purpose

Places a *string* in the object module, which is displayed in the traceback if the application fails abnormally.

Syntax

```
     ┌─NOSERV─────────────────┐
►►───┴─SERV──(──string──)─────┴──────────────────────────────────────────►◄
```

Defaults

NOSERVICE

Parameters

string
  User-specified string of characters.

Usage

When the SERVICE compiler option is in effect, the *string* in the object module is loaded into memory when the program is executing. If the application fails abnormally, the *string* is displayed in the traceback.

For z/OS XL C, if the SERVICE option is specified both on a **#pragma options** directive and on the command line, the option that is specified on the command line will be used.

You must enclose your *string* within opening and closing parentheses. You do not need to include the *string* in quotation marks.

The following restrictions apply to the *string* specified:

- The *string* cannot exceed 64 characters in length. If it does, excess characters are removed, and the *string* is truncated to 64 characters. Leading and trailing blanks are also truncated.

  Note: Leading and trailing spaces are removed first and then the excess characters are truncated.
- All quotation marks that are specified in the *string* are removed.
- All characters, including DBCS characters, are valid as part of the *string* provided they are within the opening and closing parentheses.
- Parentheses that are specified as part of the *string* must be balanced. That is, for each opening parentheses, there must be a closing one. The parentheses must match after truncation.
- When using the **#pragma options** directive (C only), the text is converted according to the locale in effect.
- Only characters which belong to the invariant character set should be used, to ensure that the signature within the object module remains readable across locales.

IPA effects

If you specify the SERVICE option on the IPA Compile step, or specify **#pragma options(service)** in your code, it has no effect on the IPA Link step. Only the SERVICE option you specify on the IPA Link step affects the generation of the service string for that step.

Predefined macros

None.

SHOWINC | NOSHOWINC

Category

Listings, messages and compiler information

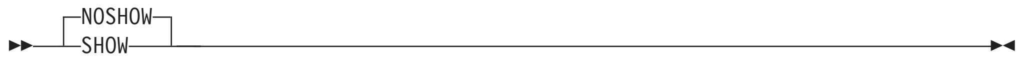
Pragma equivalent

None.

Purpose

When used with SOURCE option to generate a listing file, selectively shows user and system header files in the source and Pseudo-Assembly sections of the listing file.

Syntax



Defaults

NOSHOWINC

In the z/OS UNIX System Service environment, this option is turned on by specifying -V when using the c89 utility.

Usage

In the listing, the compiler replaces all `#include` preprocessor directives with the source that is contained in the include file.

The SHOWINC option has effect only if the SOURCE option is also in effect.

Predefined macros

None.

Related information

For more information on the SOURCE compiler option, see "SOURCE | NOSOURCE" on page 226.

SHOWMACROS | NOSHOWMACROS

Category

Compiler output

Pragma equivalent

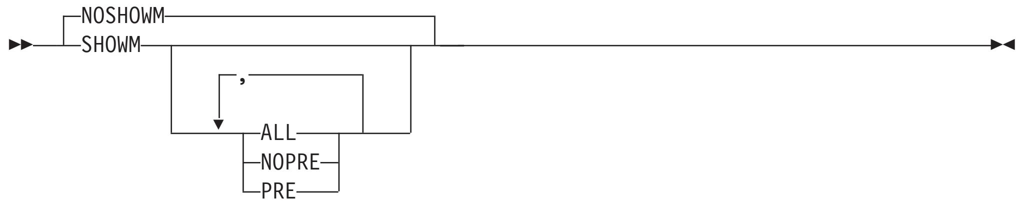
None.

Purpose

Displays macro definitions to preprocessed output.

Displaying macros to preprocessed output can help to determine the available functionality in the compiler. The macro listing may prove useful in debugging complex macro expansions.

Syntax



Defaults

NOSHOWMACROS

The SHOWMACROS option replaces the preprocessed output with the macro define directives.

Parameters

ALL
Emits all macro definitions to preprocessed output. This is the same as specifying SHOWMACROS.

PRE
Emits only predefined macro definitions to preprocessed output. This suboption has no impact on user macros.

NOPRE
Suppresses appending predefined macro definitions to preprocessed output.

Specifying **showmacros** with no suboptions is equivalent to **-qshowmacros=all**.

Usage

Note the following when using this option:

- This option has no effect unless preprocessed output is generated; for example, using the **-qpponly** option in the xlc utility, or using the **PPONLY** option through JCL and TSO.
- If a macro is defined and subsequently undefined before compilation ends, this macro will not be included in the preprocessed output.
- Only macros defined internally by the preprocessor are considered predefined; all other macros are considered as user-defined.

Predefined macros

None.

SKIPSRC

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

When a listing file is generated using the SOURCE option, SKIPSRC option can be used to determine whether the source statements skipped by the compiler are shown in the source section of the listing file.

Syntax

```
                ┌─SHOW─┐
►►──SKIPS──(────┴─HIDE─┴───)──────────────────────────────────────────►◄
```

Defaults

SKIPSRC(SHOW)

Parameters

SHOW
Shows all source statements in the listing.

HIDE
Hides the source statements skipped by the compiler. This improves the readability of the listing file.

Usage

The SKIPSRC option has effect only if the SOURCE option is also in effect. For information on the SOURCE options, see "SOURCE | NOSOURCE."

Predefined macros

None.

SOURCE | NOSOURCE

Category

Listings, messages and compiler information

Pragma equivalent

None.

Purpose

Produces a compiler listing file that includes the source section of the listing.

Syntax

```
   ┌─NOSO─┐
►►─┴─SO───┴─┬──────────────────────────────────────────────────────┬──────────►◄
            └─(──┬─Sequential filename─────────────────────┬─)─┘
                 ├─Partitioned data set────────────────────┤
                 ├─Partitioned data set (member)───────────┤
                 ├─z/OS UNIX System Services filename──────┤
                 └─z/OS UNIX System Services directory─────┘
```

Defaults

NOSOURCE

For the z/OS UNIX System Services utilities, the default for a regular compile is NOSOURCE(/dev/fd1).

In the z/OS UNIX System Services environment, this option is turned on by specifying -V when using the **c89**, **cc** or **c++** commands.

Parameters

Sequential filename
: Specifies the sequential data set file name for the compiler listing.

Partitioned data set
: Specifies the partitioned data set for the compiler listing.

Partitioned data set (member)
: Specifies the partitioned data set (member) for the compiler listing.

z/OS UNIX System Services filename
: Specifies the z/OS UNIX System Services file name for the compiler listing.

z/OS UNIX System Services directory
: Specifies the z/OS UNIX System Services directory for the compiler listing.

Usage

If you specify SOURCE(*filename*), the compiler places the listing in the file that you specified. If you do not specify a file name for the SOURCE option, the compiler uses the SYSCPRT ddname if you allocated one. Otherwise, the compiler constructs the file name as follows:

- If you are compiling a data set, the compiler uses the source file name to form the name of the listing data set. The high-level qualifier is replaced with the userid under which the compiler is running, and `.LIST` is appended as the low-level qualifier.
- If the source file is a z/OS UNIX file, the listing is written to a file that has the name of the source file with a `.lst` extension in the current working directory.

The NOSOURCE option can optionally take a file name suboption. This file name then becomes the default. If you subsequently use the SOURCE option without a file name suboption, the compiler uses the file name that you specified in the earlier NOSOURCE.

Example: The following specifications have the same result:

```
CXX HELLO (NOSO(./hello.lis) SO
```

```
CXX HELLO (SO(./hello.lis)
```

If you specify SOURCE and NOSOURCE multiple times, the compiler uses the last specified option with the last specified suboption. For example, the following specifications have the same result:

```
CXX HELLO (NOSO(./hello.lis) SO(./n1.lis)  NOSO(./test.lis) SO
```

```
CXX HELLO (SO(./test.lis)
```

Notes:

1. If you specify data set names with the SOURCE, LIST, or INLRPT option, the compiler combines all the listing sections into the last data set name specified.
2. If you use the following form of the command in a JES3 batch environment where xxx is an unallocated data set, you may get undefined results.
   ```
   SOURCE(xxx)
   ```

Predefined macros

None.

Related information

For more information on related compiler options, see:

- "LIST | NOLIST" on page 163
- "INLRPT | NOINLRPT" on page 137

SPILL | NOSPILL

Category

Compiler customization

Pragma equivalent

#pragma options (spill) (C only), **#pragma options (nospill)** (C only)

#pragma option_override(subprogram_name, ″OPT(SPILL,size)″)

Purpose

Specifies the size (in bytes) of the register spill space, the internal program storage areas used by the optimizer for register spills to storage.

When the SPILL compiler option is in effect, you can specify the size of the spill area to be used for the compilation.

When the NOSPILL compiler option is in effect, the compiler defaults to SPILL(128).

Syntax

```
       ┌─SP──┬─────────────┬─┐
       │     └─(──size──)──┘ │
►►─────┴─NOSP────────────────┴──────────────────────────────────►◄
```

Defaults

For compiles with LP64 specified, the default for the SPILL compiler option is SPILL(256). For compiles with ILP32 specified, the default for the SPILL compiler option remains as SPILL(128).

Parameters

size

An integer representing the number of bytes for the register allocation spill area.

Usage

When too many registers are in use at once, the compiler saves the contents of some registers in temporary storage, called the spill area.

If your program is very complex, or if there are too many computations to hold in registers at one time and your program needs temporary storage, you might need to increase this area. Do not enlarge the spill area unless the compiler issues a message requesting a larger spill area. In case of a conflict, the largest spill area specified is used.

The maximum spill area size is `1073741823` bytes or 2^{30}–1 bytes. Typically, you will only need to specify this option when compiling very large programs with OPTIMIZE.

Note: There is an upper limit for the combined area for your spill area, local variables, and arguments passed to called functions at OPT. For best use of the stack, do not pass large arguments, such as structures, by value.

IPA effects

If you specify the SPILL option for any compilation unit in the IPA Compile step, the compiler generates information for the IPA Link step. This option also affects the regular object module if you request one by specifying the IPA(OBJECT)

If you specify the SPILL option for the IPA Link step, the compiler sets the Compilation Unit values of the SPILL option that you specify. The IPA Link step Prolog listing section will display the value of this option.

If you do not specify the SPILL option in the IPA Link step, the setting from the IPA Compile step for each Compilation Unit will be used.

In either case, subprogram-specific SPILL options will be retained.

The IPA Link step merges and optimizes your application code, and then divides it into sections for code generation. Each of these sections is a partition. The IPA Link step uses information from the IPA Compile step to determine if a subprogram can be placed in a particular partition.

The initial overall SPILL value for a compilation unit is set to the IPA Link SPILL option value, if specified. Otherwise, it is the SPILL option that you specified during the IPA Compile step for the compilation unit.

The SPILL value for each subprogram in a partition is determined as follows:

- The SPILL value is set to the compilation unit SPILL value, unless a subprogram-specific SPILL option is present.
- During inlining, the caller subprogram SPILL value will be set to the maximum of the caller and callee SPILL values.

The overall SPILL value for a partition is set to the maximum SPILL value of any subprogram contained within that partition.

The option value that you specified for each IPA object file on the IPA Compile step appears in the IPA Link step Compiler Options Map listing section.

The Partition Map sections of the IPA Link step listing and the object module END information section display the value of the SPILL option. The Partition Map also displays any subprogram-specific SPILL values.

Predefined macros

None.

SPLITLIST | NOSPLITLIST

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Enables the z/OS XL C/C++ compiler to write the IPA Link phase listing to multiple PDS members, PDSE members, or z/OS UNIX System Services files. The SPLITLIST compiler option has no effect unless the LIST or INLRPT compiler options are also specified.

Syntax

```
     ┌─NOSPL─┐
►►───┴─SPL───┴──────────────────────────────────────────────────────────►◄
```

Defaults

NOSPLITLIST

Usage

Normally, the default listing location is `stdout` or SYSCPRT. You can instruct the compiler to output listing contents into a file by using the LIST or INLRPT options. This method can be useful when the source file is large and there is a large amount of detail in the listing. Writing the listing contents to a file, allows you to use an editor or a search utility to browse through the file. However, for the IPA Link phase, which processes the whole application instead of just one source file, there are situations when the listing file itself becomes too large, which can cause difficulties for an editor or search utility. The SPLITLIST option is designed to split a listing into multiple files so that it will be easier for you to browse and edit large listings.

The SPLITLIST option is used only in the IPA Link phase, and the location of the files, which must be a PDS, PDSE, or z/OS UNIX file system directory, must be specified by the LIST or INLRPT option. If the LIST or INLRPT option is not used to specify a location, you will receive an error message.

Table 32 shows the names given to the generated listing sections if a z/OS UNIX file system directory name is specified. In the table, we assume the location is a directory called `listing`, and there are three partitions generated by the IPA Link phase.

Table 32. Listing section names comparison for a specified z/OS UNIX file system directory

Listing section names generated with SPLITLIST	Listing section names generated with NOSPLITLIST
listing/part0	Partition 0 listing
listing/part1	Partition 1 listing
listing/part2	Partition 2 listing
listing/objmap	Object File Map
listing/srcmap	Source File Map
listing/inlrpt	Inline Report
listing/options	IPA Link Options
listing/cuopts	Compiler Options Map
listing/globsym	Global Symbols Map
listing/messages	Messages and Summary

Table 33 shows the names given to the generated listing sections if a PDS or PDSE name is specified. In the table, we assume the PDS or PDSE name is `ACCNTING.LISTING`, and that three partitions are generated by the IPA Link phase.

Table 33. Listing section names comparison for a specified PDS name

Listing section names generated with SPLITLIST	Listing section names generated with NOSPLITLIST
ACCNTING.LISTING(PART0)	Partition 0 listing
ACCNTING.LISTING(PART1)	Partition 1 listing
ACCNTING.LISTING(PART2)	Partition 2 listing
ACCNTING.LISTING(OBJMAP)	Object File Map
ACCNTING.LISTING(SRCMAP)	Source File Map
ACCNTING.LISTING(INLRPT)	Inline Report
ACCNTING.LISTING(OPTIONS)	IPA Link Options
ACCNTING.LISTING(CUOPTS)	Compiler Options Map
ACCNTING.LISTING(GLOBSYM)	Global Symbols Map
ACCNTING.LISTING(MESSAGES)	Messages and Summary

Notes:

1. The SPLITLIST option can only be specified in the IPA Link phase.
2. Repeating a SPLITLIST option is equivalent to specifying it once. The last one specified is the effective setting.
3. If the SPLITLIST option is specified but the effective location of the listing is not a z/OS UNIX file system directory, PDS data set, or PDSE data set, then a diagnostic message will be issued and the IPA Link phase return code will be at least 8.
4. A z/OS UNIX file system directory name must denote a z/OS UNIX directory which exists and is accessible by the user prior to the IPA Link. Otherwise, a diagnostic message will be issued and the minimum return code will be raised to 16.
5. The PDS name must denote a PDS or PDSE data set which exists and is accessible by the user prior to the IPA Link. Otherwise, a diagnostic message will be generated and the minimum return code will be raised to 16.

IPA effects

The SPLITLIST option will be ignored by the IPA Compile phase (since it does not generate a listing). If **-Wc,SPLITLIST** is used, the IPA Compile step will ignore it.

Predefined macros

None.

Examples

The following examples show how to use SPLITLIST:

Example 1:

```
# list must exist prior to executing the IPA link
#
mkdir list

# Generate listing sections corresponding to XREF and LIST
#
c89 -Wl,I,"XREF,LIST(./list)" -Wl,I,SPLITLIST -o a.out hello.o
```

Example 2:

```
# list must exist prior to executing the IPA link
#
mkdir list

# Since NOLIST is specified, only IPA(MAP)  sections are generated
#  However, the destination directory is the one specified in the NOLIST option
#
c89 -Wl,I,SPLITLIST -Wl,I,'NOLIST(./list)' -WI,MAP -o a.out hello.o
```

Example 3:

```
# list must exist prior to executing the IPA link
#
mkdir list

# Generate sections corresponding to INLRPT
#
c89 -Wl,I,"INLR(./list)" -Wl,I,SPLITLIST -o a.out hello.o
```

The following provides a JCL example for SPLITLIST:

```
//USRID1A  JOB (359B,2326),'USRID1',
//          MSGLEVEL=(1,1),MSGCLASS=S,CLASS=A,NOTIFY=USRID1
/*JOBPARM   T=1,L=300
```

```
//ORDER     JCLLIB ORDER=(CBC.SCCNPRC)
//*-------------------------------------------------------------------
//* Compile
//*-------------------------------------------------------------------
//C0011L01  EXEC EDCC,
//          OUTFILE='USRID1.PASS1.OBJECT(SPLLIST),DISP=SHR',
// PARM.COMPILE=('IPA(NOLINK,NOOBJECT) OPT',
//   'RENT LO ')
//SYSIN    DD *,DLM='/>'
int main()
{
    return 0;
}
/>
//*-------------------------------------------------------------------
//* IPA LINK
//*-------------------------------------------------------------------
//C0011L02  EXEC EDCI,
//          OUTFILE='USRID1.PASS2.OBJECT(SPLLIST),DISP=SHR',
// PARM.COMPILE=('LIST(USRID1.LISTPDS) IPA(LINK,MAP) OPT',
//   'RENT LO SPLITLIST')
//OBJECT  DD  DSN=USRID1.PASS1.OBJECT,DISP=SHR
//SYSIN    DD *,DLM='/>'
  INCLUDE OBJECT(SPLLIST)
/>
//
```

Related information

For more information on related compiler options, see:

- "LIST | NOLIST" on page 163
- "INLRPT | NOINLRPT" on page 137

SQL | NOSQL

Category

Language element control

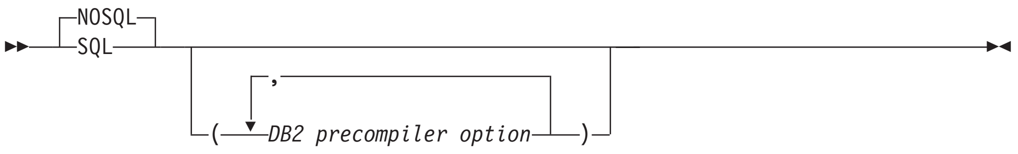
Pragma equivalent

None.

Purpose

Enables the compiler to process embedded SQL statements.

Syntax



Defaults

NOSQL

Parameters

DB2 precompiler option
: The SQL coprocessor options are only passed to the SQL statement coprocessor; the z/OS XL C/C++ compiler does not act on any of the options.

Refer to *DB2 for z/OS Application Programming and SQL Guide* at http://www.ibm.com/software/data/db2/zos/library.html for details.

Usage

You may use this option to compile C and C++ programs containing embedded SQL statements, that have not been pre-compiled by the DB2 Precompiler. When you specify this option, the compiler writes the database request module (DBRM Bind file) to the ddname DBRMLIB. This option is not supported under AMODE 64 (LP64 compiler option).

Note: To use this option, the z/OS XL C/C++ compiler requires access to DB2 Version 7 or later. Ensure you specify the DB2 load module data set in your compile step STEPLIB.

To use this option with the supplied proc, specify the required items in your JCL, as in the following example:

```
//SQLCOMP EXEC EDCC,
// CPARM='SQL',
// INFILE=PAYROLL.SOURCE(MASTER)'
//STEPLIB DD DSN=CEE.SCEERUN,DISP=SHR
// DD DSN=CEE.SCEERUN2,DISP=SHR
// DD DSN=CBC.SCCNCMP,DISP=SHR
// DD DSN=hlq.SDSNLOAD,DISP=SHR
//DBRMLIB DD DSN=PAYROLL.DBRMLIB.DATA(MASTER),DISP=SHR
```

where `hlq.SDSNLOAD` is a generic data set name.

An SQL INCLUDE statement is treated the same as an `#include` directive. The following two lines are processed the same way by the compiler:

```
EXEC SQL INCLUDE name;
#include "name"
```

The library search order for SQL INCLUDE statements is the same as specified in the LSEARCH option or the USERLIB ddname. Nested SQL INCLUDE statements, that are not supported with the DB2 Precompiler, are supported by the SQL compiler option.

For C++, host variable names do not need to be unique, as they are previously required to be by the DB2 Precompiler. You may declare host variables, using the SQL BEGIN DECLARE SECTION and SQL END DECLARE SECTION statements, of the same name but in different lexical scopes.

Example: The same lexical scoping rules for C/C++ variables apply when they are used as host variables in SQL statements:

```
EXEC SQL BEGIN DECLARE SECTION;
int salary;
EXEC SQL END DECLARE SECTION;

main() {
  EXEC SQL BEGIN DECLARE SECTION; /* (1) */
  int salary;
  EXEC SQL END DECLARE SECTION;    /* (2) */

  /* The local variable salary will be used here */
  EXEC SQL SELECT SALARY INTO :salary FROM ctab WHERE EMPNO = 12345;
}
```

If the local variable has not been declared as host variable, that is, the SQL BEGIN DECLARE SECTION statement (1) and SQL END DECLARE SECTION statement (2) are missing, you will get a compiler error.

When you specify the DFP and SQL compiler options with the XL C/C++ compiler, decimal floating-point typed identifiers can be designated as host variables and used in embedded SQL statements. This will allow you to write applications with embedded SQL statements for DB2 databases containing decimal floating-point data. SQL for DB2 V9 provides support for decimal floating-point types through the DECFLOAT data type. For further information on the DFP type host variable, see the description for the DECFLOAT scalar function in the *DB2 Version 9.1 for z/OS SQL Reference* at http://www.ibm.com/software/data/db2/zos/library.html. For more information on the DFP compiler option, see “DFP | NODFP” on page 100.

Predefined macros

__SQL__ is predefined to 1 when the SQL compiler option is in effect; otherwise it is undefined.

The following macros are supported when the SQL compiler option is in effect in order to assist with portability of embedded SQL source code and with initializing SQL variables:

- SQL_VARBINARY_INIT
- SQL_BLOB_INIT
- SQL_CLOB_INIT
- SQL_DBCLOB_INIT

These macros will behave as if they were user-defined macros with the following definitions:

```
#define SQL_VARBINARY_INIT(s) {sizeof(s)-1, s}
#define SQL_BLOB_INIT(s) {sizeof(s)-1, s}
#define SQL_CLOB_INIT(s) {sizeof(s)-1, s}
#define SQL_DBCLOB_INIT(s) {(sizeof(s)/2)-1, s} (31-bit mode)
#define SQL_DBCLOB_INIT(s) {(sizeof(s)/4)-1, s} (64-bit mode)
```

Refer to *DB2 Version 9.1 for z/OS SQL Reference* at http://www.ibm.com/software/data/db2/zos/library.html for further information on the VARBINARY, BLOB, CLOB, and DBCLOB functions that are related to these macros.

SSCOMM | NOSSCOMM (C only)

Category

Language element control

Pragma equivalent

None.

Purpose

Allows comments to be specified by two slashes (//), which supports C++ style comments in C code.

When the SSCOMM option is in effect, it instructs the C compiler to recognize two slashes (//) as the beginning of a comment, which terminates at the end of the line. It will continue to recognize /* */ as comments.

When the NOSSCOMM compiler option is in effect, /* */ is the only valid comment format.

Syntax

```
     ┌─NOSS─┐
►►───┴─SS───┴────────────────────────────────────────────────────────►◄
```

Defaults

NOSSCOMM

For LANGLVL(STDC99) and LANGLVL(EXTC99), the default is SSCOMM.

Usage

C++ Note: You can include the same delimiter in your JCL for C++ source code, however you do not need to use the SSCOMM option.

When using the **xlc** command in z/OS UNIX System Services, the equivalent option for SSCOMM is **-qcpluscmt**.

Predefined macros

None.

Examples

If you include your z/OS XL C program in your JCL stream, be sure to change the delimiters so that your comments are recognized as z/OS XL C comments and not as JCL statements:

```
//COMPILE.SYSIN DD DATA,DLM=@@
#include <stdio.h>
void main(){
// z/OS XL C comment
printf("hello world\n");
// A nested z/OS XL C /*  */ comment
}
@@
//* JCL comment
```

START | NOSTART

Category

Object code control

Pragma equivalent

#pragma options(start) (C only), **#pragma options(nostart)** (C only)

Purpose

Generates a CEESTART, which is an object that controls initialization at execution, when necessary.

When the START option is in effect, a comment that indicates its use will be generated in your object module to aid you in diagnosing your program.

When the NOSTART compiler option is in effect, it indicates that CEESTART is never to be generated.

Syntax

```
     ┌─STA───┐
►►───┴─NOSTA─┴──────────────────────────────────────────────────►◄
```

Defaults

START

IPA effects

If you specify the START option for any compilation unit in the IPA Compile step, the compiler generates information for the IPA Link step. This option also affects the regular object module if you request one by specifying the IPA(OBJECT) option.

The IPA Link step uses the value of the START option that you specify for that step. It does not use the value that you specify for the IPA Compile step.

Predefined macros

None.

STATICINLINE | NOSTATICINLINE (C++ only)

Category

Language element control

Pragma equivalent

None.

Purpose

Controls whether inline functions are treated as having `static` or `extern` linkage.

When NOSTATICINLINE is in effect, the compiler treats inline functions as `extern`: only one function body is generated for a function marked with the `inline` function specifier, regardless of how many definitions of the same function appear in different source files. When STATICINLINE is in effect, the compiler treats inline functions as having `static` linkage: a separate function body is generated for each definition in a different source file of the same function marked with the `inline` function specifier.

Syntax

```
     ┌─NOSTATICI─┐
►►───┴─STATICI───┴──────────────────────────────────────────────►◄
```

Defaults

NOSTATICINLINE

Predefined macros

None.

Examples

Using the STATICINLINE option causes function `f` in the following declaration to be treated as static, even though it is not explicitly declared as such. A separate function body is created for each definition of the function. Note that this can lead to a substantial increase in code size.

```
inline void f() {/*...*/};
```

Using the NOSTATICINLINE compiler option gives `f` external linkage.

STRICT | NOSTRICT

Category

Optimization and tuning

Pragma equivalent

#pragma option_override(subprogram_name, ″OPT(STRICT)″)

Purpose

Used to prevent optimizations done by default at optimization levels OPT(3), and, optionally at OPT(2), from re-ordering instructions that could introduce rounding errors.

When the STRICT option is in effect, the compiler performs computational operations in a rigidly-defined order such that the results are always determinable and recreatable.

When the NOSTRICT compiler option is in effect, the compiler can reorder certain computations for better performance. However, the end result may differ from the result obtained when STRICT is specified.

Syntax

For NOOPT and OPT(2):

```
      ┌─STRICT───┐
►►────┴─NOSTRICT─┴──────────────────────────────────────────────────────►◄
```

For OPT(3):

```
      ┌─NOSTRICT─┐
►►────┴─STRICT───┴──────────────────────────────────────────────────────►◄
```

Defaults

For NOOPT and OPT(2), the default option is STRICT. For OPT(3), the default option is NOSTRICT.

Usage

STRICT disables the following optimizations:

- Performing code motion and scheduling on computations such as loads and floating-point computations that may trigger an exception.
- Relaxing conformance to IEEE rules.
- Reassociating floating-point expressions.

In IEEE floating-point mode, NOSTRICT sets FLOAT(MAF). To avoid this behavior, explicitly specify FLOAT(NOMAF).

IPA effects

The STRICT option has the same effect on the IPA compile step as it does for normal compilation.

See “FLOAT” on page 114 for information on the effect of the STRICT option on the IPA Link step.

Predefined macros

None.

STRICT_INDUCTION | NOSTRICT_INDUCTION

Category

Optimization and tuning

Pragma equivalent

None.

Purpose

Prevents the compiler from performing induction (loop counter) variable optimizations. These optimizations may be unsafe (may alter the semantics of your program) when there are integer overflow operations involving the induction variables.

When the STRICT_INDUCTION option is in effect, the compiler disables loop induction variable optimizations.

When the NOSTRICT_INDUCTION compiler option is in effect, the compiler permits loop induction variable optimizations.

Syntax

```
      ┌NOSTRICT_INDUC┐
►►────┴STRICT_INDUC──┴──────────────────────────────────────────────►◄
```

Defaults

NOSTRICT_INDUCTION

Note: The **c99** compiler invocation command for a regular compile in the z/OS UNIX System Services environment uses STRICT_INDUCTION as the default option.

Usage

Loop induction variable optimizations can change the result of a program if truncation or sign extension of a loop induction variable occurs as a result of variable overflow or wrap-around.

The STRICT_INDUCTION option only affects loops which have an induction (loop counter) variable declared as a different size than a register. Unless you intend such variables to overflow or wrap-around, use NOSTRICT_INDUCTION.

IPA effects

If you specify the STRICT_INDUCTION option for any compilation unit in the IPA Compile step, the compiler generates information for the IPA Link step. This option also affects the regular object module if you request one by specifying the IPA(OBJECT) option.

The IPA Link step merges and optimizes your application's code, and then divides it into sections for code generation. Each of these sections is a partition. The IPA Link step uses information from the IPA Compile step to ensure that an object is included in a compatible partition.

The compiler sets the value of the STRICT_INDUCTION option for a partition to the value of the first subprogram that is placed in the partition. During IPA inlining, subprograms with different STRICT_INDUCTION settings may be combined in the same partition. When this occurs, the resulting partition is always set to STRICT_INDUCTION.

You can override the setting of STRICT_INDUCTION by specifying the option on the IPA Link step. If you do so, all partitions will contain that value, and the prolog section of the IPA Link step listing will display the value.

Predefined macros

None.

SUPPRESS | NOSUPPRESS

Category

Listings, messages and compiler information

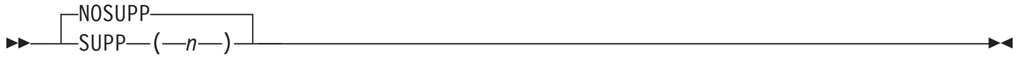
Pragma equivalent

None.

Purpose

Prevents specific informational or warning messages from being displayed or added to the listing file, if one is generated.

Syntax



Defaults

NOSUPPRESS

Note: The **c99** compiler invocation command for a regular compile in the z/OS UNIX System Services environment uses STRICT_INDUCTION as the default option.

Parameters

n Comma separated list of messages IDs.

Usage

For C, the message ID range that is affected is CCN3000 through CCN4399. For C++, the message ID range that is affected is CCN5000 to CCN6999, and CCN7500 to CCN8999. Note that this option has no effect on linker or operating system messages. Compiler messages that cause compilation to stop, such as (S) and (U) level messages cannot be suppressed.

IPA effects

The SUPPRESS option has the same effect on the IPA Link step that it does on a regular compilation.

Predefined macros

None.

TARGET

Category

Object code control

Pragma equivalent

#pragma target (C only)

Purpose

Generates an object module for the targeted operating system or run-time library.

Syntax

```
                     ┌─,──────────────────────┐
                     │  ┌─CURRENT──────────┐  │
                     │  │ ┌─LE─┐           │  │
►►──TARG──(──────────▼──┴─┴─IMS─┴──────────┴──┴──)──────────────────────────►◄
                        └─zOSV1R9───────┘
                          ├─zOSV1R10──────┤
                          ├─zOSV1R11──────┤
                          └─0xnnnnnnnn────┘
```

Defaults

TARGET(LE, CURRENT)

Parameters

The following suboptions target the run-time environment:

LE
: Generates object code to run under the Language Environment run-time environment. This is the default.

IMS
: Generates object code to run under the Information Management System (IMS) subsystem. If you are compiling the main program, you must also specify the PLIST(OS) option. TARGET(IMS) is not supported with LP64.

The following suboptions target the release at program run time:

CURRENT
: Generates object code that runs under the same version of z/OS with which the compiler is included. As the compiler is included with z/OS V1R11, TARGET(CURRENT) is the same as TARGET(zOSV1R11). This is the default.[2]

Note: The binder has an option called COMPAT that must be used when binding code to be run on an earlier version of the operating system. As of z/OS V1R8, you may need to explicitly specify the COMPAT option as

2. Note that for some releases of z/OS, z/OS XL C/C++ might not include a new release of the compiler. The same release of the compiler is then included with more than one z/OS release. The compiler is designed to run on all these z/OS releases. In this case, the compiler sets CURRENT to the z/OS release on which it is running. (It does so by querying the Language Environment Library version of the system.) You can specify a zOSV*x*R*y* suboption that corresponds to a release that is earlier or the same as CURRENT. You cannot specify a zOSV*x*R*y* suboption that corresponds to a release later than CURRENT.

the default is no longer emitted by the c89 utility or C/C++ cataloged procedures. As of z/OS V1R3, the binder output defaults to PM4.

zOSV1R9
Generates object code to run under z/OS Version 1 Release 9 and subsequent releases.

zOSV1R10
Generates object code to run under z/OS Version 1 Release 10 and subsequent releases.

| **zOSV1R11**
| Generates object code to run under z/OS Version 1 Release 11 and subsequent
| releases.

0x*nnnnnnnn*
An eight-digit hexadecimal literal string that specifies an operating system level. This string is intended for library providers and vendors to test header files on future releases and is an advanced feature. Most applications should use the other release suboptions. The layout of this literal is the same as the __TARGET_LIB__ macro.

Usage

Note: The suboptions for TARGET are not case-sensitive.

With the TARGET option, you can specify the run-time environment and release for your program's object module that z/OS XL C/C++ generates. This enables you to generate code that is downward compatible with earlier levels of the operating system while at the same time disallowing you from using library functions not available on the targeted release. With the TARGET option, you can compile and link an application on a higher level system, and run the application on a lower level system.

The compiler generates a comment that indicates the value of TARGET in your object module to aid you in diagnosing problems in your program.

| To use the TARGET option, select a run-time environment of either LE or IMS. Then
| select the desired release, for example, zOSV1R9. If you do not select a run-time
| environment or release, the compiler uses a default of TARGET(LE, zOSV1R11).

TARGET() generates object code to run under the Language Environment run-time environment. It is the same as TARGET(LE,CURRENT).

If you specify more than one suboption from each group of suboptions (that is, the run-time environment, or the release) the compiler uses the last specified suboption for each group.

The compiler applies and resolves defaults after it views all the entered suboptions. For example, TARGET(LE,0x41080000, IMS, zOSV1R9, LE) resolves to TARGET(LE, zOSV1R9). TARGET(LE, 0x41080000, IMS, zOSV1R9) resolves to TARGET(IMS, zOSV1R9). TARGET(LE, 0x41080000, IMS) resolves to TARGET(IMS, 0x41080000).

| The default value of the ARCHITECTURE compiler option depends on the value of
| the TARGET release suboption. For TARGET(zOSV1R9) and above, the default is
| ARCH(5).

The TARGET release suboptions (CURRENT, zOSV1R9, zOSV1R10 and zOSV1R11) help you to generate code that can be executed on a particular release of a z/OS system, and on subsequent releases.

In order to use these suboptions, you must:
- Use the z/OS V1R11 class library header files (found in the CBC.SCLBH.* data sets) during compilation

See Appendix A, "Prelinking and linking z/OS XL C/C++ programs," on page 601 for details on prelinking and linking applications.

The TARGET release suboptions will not allow you to exploit new functions provided on the newer release. Rather, they allow you to build an application on a newer release and run it on an older release.

The hexadecimal string literal suboption enables you to specify an operating system level. It is an advanced feature that is intended for library providers and vendors to test header files on future releases. Most applications should use the other release suboptions instead of this string literal. The layout of this literal is the same as the __TARGET_LIB__ macro.

The compiler checks that there are exactly 8 hexadecimal digits. The compiler performs no further validation checks.

The compiler uses a two step process to specify the operating system level:
- The hexadecimal value will be used, as specified, to set the __TARGET_LIB__ macro.
- The compiler determines the operating system level implied by this literal.

If the level corresponds to a valid suboption name, the compiler behaves as though that suboption is specified. Otherwise, the compiler uses the next lower operating system suboption name. If there is no lower suboption name, the compiler behaves as though you have specified an unsupported release. Note that the compiler sets the __TARGET_LIB__ macro to the value that you specify, even if it does not correspond to a valid operating system level.

The following information describes the restrictions for C/C++:
- All input libraries used during the application build process must be the appropriate level for the target release.
  - The current level of the Language Environment data sets can be used to target to previous releases. Use these Language Environment data sets during the assembly, compilation, pre-link, link-edit, and bind phases.
  - For C++ class libraries, use the current release class library header files during compilation; use the class library data sets for the targeted release during pre-link, link-edit, and bind.
  - Ensure that any other libraries incorporated in the application, are compatible with the target release.

While there are no restrictions on the use of ARCH and TUNE with TARGET, ensure that the level specified is consistent with the target hardware.

TARGET Release Suboption	**Restrictions**
zOSV1R10	LE functions are used for wmem() built-in functions.

TARGET Release Suboption	Restrictions
zOSV1R9	ISO/IEC TR 19768 – Technical Report on C++ Library Extensions (informally named "TR1") is not supported.

Only options or features that cannot be supported on that operating system level are disabled. For example, STRICT_INDUCTION is allowed on all operating system levels. An option or feature that is disabled by one operating system level is also disabled by all earlier operating system levels.

The following information describes the restrictions for C:

- See the *z/OS XL C/C++ Language Reference* for information on the **#pragma target()** directive.
- If you specify TARGET(zOSV*x*R*y*) on the command line, and one or more of the disallowed options is specified, the compiler issues a warning message and disables the option.

IPA effects

If you specify the TARGET option for any compilation unit in the IPA Compile step, the compiler generates information for the IPA Link step. This option also affects the regular object module if you request one by specifying the IPA(OBJECT) option.

When you are performing the IPA Compile to generate IPA Object files, ensure that you are using the appropriate header library files.

If you specify TARGET on the IPA Link step:

- It overrides the TARGET value that you specified for the IPA Compile step.
- It overrides the value that you specified for **#pragma runopts(ENV)**. If you specify TARGET(LE) or TARGET(), the IPA Link step specifies **#pragma runopts(ENV(MVS))**. If you specify TARGET(IMS), the IPA Link step specifies **#pragma runopts(ENV(IMS))**.
- It may override the value that you specified for **#pragma runopts(PLIST)**, which specifies the run-time option during program execution. If you specify TARGET(LE) or TARGET(), and you set the value set for the PLIST option to something other than HOST, the IPA Link step sets the values of **#pragma runopts(PLIST)** and the PLIST compiler option to IMS. If you specify TARGET(IMS), the IPA Link step unconditionally sets the value of **#pragma runopts(PLIST)** to IMS.

The IPA Link step accepts the release suboptions, for example, CURRENT or zOSV1R10. However, when using TARGET suboptions ensure that:

- All IPA Object files are compiled with the appropriate TARGET suboption and header files
- All non-IPA object files are compiled with the appropriate TARGET suboption and header files
- All other input libraries are compatible with the specified run-time release

Predefined macros

When you invoke the TARGET(OSV*x*R*y*) release suboptions, the compiler sets the __TARGET_LIB__ macro. See *z/OS XL C/C++ Language Reference* for more information about this macro.

Examples

To generate code that will execute on a z/OS V1R9 system, using a z/OS V1R11 application development system, you must:

- Specify the compiler option TARGET(zOSV1R9) on the C/C++ compiles. Note: The programmer is responsible for ensuring that they are not exploiting any Language Environment functions that are unavailable on z/OS V1R9.
- Use the z/OS V1R11 Language Environment data sets (CEE.SCEE*) during the assembly, compilation, pre-link, link-edit, and bind phases.
- Use the z/OS V1R11 class library header data sets (CBC.SCLBH.*) during compilation. Use the z/OS V1R9 class library data sets (SCLBCPP, SCLBOBC, SCLBOXL, SCLBSID, SCLBXL) during pre-link, link-edit, and bind.

The following information shows examples of using the hexadecimal string literal suboption:

TARGET(0x410B0000)
: Equivalent to TARGET(zOSV1R11).

TARGET(0x410A0000)
: Equivalent to TARGET(zOSV1R10).

TARGET(0x41090000)
: Equivalent to TARGET(zOSV1R9).

TARGET(0xA3120000)
: This does not match any existing operating system release suboption name. The next lower operating system level implied by this literal, which the compiler considers valid, is CURRENT. Thus, the compiler sets the __TARGET_LIB__ macro to `0xA3120000`, and behaves as though you have specified TARGET(CURRENT).

TARGET(0x21010000)
: This does not match any existing operating system release suboption name, and specifies a release earlier than the earliest supported release. In this instance, the compiler sets the __TARGET_LIB__ macro to `0x21010000`, and behaves as though you have specified an unsupported release.

TEMPINC | NOTEMPINC (C++ only)

Category

C++ template

Pragma equivalent

None.

Purpose

Generates separate template instantiation files for template functions and class declarations, and places these files in a directory or PDS, which can be optionally specified.

Note: The recommended method for handling template instantiations is to use the TEMPLATEREGISTRY compiler option instead of the TEMPINC compiler option. For more information on the TEMPLATEREGISTRY compiler option, see "TEMPLATEREGISTRY | NOTEMPLATEREGISTRY (C++ only)" on page 247.

Syntax

```
     ┌─TEMPINC───┐
►►───┴─NOTEMPINC─┴─┬───────────────┬──────────────────────────────────►◄
                   └─(─location─)─┘
```

Defaults

For a PDS directory, the default option is TEMPINC(TEMPINC). For a z/OS UNIX System Services file system directory, the default option is TEMPINC(./tempinc).

Note: The **c++** compiler invocation command for a regular compile in the z/OS UNIX environment uses TEMPINC(tempinc) as the default option.

In the z/OS UNIX environment, the template instantiation files are by default produced in a `./tempinc` directory when C++ source files are compiled with the TEMPINC option.

When the bind step is invoked, using the **c++** or **cxx** commands, and in the presence of `./tempinc` directory, the XL C++ compiler is automatically invoked to compile all template instantiation files in the `./tempinc` directory. If the command line only includes binder options, the template instantiation files are compiled using the XL C++ compiler defaults. If this is not appropriate for compiling the template instantiation files, all required XL C++ compiler options must be specified on the command line even though the command line is intended to invoke the bind step.

Automatic invocation of the XL C++ compiler is performed by the c89 utility when using the **c++** and **cxx** commands. The same is true when using C++ invocation commands from the xlc utility, except the xlc utility invokes the bind step using the c89 utility. When the xlc utility invokes the c89 utility for the bind step it only passes the binder options, so the template instantiation files are always compiled with the XL C++ compiler defaults. For this reason, the TEMPINC method for processing template instantiations is not recommended with the xlc utility. The TEMPLATEREGISTRY method should be used instead.

Parameters

location
A PDS or a z/OS UNIX file system directory that will contain all template instantiation files. When a PDS is used to contain all template instantiation files, all compilations of a given application must be sequential otherwise two different compilations might need access to the same PDS member at the same time. This can cause a collision leading to incorrect compilation results. For parallel builds, use a z/OS UNIX file system directory for the template instantiation files.

Usage

If you do not specify a *location*, the compiler places all template instantiation files in a default location. If the source resides in a data set, the default location is a PDS with a low-level qualifier of TEMPINC. The high-level qualifier is the userid under which the compiler is running. If the source resides in a z/OS UNIX file, the default location is the z/OS UNIX file system directory `./tempinc`.

The NOTEMPINC option can optionally take a *filename* suboption. This *filename* then becomes the default. If you subsequently use the TEMPINC option without a *filename* suboption, then the compiler uses the *filename* that you specified in the earlier NOTEMPINC. For example, the following specifications have the same result:

```
c++ -Wc,"NOTEMPINC(hello)" -Wc,TEMPINC ./hello.C
```

```
c++ -Wc,"TEMPINC(hello)" ./hello.C
```

If you specify TEMPINC and NOTEMPINC multiple times, the compiler uses the last specified option with the last specified suboption. For example, the following specifications have the same result:

```
c++ -Wc,"NOTEMPINC(hello)" -Wc,"TEMPINC(n1)" -Wc,"NOTEMPINC(test)" -Wc,TEMPINC
./hello.C
```

```
c++ -Wc,"TEMPINC(test)" ./hello.C
```

If you have large numbers of recursive templates, consider using FASTT. See "FASTTEMPINC | NOFASTTEMPINC (C++ only)" on page 112 for details.

Note: If you use the following form of the command in the batch environment where xxx is an unallocated data set, you may get undefined results.

```
TEMPINC(xxx)
```

IPA effects

The IPA Link step issues a diagnostic message if you specify the TEMPINC option for that step.

Predefined macros

__TEMPINC__ is predefined to 1 when the TEMPINC compiler option is in effect; otherwise it is undefined.

TEMPLATERECOMPILE | NOTEMPLATERECOMPILE (C++ only)

Category

C++ template

Pragma equivalent

None.

Purpose

Helps manage dependencies between compilation units that have been compiled using the TEMPLATEREGISTRY compiler option.

Syntax

```
     ┌─TEMPLATEREC───┐
►►───┴─NOTEMPLATEREC─┴──────────────────────────────────────────────────►◄
```

Defaults

TEMPLATERECOMPILE

Usage

If a source file that has been compiled previously is compiled again, the TEMPLATERECOMPILE option consults the template registry to determine whether changes to this source file require the recompile of other compilation units. This can occur when the source file has changed in such a way that it no longer references a given instantiation and the corresponding object file previously contained the instantiation. If so, affected compilation units will be recompiled automatically.

The TEMPLATERECOMPILE option requires that object files generated by the compiler remain in the PDS or subdirectory to which they were originally written. If your automated build process moves object files from their original PDS or subdirectory, use the NOTEMPLATERECOMPILE option whenever TEMPLATEREGISTRY is enabled.

IPA effects

The IPA Link step does not accept the TEMPLATERECOMPILE option. The compiler issues a warning message if you specify this option in the IPA Link step.

Predefined macros

None.

TEMPLATEREGISTRY | NOTEMPLATEREGISTRY (C++ only)

Category

C++ template

Pragma equivalent

None.

Purpose

Maintains records of all templates as they are encountered in the source and is designed to ensure that only one instantiation of each template is made.

Syntax

```
     ┌─NOTEMPL─┐
►►───┴─TEMPL───┴──┬──────────────────┬───────────────────────────────────────►◄
                  └─(registryFile)───┘
```

Defaults

NOTEMPLATEREGISTRY

Parameters

registryFile
: The location for template registry information. The default location is dependent on the OE compiler option. When a template registry file is in a sequential data set file, all compilations of a given application must be sequential otherwise two different compilations might need access to the same sequential data set file at the same time. This can cause a collision leading to incorrect compilation results. For parallel builds, use a z/OS UNIX System Services file as the template registry file.

Usage

When the TEMPLATEREGISTRY compiler option is in effect, and the compiler encounters a reference to a template instantiation for the first time, the instantiation is generated and the related object code is placed in the current object file. Any further references to identical instantiations of the same template in different compilation units are recorded but the redundant instantiations are not generated.

No special file organization is required to use the TEMPLATEREGISTRY option. If you do not specify a location, the compiler places all template registry information in a default location. If the NOOE compiler option is in effect, the default location is a sequential data set that has a high-level qualifier that is the userid under which the compiler is running, with `.TEMPLREG` appended as the low-level qualifier. If the OE compiler option is in effect, the default location is the z/OS UNIX file `./templreg`. If a file currently exists with the name of the file name used for TEMPLATEREGISTRY, then that file will be overwritten. For more information on using the TEMPLATEREGISTRY option, see *z/OS XL C/C++ Programming Guide*.

Note: TEMPINC and TEMPLATEREGISTRY cannot be used together because they are mutually exclusive. If you specify TEMPLATEREGISTRY, then you set NOTEMPINC. If you use the following form of the command in a JES3 batch environment where `xxx` is an unallocated data set, you may get undefined results.

```
TEMPLREG(xxx)
```

IPA effects

The IPA Link step issues a diagnostic message if you specify the TEMPLATEREGISTRY option for that step.

Predefined macros

None.

TERMINAL | NOTERMINAL

Category

Listings, messages, and compiler information

Pragma equivalent

None.

Purpose

Directs diagnostic messages to be displayed on the terminal.

Syntax

```
      ┌─TERM───┐
►►────┴─NOTERM─┴──────────────────────────────────────────────────────────►◄
```

Defaults

TERMINAL

Usage

When the TERMINAL compiler option is in effect, it directs all of the diagnostic messages of the compiler to `stderr`.

Under z/OS batch, the default for `stderr` is SYSPRINT.

If you specify the PPONLY option, the compiler turns on TERM.

IPA effects

The TERMINAL compiler option has the same effect on the IPA Link step as it does on a regular compile step.

Predefined macros

None.

Related information

For more information on the PPONLY compiler option, see "PPONLY | NOPPONLY" on page 203.

TEST | NOTEST

Category

Error checking and debugging

Pragma equivalent

#pragma options(test) (C only), **#pragma options(notest)** (C only)

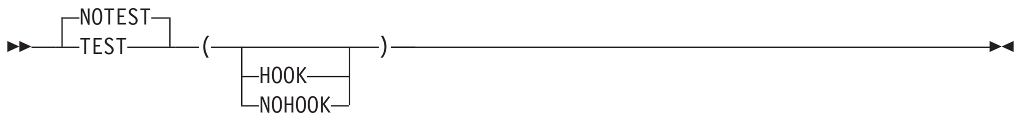
Purpose

Generates debugging information that Debug Tool needs to debug your program.

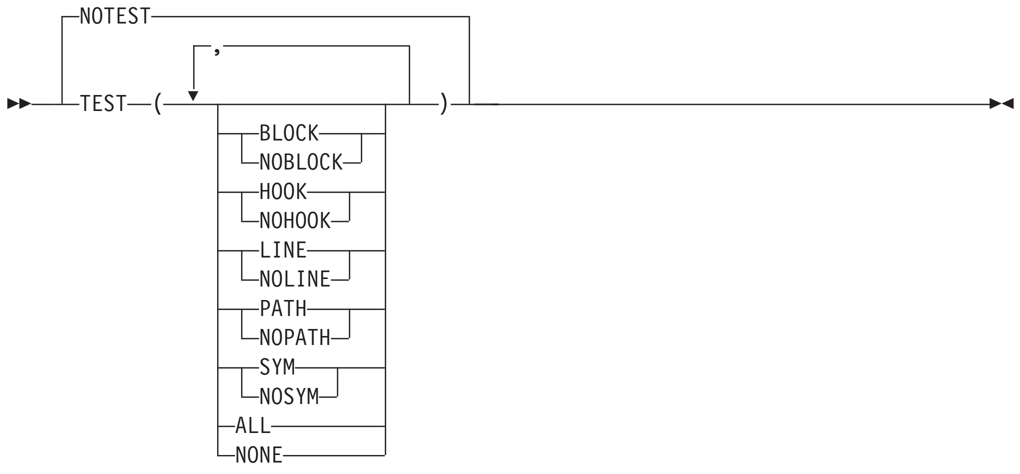
When the NOTEST compiler option is in effect, debugging information is not generated and you cannot trace your program with the Performance Analyzer.

Syntax

The TEST suboptions that are common to C compile, C++ compile, and IPA Link steps are:



Additional z/OS XL C compile suboptions are:



Defaults

For C++, the default option is NOTEST(HOOK). For C, the default option is: NOTEST(HOOK,SYM,BLOCK,LINE,PATH).

The default for the z/OS UNIX System Services utilities is NOTEST.

Parameters

The TEST suboptions parameters that are common to C compile, C++ compile, and IPA Link steps are:

HOOK | NOHOOK	When NOOPT is in effect	When OPT is in effect
HOOK	• For C++ compile, generates all possible hooks. For C compile, generates all possible hooks based on current settings of BLOCK, LINE, and PATH suboptions. For IPA Link, generates Function Entry, Function Exit, Function Call, and Function Return hooks. • For C++ compile, generates symbol information. For C compile, generates symbol information unless NOSYM is specified. For IPA Link, does not generate symbol information.	• Generates Function Entry, Function Exit, Function Call and Function Return hooks. • Does not generate symbol information.
NOHOOK	• Does not generate any hooks. • For C++ compile, generates symbol information. For C compile, generates symbol information based on the current settings of SYM and BLOCK. For IPA Link, does not generate any symbol information.	• Does not generate any hooks. • Does not generate symbol information.

Additional z/OS XL C compile suboptions parameters are:

SYM
: Generates symbol tables in the object output of the program that give you access to variables and other symbol information.

- You can reference all program variables by name, allowing you to examine them or use them in expressions.
- You can use the Debug Tool command GOTO to branch to a label (paragraph or section name).
- The Performance Analyzer does not use symbol information. Specify NOSYM if you want to trace the program with the Performance Analyzer.

BLOCK
: Inserts only block entry and exit hooks into the object output of the program. A block is any number of data definitions, declarations, or statements that are enclosed within a single set of braces. BLOCK also creates entry hooks and exit hooks for nested blocks. If SYM is enabled, symbol tables are generated for variables local to these nested blocks.

- You can only gain control at entry and exit of blocks.
- Issuing a command such as STEP causes your program to run, until it reaches the exit point.
- The Performance Analyzer does not use block entry and exit hooks. Specify NOBLOCK if you want to trace the program with the Performance Analyzer.

LINE
: Generates hooks at most executable statements. Hooks are not generated for the following:

- Lines that identify blocks (lines that contain braces)
- Null statements
- Labels
- Statements that begin in an `#include` file
- The Performance Analyzer does not use statement hooks. Specify NOLINE if you want to trace the program with the Performance Analyzer.

PATH Generates hooks at all path points; for example, hooks are inserted at if-then-else points.

- This option does not influence the generation of entry and exit hooks for nested blocks. You must specify the BLOCK suboption if you desire such hooks.
- Debug Tool can gain control only at path points and block entry and exit points. If you attempt to STEP through your program, Debug Tool gains control only at statements that coincide with path points, giving the appearance that not all statements are executed.
- The Debug Tool command GOTO is valid only for statements and labels that coincide with path points.
- The Performance Analyzer uses function call and function return hooks. Specify PATH if you want to trace the program with the Performance Analyzer.

ALL Inserts block and line hooks, and generates symbol table. Hooks are generated at all statements, all path points (if-then-else, calls, and so on), and all function entry and exit points.

ALL is equivalent to TEST(HOOK, BLOCK, LINE, PATH, SYM).

NONE Generates all compiled-in hooks only at function entry and exit points. Block hooks and line hooks are not inserted, and the symbol tables are suppressed.

TEST(NONE) is equivalent to TEST(HOOK, NOBLOCK, NOLINE, NOPATH, NOSYM).

Usage

The TEST suboptions generate symbol tables and program hooks. Debug Tool uses these tables and hooks to debug your program. The Performance Analyzer uses these hooks to trace your program. The choices you make when compiling your program affect the amount of Debug Tool function available during your debugging session. These choices also impact the ability of the Performance Analyzer to trace your program.

To look at the flow of your code with Debug Tool, or to trace the flow of your code with the Performance Analyzer, use the HOOK suboption with OPT in effect. These suboptions generate function entry, function exit, function call, and function return hooks. They do not generate symbol information.

When NOOPT is in effect, and you use the HOOK suboption, the debugger runs slower, but all Debug Tool commands such as AT ENTRY * are available. You must specify the HOOK suboption in order to trace your program with the Performance Analyzer.

In order for the debugger to access the source lines, the primary source file of a compilation unit should come from one file (or sequential data set or PDS member), and not be the result of concatenated DD statements. This is because only the name of the first data set is known to the compiler reading the concatenation; the debug information generated in this case would contain only the first data set name.

All the source files, including header files, should not be temporary files, and should be available to the debugger under the same name as used during compilation.

You can use the CSECT option with the TEST option to place your debug information in a named CSECT. This enables the compiler and linker to collect the debug information in your module together, which may improve the run-time performance of your program.

If you specify the INLINE and TEST compiler options when NOOPTIMIZE is in effect, INLINE is ignored.

If you specify the TEST option, the compiler turns on GONUMBER.

Note: If your code uses any of the following, you cannot debug it with the MFI Debug Tool:

- IEEE code
- Code that uses the long long data type
- Code that runs in a POSIX environment

You must use either the C/C++ Productivity Tools for OS/390 or dbx.

The TEST suboptions BLOCK, LINE, and PATH regulate the points where the compiler inserts program hooks. When you set breakpoints, they are associated with the hooks which are used to instruct Debug Tool where to gain control of your program.

The symbol table suboption SYM regulates the inclusion of symbol tables into the object output of the compiler. Debug Tool uses the symbol tables to obtain information about the variables in the program.

Note: When the OPTIMIZE and TEST options are both specified, the TEST suboptions are set by the compiler to TEST(HOOK, NOBLOCK, NOLINE, NOPATH, NOSYM) regardless of what you have specified. The behavior of the TEST option in this case is as described in the table in the z/OS XL C/C++ section of the TEST | NOTEST option for the HOOK suboption.

For z/OS XL C compile, you can specify the TEST | NOTEST option on the command line and in the **#pragma options** preprocessor directive. When you use both methods, the option on the command line takes precedence. For example, if you usually do not want to generate debugging information when you compile a program, you can specify the NOTEST option on a **#pragma options** preprocessor directive. When you do want to generate debugging information, you can then override the NOTEST option by specifying TEST on the command line rather than editing your source program. Suboptions that you specified in a **#pragma options (notest)** directive, or with the NOTEST compiler option, are used if TEST is subsequently specified on the command line.

Note: The TEST compiler option is ignored when specified with the LP64 compiler option.

IPA effects

On the IPA Compile step, you can specify all of the TEST suboptions that are appropriate for the language of the code that you are compiling. However, they affect processing only if you requested code generation, and only the conventional object file is affected. If you specify the NOOBJECT suboption of the IPA compiler option on the IPA Compile step, the IPA Compile step ignores the TEST option.

The IPA Link step supports only the TEST, TEST(HOOK), TEST(NOHOOK), and NOTEST options. If you specify TEST(HOOK) or TEST, the IPA Link step generates function call, entry, exit, and return hooks. It does not generate symbol table information. If you specify TEST(NOHOOK), the IPA Link step generates limited debug information without any hooks. If you specify any other TEST suboptions for the IPA Link step, it turns them off and issues a warning message.

Note: See “DEBUG | NODEBUG” on page 94 for more information on debugging applications linked with IPA.

Predefined macros
None.

TMPLPARSE (C++ only)

Category
C++ template

Pragma equivalent
None.

Purpose
Controls whether parsing and semantic checking are applied to template definitions.

Syntax

```
                      ┌─NO──────┐
►►──TMPLPARSE──(──────┼─WARNING─┼──)─────────────────────────────────────────►◄
                      └─ERROR───┘
```

Defaults
TMPLPARSE(NO)

Parameters

ERROR
Treats problems in template definitions as errors, even if the template is not instantiated.

NO
Do not parse template definitions.

WARNING
Parses template definitions and issues warning messages for semantic errors.

Usage
This option applies to template definitions, not their instantiations. Regardless of the setting of this option, error messages are produced for problems that appear outside definitions. For example, messages are always produced for errors found during the parsing or semantic checking of constructs such as the following:

- return type of a function template
- parameter list of a function template

IPA effects
The IPA Link step issues a diagnostic message if you specify the TMPLPARSE option for that step.

Predefined macros

None.

TUNE

Category

Optimization and tuning

Pragma equivalent

#pragma options(tune) (C only)

Purpose

Tunes instruction selection, scheduling, and other implementation-dependent performance enhancements for a specific implementation of a hardware architecture.

Syntax

```
►►──TUN──(──n──)──────────────────────────────────────────────────────►◄
```

Defaults

TUNE(5)

Parameters

n
: Specifies the group to which a model number belongs as a sub-parameter. If you specify a model which does not exist or is not supported, a warning message is issued stating that the suboption is invalid and that the default will be used. Current models that are supported are:

0 This option generates code that is executable on all models, but it will not be able to take advantage of architectural differences on the models specified in the following information.

1 This option generates code that is executable on all models but that is optimized for the following models:
- 9021-520, 9021-640, 9021-660, 9021-740, 9021-820, 9021-860, and 9021-900
- 9021-xx1, 9021-xx2, and 9672-Rx2 (G1)

2 This option generates code that is executable on all models but that is optimized for the following models:
- 9672-Rx3 (G2), 9672-Rx4 (G3), and 2003
- 9672-Rx1, 9672-Exx, and 9672-Pxx

3 This option generates code that is executable on all models but that is optimized for the following and follow-on models: 9672-Rx5 (G4), 9672-xx6 (G5), and 9672-xx7 (G6).

4 This option generates code that is executable on all models but that is optimized for the model 2064-100 (z900).

5 This option is the default. This option generates code that is executable on all models but that is optimized for the model 2064-100 (z900) in z/Architecture mode.

6 This option generates code that is executable on all models, but is optimized for the 2084-xxx (z990) models.

7 This option generates code that is executable on all models, but is optimized for the 2094-xxx (IBM System z9) models.

8 This option generates code that is executable on all models, but is optimized for the 2097-xxx (IBM System z10) models.

Note: For these system machine models, x indicates any value. For example, 9672-Rx4 means 9672-RA4 through to 9672-RY4 and 9672-R14 through to 9672-R94 (the entire range of G3 processors), not just 9672-RX4.

Usage

The TUNE option specifies the architecture for which the executable program will be optimized. The TUNE level controls how the compiler selects and orders the available machine instructions, while staying within the restrictions of the ARCH level in effect. It does so in order to provide the highest performance possible on the given TUNE architecture from those that are allowed in the generated code. It also controls instruction scheduling (the order in which instructions are generated to perform a particular operation). Note that TUNE impacts performance only; it does not impact the processor model on which you will be able to run your application.

Select TUNE to match the architecture of the machine where your application will run most often. Use TUNE in cooperation with ARCH. TUNE must always be greater or equal to ARCH because you will want to tune an application for a machine on which it can run. The compiler enforces this by adjusting TUNE up rather than ARCH down. TUNE does not specify where an application can run. It is primarily an optimization option. For many models, the best TUNE level is not the best ARCH level. For example, the correct choices for model 9672-Rx5 (G4) are ARCH(2) and TUNE(3). For more information on the interaction between TUNE and ARCH see “ARCHITECTURE” on page 69.

Note: If the TUNE level is lower than the specified ARCH level, the compiler forces TUNE to match the ARCH level or uses the default TUNE level, whichever is greater.

A comment that indicates the level of the TUNE option will be generated in your object module to aid you in diagnosing your program.

IPA effects

If you specify the TUNE option for any compilation unit in the IPA Compile step, the compiler saves information for the IPA Link step. This option also affects the regular object module if you request one by specifying the IPA(OBJECT) option.

The IPA Link step merges and optimizes the application code, and then divides it into sections for code generation. Each of these sections is a partition.

If you specify the TUNE option for the IPA Link step, it uses the value of the option you specify. The value you specify appears in the IPA Link step Prolog listing section and all Partition Map listing sections.

If you do not specify the option on the IPA Link step, the value it uses for a partition depends upon the TUNE option you specified during the IPA Compile step for any compilation unit that provided code for that partition. If you specified the same TUNE value for all compilation units, the IPA Link step uses that value. If you specified different TUNE values, the IPA Link step uses the highest value of TUNE.

If the resulting level of TUNE is lower than the level of ARCH, TUNE is set to the level of ARCH.

The Partition Map section of the IPA Link step listing, and the object module display the final option value for each partition. If you override this option on the IPA Link step, the Prolog section of the IPA Link step listing displays the value of the option.

The Compiler Options Map section of the IPA Link step listing displays the value of the TUNE option that you specified on the IPA Compile step for each object file.

Predefined macros

__TUNE__ is predefined to the value specified by the TUNE compiler option.

UNDEFINE

Category

Language element control

Pragma equivalent

None.

Purpose

Undefines preprocessor macro names.

Syntax

```
                 ┌─,──────┐
                 ▼        │
►►──UNDEF──(────name──────┴──)──────────────────────────────────────────►◄
```

Defaults

Not applicable.

Parameters

name Specifies a preprocessor macro name.

Usage

UNDEFINE(*name*) removes any value that *name* may have and makes its value undefined. For example, if you set `OS2` to `1` with `DEF(OS2=1)`, you can use the UNDEF(OS2) option to remove that value.

In the z/OS UNIX System Services environment, you can unset variables by specifying -U when using the **c89**, **cc**, or **c++** commands.

Note: **c89** preprocesses -D and -U flags before passing them to the compiler. **xlc** just passes -D and -U to the compiler, which interprets them as DEFINE and UNDEFINE. For more information, see Chapter 21, “c89 — Compiler invocation using host environment variables,” on page 535 or Chapter 24, “xlc — Compiler invocation using a customizable configuration file,” on page 577.

Predefined macros

None.

UNROLL

Category

Optimization and tuning

Pragma equivalent

#pragma unroll

Purpose

Controls loop unrolling, for improved performance.

Syntax

```
              ┌─AUTO─┐
►►──UNROLL────┼─YES──┼──────────────────────────────────────────►◄
              └─NO───┘
```

Defaults

UNROLL(AUTO)

Parameters

YES	Allows the compiler to unroll loops that are annotated (for example, using a pragma), unless it is overridden by **#pragma nounroll**.
NO	Means that the compiler is not permitted to unroll loops in the compilation unit, unless **unroll** or **unroll(n)** pragmas are specified for particular loops.
AUTO	This option is the default. It enables the compiler to unroll loops that are annotated (for example, using a pragma) and loops which the compiler has decided (via heuristics) are appropriate for unrolling. AUTO should only be specified if you have specified OPTIMIZE(2) or greater and COMPACT is not specified.

Usage

The UNROLL compiler option instructs the compiler to perform loop unrolling, which is an optimization that replicates a loop body multiple times, and adjusts the loop control code accordingly. Loop unrolling exposes instruction level parallelism for instruction scheduling and software pipelining and thus can improve a program's performance. It also increases code size in the new loop body, which may increase pressure on register allocation, cause register spilling, and therefore cause a loss in performance. Before applying unrolling to a loop, you must evaluate these tradeoffs. In order to check if the unroll option improves performance of a particular application, you should compile your program with the usual options, run it with a representative workload, recompile it with the UNROLL option and/or unroll pragmas, and rerun it under the same conditions to see if the UNROLL option leads to a performance improvement.

Predefined macros

None.

UPCONV | NOUPCONV (C only)

Category

Portability and migration

Pragma equivalent

#pragma options(upconv) (C only), **#pragma options(noupconv)** (C only)

Purpose

Specifies whether the `unsigned` specification is preserved when integral promotions are performed.

Syntax

```
      ┌─NOUPC─┐
►►────┴─UPC───┴──────────────────────────────────────────────────────►◄
```

Defaults

NOUPCONV

Note: The **cc** compiler invocation command for a regular compile in the z/OS UNIX System Services environment uses UPCONV as the default option.

Usage

The UPCONV option causes the z/OS XL C compiler to follow unsignedness preserving rules when doing C type conversions; that is, when widening all integral types (`char`, `short`, `int`, `long`). Use this option when compiling older C programs that depend on the *K&R C* conversion rules.

Note: This document uses the term *K&R C* to refer to the C language plus the generally accepted extensions produced by Brian Kernighan and Dennis Ritchie that were in use prior to the ISO standardization of C.

Whenever you specify the UPCONV compiler option, a comment noting its use will be generated in your object module to aid you in diagnosing your program.

Predefined macros

None.

WARN64 | NOWARN64

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Generates diagnostic messages, which enable checking for possible data conversion problems between 32-bit and 64-bit compiler modes.

Syntax

```
      ┌─NOWARN64─┐
►►────┴─WARN64───┴───────────────────────────────────────────────────►◄
```

Defaults

NOWARN64

Usage

Use the FLAG(I) option to display any informational messages.

WARN64 warns you about any code fragments that have the following types of portability errors:

- A constant that selected an `unsigned long int` data type in 31-bit mode may fit within a `long int` data type in 64-bit mode
- A constant larger than UINT_MAX, but smaller than ULONGLONG_MAX will overflow in 31-bit mode, but will be acceptable in an `unsigned long` or `signed long` in 64-bit mode

It also warns you about the following types of possible portability errors:

- Loss of digits when you assign a `long` type to an `int` type
- Change in the result when you assign an `int` to a `long` type
- Loss of high-order bytes of a pointer when a pointer type is assigned to an `int` type
- Incorrect pointer when an `int` type is assigned to a pointer type
- Change of a constant value when the constant is assigned to a `long` type

Predefined macros

None.

WARN0X | NOWARN0X (C++0X only)

Category

Error checking and debugging

Pragma equivalent

None.

Purpose

Generates messages about differences caused by migration from the C++98 standard to the C++0x standard.

Syntax

```
      ┌─NOWARN0X─┐
►►────┴─WARN0X───┴──────────────────────────────────────────────►◄
```

Defaults

NOWARN0X

Usage

This option controls whether to inform you with messages about differences in their programs caused by migration from the C++98 standard to the C++0x standard.

Note: C++0x is a new version of the standard for the C++ programming language. This is a draft standard and has not been officially adopted in its entirety. The implementation of this language level is based on IBM's interpretation of the draft C++0x standard and is subject to change at any time without notice. IBM will make no attempt to maintain compatibility with earlier releases and therefore the C++0x language extension should not be relied on as a stable programming interface.

Predefined macros
None.

WSIZEOF | NOWSIZEOF

Category
Object code control

Pragma equivalent
#pragma wsizeof(on)

Purpose
Causes the `sizeof` operator to return the widened size for function return types.

When the WSIZEOF compiler option is in effect, `sizeof` returns the size of the widened type for function return types instead of the size of the original return type.

When the NOWSIZEOF compiler option is in effect, `sizeof` returns the size of the original return type.

Syntax

```
      ┌─NOWSIZEOF─┐
►►────┴─WSIZEOF───┴──────────────────────────────────────────────────►◄
```

Defaults
NOWSIZEOF

Usage
When the `sizeof` operator was applied to a function return type using the WSIZEOF compiler option, earlier C and C++ compilers (prior to and including C/C++ for IBM MVS/ESA™ Version 3 Release 1) returned the size of the widened type instead of the original type. For example, if the following code fragment, was compiled with an earlier compiler, `i` would have a value of `4`.

```
char foo();
i = sizeof foo();
```

Using the z/OS XL C/C++ compiler, `i` has a value of `1`, which is the size of the original type `char`.

The WSIZEOF compiler option toggles the behavior of the `sizeof` operator between that of the C and C++ compilers prior to and including C/C++ MVS Version 3 Release 1, and z/OS XL C/C++.

Predefined macros
None.

XPLINK | NOXPLINK

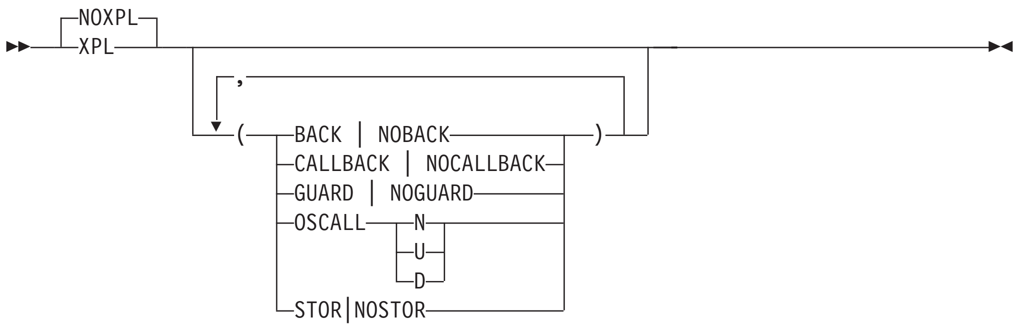
Category
Object code control

Pragma equivalent
None.

Purpose

Uses a z/OS linkage specifically designed to increase performance.

Syntax



Defaults

- NOXPLINK
- For LP64, the default is XPLINK.

Parameters

BACKCHAIN | NOBACKCHAIN

DEFAULT: NOBACKCHAIN

If you specify BACKCHAIN, the compiler generates a prolog that saves information about the calling function in the stack frame of the called function. This facilitates debugging using storage dumps. Use this suboption in conjunction with STOREARGS to make storage dumps more useful.

CALLBACK | NOCALLBACK

DEFAULT: NOCALLBACK

XPLINK(CALLBACK) is primarily intended to enable function pointer calls across XPLINK DLLs and non-XPLINK programs. With XPLINK, function calls are supported across a DLL boundary with certain restrictions. In particular, if a function pointer is created by a non-XPLINK caller pointing to an XPLINK function, it can be passed as an argument via an exported function into the DLL, which can then use it as callback. This is because the compiler knows about the function pointer argument and is able to insert code to fix-up the function pointer. However, non-XPLINK function pointers passed into the DLL by other means are not supported. If you specify CALLBACK, all calls via function pointers will be considered potentially incompatible, and fix-up code will be inserted by the compiler at the locations of the incompatible DLL callbacks through function pointers to assist the call. The application will be impacted by a performance penalty. In an XPLINK(NOCALLBACK) compilation, if a function pointer is declared using the `__callback` qualifier keyword, the compiler will insert fix-up code to assist the call. For more information on this keyword, see *z/OS XL C/C++ Language Reference*.

Note: In LP64 mode, the only linkage supported is XPLINK. Do not use XPLINK(CALLBACK) in LP64 mode.

GUARD | NOGUARD

DEFAULT: GUARD

If you specify NOGUARD, the compiler generates an explicit check for stack overflow, which enables the storage run-time option. Using NOGUARD causes a performance degradation at run time, even if you do not use the Language Environment run-time STORAGE option.

OSCALL(NOSTACK| UPSTACK| DOWNSTACK)
DEFAULT: NOSTACK

This suboption directs the compiler to use the linkage (OS_NOSTACK, OS_UPSTACK, or OS_DOWNSTACK) as specified in this suboption for any **#pragma linkage(identifier, OS)** calls in your application.

Note: The OSCALL suboption should only be specified when the ILP32 compiler option is also specified.

This value causes the compiler to use the following linkage wherever linkage `OS` is specified by **#pragma linkage** in C, or the `extern` keyword in C++:

OSCALL suboption	Linkage specified in #pragma linkage/extern keyword
NOSTACK	OS_NOSTACK or OS31_NOSTACK (equivalent)
UPSTACK	OS_UPSTACK
DOWNSTACK	OS_DOWNSTACK

For example, since the default of this option is NOSTACK, any **#pragma linkage(identifier,OS)** in C code, works just as if **#pragma linkage(identifier,OS31_NOSTACK)** had been specified.

The abbreviated form of this suboption is OSCALL(N | U | D).

This suboption only applies to routines that are referenced but not defined in the compilation unit.

STOREARGS| NOSTOREARGS
DEFAULT: NOSTOREARGS

If you specify the STOREARGS suboption, the compiler generates code to store arguments that are normally only passed in registers, into the caller's argument area. This facilitates debugging using storage dumps. Use this suboption in conjunction with the BACKCHAIN suboption to make storage dumps more useful.

Note that the values in the argument area may be modified by the called function.

The abbreviated form of this suboption is STOR.

Usage

Using the XPLINK option increases the performance of C/C++ routines by reducing linkage overhead and by passing function call parameters in registers. It supports both reentrant and non-reentrant code, as well as calls to functions exported from DLLs.

The extra performance linkage resulting from XPLINK is a common linkage convention for C and C++. Therefore, it is possible for a C function pointer to reference a non-″extern C″ C++ function. It is also possible for a non-″extern C″ C++ function to reference a C function pointer. With this linkage, casting integers to function pointers is the same as on other platforms such as AIX, making it easier to port applications to z/OS using the C/C++ compiler.

You can not bind XPLINK object decks together with non-XPLINK object decks, with the exception of object decks using OS_UPSTACK or OS_NOSTACK. XPLINK parts of an application can work with non-XPLINK parts across DLL and `fetch()` boundaries.

When compiling using the XPLINK option, the compiler uses the following options as defaults:

- CSECT()
- GOFF
- LONGNAME
- NORENT

You may override these options. However, the XPLINK option requires the GOFF option. If you specify the NOGOFF option, the compiler issues a warning message and promotes the option to GOFF.

In addition, the XPLINK option requires that the value of ARCH must be 2 or greater. The compiler issues a message if you specify ARCH(0) or ARCH(1) with XPLINK and forces the value of ARCH to be 2.

Note: When using XPLINK and source files with duplicate file names, the linker may emit an error and discard one of the code sections. In this case, turn off the CSECT option by specifying NOCSECT.

To build a non-XPLINK C++ application, you can use the Standard C++ Library. It only supports dynamic binding by linking side-decks from CEE.SCEELIB(C128N). In addition, for iostream classes, you can either link the USL iostream Class Library objects from CBC.SCLBCPP or use the side-deck from CBC.SCLBSID(IOSTREAM) for the DLL version. If you are using USL iostream classes, you must ensure that the iostream header files are resolved from CEE.SCEEH.H.

To get the proper data set allocation at prelink/link time, the following **c++** or **cxx** environment variables, if exported, should include the required concatenations. If they are unset, these variables take the default values, which already include the concatenations.

For static binding with USL iostream objects:

- `_CXX_PSYSIX="{_CXX_PLIB_PREFIX}.SCEELIB(C128N)"`
- `_CXX_PSYSLIB="{_CXX_PLIB_PREFIX}.SCEEOBJ:{_CXX_PLIB_PREFIX}.SCEECPP:{_CXX_CLASSLIB_PREFIX}.SCLBCPP"`
- `_CXX_LSYSLIB="{_CXX_PLIB_PREFIX}.SCEELKEX:{_CXX_PLIB_PREFIX}.SCEELKED"`

For USL iostream DLL:

- `_CXX_PSYSIX="{_CXX_CLASSLIB_PREFIX}.SCLBSID(IOSTREAM):{_CXX_PLIB_PREFIX}.SCEELIB(C128N)"`
- `_CXX_PSYSLIB="{_CXX_PLIB_PREFIX}.SCEEOBJ:{_CXX_PLIB_PREFIX}.SCEECPP"`
- `_CXX_LSYSLIB="{_CXX_PLIB_PREFIX}.SCEELKEX:{_CXX_PLIB_PREFIX}.SCEELKED"`

For building without USL iostream DLL:

- `_CXX_PSYSIX="{_CXX_PLIB_PREFIX}.SCEELIB(C128N)"`
- `_CXX_PSYSLIB="{_CXX_PLIB_PREFIX}.SCEEOBJ:{_CXX_PLIB_PREFIX}.SCEECPP"`
- `_CXX_LSYSLIB="{_CXX_PLIB_PREFIX}.SCEELKEX:{_CXX_PLIB_PREFIX}.SCEELKED"`

IPA effects

The IPA Compile step generates information for the IPA Link step. The IPA information in an IPA object file is always generated using the XOBJ format.

This option affects the IPA optimized object module that is generated by specifying the IPA(OBJONLY) option. The object format used to encode this object depends on the GOFF option.

The XPLINK option also affects the regular object module if you request one by specifying the IPA(OBJECT) option. The object format used to encode the regular object depends on the GOFF option.

The IPA Link step accepts the XPLINK option, but ignores it. This is because the linkage convention for a particular subprogram is set during source analysis based on the compile options and #pragmas. It is not possible to change this during the IPA Link step.

The IPA Link step links and merges the application code. All symbol definition and references are checked for compatible attributes, and subprogram calls are checked for compatible linkage conventions. If incompatibilities are found, a diagnostic message is issued and processing is terminated.

The IPA Link step next optimizes the application code, and then divides it into sections for code generation. Each of these sections is a partition. The IPA Link step uses information from the IPA Compile step to determine if a subprogram can be placed in a particular partition. Only compatible subprograms are included in a given partition.

The value of the XPLINK option for a partition is set to the value of the first subprogram that is placed in the partition. The partition map sections of the IPA Link step listing and the object module display the value of the XPLINK option.

Partitions with the XPLINK option are always generated with the GOFF option.

Predefined macros

__XPLINK__ is predefined to a value of 1 when the XPLINK compiler option is in effect; otherwise, it is undefined.

XREF | NOXREF

Category

Listings, messages and compiler information

Pragma equivalent

#pragma options(xref) (C only), **#pragma options(noxref)** (C only)

Purpose

Produces a compiler listing that includes a cross-reference listing of all identifiers.

Syntax

```
      ┌─NOXR──────────────┐
►►────┴─XR──┬──────────┬──┴──────────────────────────────────────────────────►◄
            └─(FULL)───┘
```

Defaults

NOXREF

In the z/OS UNIX System Services environment, this option is turned on by specifying -V when using the **c89**, **cc**, or **c++** commands.

Parameters

FULL
: (Only for C++) Reports all identifiers in the program. When XR is specified without FULL, the report is generated only for the referenced symbols.

Usage

The XREF option generates a cross reference listing that shows file definition, line definition, reference, and modification information for each symbol. It also generates the External Symbol Cross Reference and Static Map.

IPA effects

During the IPA Compile step, the compiler saves symbol storage offset information in the IPA object file as follows:

- For C, if you specify the XREF, IPA(ATTRIBUTE), or IPA(XREF) options or the **#pragma options(XREF)**
- For C++, if you specify the ATTR, XREF, IPA(ATTRIBUTE), or IPA(XREF) options

If regular object code or data is produced using the IPA(OBJECT) option, the cross reference sections of the compile listing will be controlled by the ATTR and XREF options.

If you specify the ATTR or XREF options for the IPA Link step, it generates External Symbol Cross Reference and Static Map listing sections for each partition.

The IPA Link step creates a Storage Offset listing section if during the IPA Compile step you requested the additional symbol storage offset information for your IPA objects.

Predefined macros

None.

Using the z/OS XL C compiler listing

If you select the SOURCE or LIST option, the compiler creates a listing that contains information about the source program and the compilation. If the compilation terminates before reaching a particular stage of processing, the compiler does not generate corresponding parts of the listing. The listing contains standard information that always appears, together with optional information that is supplied by default or specified through compiler options.

In an interactive environment you can also use the TERMINAL option to direct all compiler diagnostic messages to your terminal. The TERMINAL option directs only the diagnostic messages part of the compiler listing to your terminal.

Note: Although the compiler listing is for your use, it is not a programming interface and is subject to change.

IPA considerations

The listings that the IPA Compile step produces are basically the same as those that a regular compilation produces. Any differences are noted throughout this section.

The IPA Link step listing has a separate format from the other compiler listings. Many listing sections are similar to those that are produced by a regular compilation or the IPA Compile step with the IPA(OBJECT) option specified. Refer to “Using the IPA Link step listing” on page 317 for information about IPA Link step listings.

Example of a C compiler listing

Figure 11 shows an example of a C compiler listing.

```
15694A01 V1.11 z/OS XL C                 'CBC.SCCNSAM(CCNUAAM)'                    04/29/2009 04:55:12   Page     1


                                       * * * * *   P R O L O G   * * * * *

  Compile Time Library . . . . . . : 410B0000
  Command options:
     Program name. . . . . . . . . : 'CBC.SCCNSAM(CCNUAAM)'
     Compiler options. . . . . . . : *NOGONUMBER *NOALIAS    *NORENT     *TERMINAL   *NOUPCONV   *SOURCE     *LIST
                                   : *XREF       *AGG        *NOPPONLY   *NOEXPMAC   *NOSHOWINC  *NOOFFSET   *MEMORY     *NOSSCOMM
                                   : *NOSHOWMACROS           *SKIPSRC(SHOW)          *NOREPORT   *NOMAKEDEP  *PREFETCH
                                   : *NOLONGNAME *START      *EXECOPS    *ARGPARSE   *NOEXPORTALL*NODLL(NOCALLBACKANY)
                                   : *NOLIBANSI  *NOWSIZEOF  *REDIR      *ANSIALIAS  *DIGRAPH    *NOROCONST  *ROSTRING
                                   : *TUNE(5)    *ARCH(5)    *SPILL(128) *MAXMEM(2097152)        *NOCOMPACT
                                   : *TARGET(LE,CURRENT)     *FLAG(I)    *NOTEST(SYM,BLOCK,LINE,PATH,HOOK)   *NOOPTIMIZE
                                   : *INLINE(AUTO,REPORT,100,1000)       *NESTINC(255)           *BITFIELD(UNSIGNED)
                                   : *NOCHECKOUT(NOPPTRACE,PPCHECK,GOTO,ACCURACY,PARM,NOENUM,
                                   :             NOEXTERN,TRUNC,INIT,NOPORT,GENERAL,CAST)
                                   : *NODFP
                                   : *FLOAT(HEX,FOLD,AFP(NOVOLATILE))    *STRICT     *NOIGNERRNO *NOINITAUTO
                                   : *NOCOMPRESS *NOSTRICT_INDUCTION     *AGGRCOPY(NOOVERLAP)    *CHARS(UNSIGNED)
                                   : *NOCSECT
                                   : *NOEVENTS
                                   : *ASSERT(RESTRICT)
                                   : *OBJECT
                                   : *NOGENASM
                                   : *NOOPTFILE
                                   : *NOSERVICE
                                   : *NOOE
                                   : *NOIPA
                                   : *SEARCH(//'CEE.SCEEH.+')
                                   : *NOLSEARCH
                                   : *NOLOCALE   *HALT(16)   *PLIST(HOST)
                                   : *NOCONVLIT
                                   : *NOASCII
                                   : *NOGOFF     *ILP32      *NOWARN64   *NOHGPR(NOPRESERVE)     *NOHOT      *NOMETAL    *NOARMODE
                                   : *NOXPLINK(NOBACKCHAIN,NOSTOREARGS,NOCALLBACK,GUARD,OSCALL(NOSTACK))
                                   : *ENUMSIZE(SMALL)
                                   : *NOHALTONMSG
                                   : *NOSUPPRESS
                                   : *NORTCHECK
                                   : *NODEBUG
                                   : *NOSQL
                                   : *NOCICS
                                   : *UNROLL(AUTO)
     Version Macros. . . . . . . . : __COMPILER_VER__=0x410B0000 __LIBREL__=0x410B0000 __TARGET_LIB__=0x410B0000
     Language level. . . . . . . . : *EXTENDED
     Source margins. . . . . . . . :
       Varying length. . . . . . . : 1 - 32760
       Fixed length. . . . . . . . : 1 - 32760
     Sequence columns. . . . . . . :
       Varying length. . . . . . . : none
       Fixed length. . . . . . . . : none

                                * * * * *   E N D   O F   P R O L O G   * * * * *
```

Figure 11. Example of a C listing (Part 1 of 34)

```
15694A01 V1.11 z/OS XL C                 'CBC.SCCNSAM(CCNUAAM)'                    04/29/2009 04:55:12   Page     2


                                           * * * * *   S O U R C E   * * * * *

  LINE  STMT
SEQNBR INCNO
              *...+....1....+....2....+....3....+....4....+....5....+....6....+....7....+....8....+....9....+..*
     1       |#include <stdio.h>                                                                                |
  1
     2       |                                                                                                  |
  2
     3       |#include "ccnuaan.h"                                                                              |
  3
     4       |                                                                                                  |
  4
     5       |void convert(double);                                                                             |
  5
     6       |                                                                                                  |
  6
     7       |int main(int argc, char **argv)                                                                   |
  7
     8       |{                                                                                                 |
  8
     9       |    double c_temp;                                                                                |
  9
    10       |                                                                                                  |
 10
    11     1 |    if (argc == 1) {  /* get Celsius value from stdin */                                          |
 11
    12       |       int ch;                                                                                    |
 12
    13       |                                                                                                  |
 13
    14     2 |       printf("Enter Celsius temperature: \n");                                                   |
 14
    15       |                                                                                                  |
 15
    16     3 |       if (scanf("%f", &c_temp) != 1) {                                                           |
 16
    17     4 |          printf("You must enter a valid temperature\n");                                         |
 17
    18       |       }                                                                                          |
 18
    19       |       else {                                                                                     |
 19
    20     5 |          convert(c_temp);                                                                        |
 20
    21       |       }                                                                                          |
 21
    22       |    }                                                                                             |
 22
    23       |    else {  /* convert the command-line arguments to Fahrenheit */                                |
 23
    24       |       int i;                                                                                     |
 24
    25       |                                                                                                  |
 25
    26     6 |       for (i = 1; i < argc; ++i) {                                                               |
 26
    27     7 |           if (sscanf(argv[i], "%f", &c_temp) != 1)                                               |
 27
    28     8 |              printf("%s is not a valid temperature\n",argv[i]);                                  |
 28
    29       |           else                                                                                   |
 29
    30     9 |              convert(c_temp);                                                                    |
 30
    31       |       }                                                                                          |
 31
    32       |    }                                                                                             |
 32
    33    10 |    return 0;                                                                                     |
 33
    34       |}                                                                                                 |
 34
    35       |                                                                                                  |
 35
    36       |void convert(double c_temp) {                                                                     |
 36
    37    11 |   double f_temp = (c_temp * CONV + OFFSET);                                                      |
 37
    38    12 |   printf("%5.2f Celsius is %5.2f Fahrenheit\n",c_temp, f_temp);                                  |
 38
    39       |}                                                                                                 |
 39
```

Figure 11. Example of a C listing (Part 2 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                 04/29/2009 04:55:12   Page     3

                         * * * * *   I N C L U D E S   * * * * *

 INCLUDE FILES  ---  FILE#   NAME

                       1   CEE.SCEEH.H(STDIO)
                       2   CEE.SCEEH.H(FEATURES)
                       3   CEE.SCEEH.SYS.H(TYPES)
                       4   CBC.SCCNSAM(CCNUAAN)

                         * * * * *   E N D   O F   I N C L U D E S   * * * * *
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                 04/29/2009 04:55:12   Page     4

                                 * * * * *   C R O S S   R E F E R E N C E   L I S T I N G   * * * * *

 IDENTIFIER          DEFINITION      ATTRIBUTES
                                     <SEQNBR>-<FILE NO>:<FILE LINE NO>

 ___valist           1-1:142         Class = typedef, Length = 8
                                     Type = array[2] of pointer to unsigned char
                                     1-1:145, 1-1:456, 1-1:457, 1-1:459

 __abend             1-1:895         Type = struct with no tag in union at offset 0

 __alloc             1-1:905         Type = struct with no tag in union at offset 0

 __amrc_noseek_to_seek
                     1-1:938         Type = unsigned char in struct __amrctype at offset 232

 __amrc_pad          1-1:940         Type = array[23] of unsigned char in struct __amrctype at offset 233

 __amrc_ptr          1-1:947         Class = typedef, Length = 4
                                     Type = pointer to struct __amrctype

 __amrc_type         1-1:943         Class = typedef, Length = 256
                                     Type = struct __amrctype
                                     1-1:947

 __amrctype          1-1:877         Class = struct tag

 __amrc2_ptr         1-1:961         Class = typedef, Length = 4
                                     Type = pointer to struct __amrc2type

 __amrc2_type        1-1:957         Class = typedef, Length = 32
                                     Type = struct __amrc2type
                                     1-1:961

 __amrc2type         1-1:952         Class = struct tag

 __blksize           1-1:729         Type = unsigned long in struct __fileData at offset 8

 __bufPtr            1-1:78          Type = pointer to unsigned char in struct __file at offset 0

 __cntlinterpret     1-1:83          Type = unsigned int:1 in struct __file at offset 20(0)

 __code              1-1:906         Type = union with no tag in struct __amrctype at offset 0

 __countIn           1-1:79          Type = long in struct __file at offset 4

 __countOut          1-1:80          Type = long in struct __file at offset 8

 __cusp              1-1:201         Class = typedef, Length = 4
                                     Type = pointer to const unsigned short

 __device            1-1:728         Type = enum with no tag in struct __fileData at offset 4

 __device_specific   1-1:745         Type = union with no tag in struct __fileData at offset 16

 __device_t          1-1:663         Class = typedef, Length = 1
                                     Type = enum with no tag
```

Figure 11. Example of a C listing (Part 3 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                04/29/2009 04:55:12   Page     5


                              * * * * *  C R O S S   R E F E R E N C E   L I S T I N G  * * * * *

 IDENTIFIER          DEFINITION      ATTRIBUTES
                                     <SEQNBR>-<FILE NO>:<FILE LINE NO>
                                     1-1:728

 __disk              1-1:643         Class = enumeration constant: 0, Length = 4
                                     Type = int

 __disk              1-1:743         Type = struct with no tag in union at offset 0

 __disk_access_method
                     1-1:740         Type = unsigned char in struct at offset 2

 __disk_noseek_to_seek
                     1-1:741         Type = unsigned char in struct at offset 3

 __disk_reserve      1-1:742         Type = array[2] of long in struct at offset 4

 __disk_vsam_type    1-1:739         Type = unsigned short in struct at offset 0

 __dsname            1-1:746         Type = pointer to unsigned char in struct __fileData at offset 28

 __dsorgConcat       1-1:703         Type = unsigned int:1 in struct __fileData at offset 1(3)

 __dsorgHiper        1-1:705         Type = unsigned int:1 in struct __fileData at offset 1(5)

 __dsorgHFS          1-1:710         Type = unsigned int:1 in struct __fileData at offset 2(0)

 __dsorgMem          1-1:704         Type = unsigned int:1 in struct __fileData at offset 1(4)

 __dsorgPDSdir       1-1:701         Type = unsigned int:1 in struct __fileData at offset 1(1)

 __dsorgPDSmem       1-1:700         Type = unsigned int:1 in struct __fileData at offset 1(0)

 __dsorgPDSE         1-1:717         Type = unsigned int:1 in struct __fileData at offset 2(7)

 __dsorgPO           1-1:699         Type = unsigned int:1 in struct __fileData at offset 0(7)

 __dsorgPS           1-1:702         Type = unsigned int:1 in struct __fileData at offset 1(2)

 __dsorgTemp         1-1:706         Type = unsigned int:1 in struct __fileData at offset 1(6)

 __dsorgVSAM         1-1:707         Type = unsigned int:1 in struct __fileData at offset 1(7)

 __dummy             1-1:648         Class = enumeration constant: 6, Length = 4
                                     Type = int

 __error             1-1:889         Type = int in union at offset 0

 __error2            1-1:953         Type = int in struct __amrc2type at offset 0

 __fcb_ascii         1-1:84          Type = unsigned int:1 in struct __file at offset 20(1)

 __fcb_orientation   1-1:85          Type = unsigned int:2 in struct __file at offset 20(2)

 __fcbgetc           1-1:81          Type = pointer to function returning int in struct __file at offset 12
```

Figure 11. Example of a C listing (Part 4 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                04/29/2009 04:55:12   Page     6


                                 * * * * *   C R O S S   R E F E R E N C E   L I S T I N G   * * * * *

 IDENTIFIER          DEFINITION      ATTRIBUTES
                                     <SEQNBR>-<FILE NO>:<FILE LINE NO>

 __fcbputc           1-1:82          Type = pointer to function returning int in struct __file at offset 16

 __fdbk              1-1:900         Type = unsigned char in struct at offset 3

 __fdbk_fill         1-1:897         Type = unsigned char in struct at offset 0

 __feedback          1-1:901         Type = struct with no tag in union at offset 0

 __ffile             1-1:88          Class = struct tag
                                     1-1:93, 1-1:99

 __file              1-1:73          Class = struct tag
                                     1-1:74, 1-1:75, 1-1:90

 __fileptr           1-1:955         Type = pointer to struct __ffile in struct __amrc2type at offset 4

 __fileData          1-1:691         Class = struct tag
                                     1-1:750

 __fill              1-1:864         Type = unsigned int in struct at offset 0

 __filler1           1-1:626         Type = unsigned char in struct __S99emparms at offset 3

 __fill2             1-1:920         Type = array[2] of unsigned int in struct at offset 136

 __fp                1-1:90          Type = pointer to struct __file in struct __ffile at offset 0

 __fpos_elem         1-1:104         Type = array[8] of long in struct __fpos_t at offset 0

 __fpos_t            1-1:103         Class = struct tag
                                     1-1:107

 __ftncd             1-1:899         Type = unsigned char in struct at offset 2

 __func__            36-0:36         Class = static, Length = 8
                                     Type = array[8] of const unsigned char in function convert

 __func__            8-0:8           Class = static, Length = 5
                                     Type = array[5] of const unsigned char in function main

 __hfs               1-1:655         Class = enumeration constant: 9, Length = 4
                                     Type = int

 __hiperspace        1-1:656         Class = enumeration constant: 10, Length = 4
                                     Type = int

 __last_op           1-1:913         Type = unsigned int in struct __amrctype at offset 8

 __len               1-1:916         Type = unsigned int in struct at offset 4

 __len_fill          1-1:915         Type = unsigned int in struct at offset 0
```

Figure 11. Example of a C listing (Part 5 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                04/29/2009 04:55:12   Page     7


                                 * * * * *   C R O S S   R E F E R E N C E   L I S T I N G   * * * * *

 IDENTIFIER          DEFINITION      ATTRIBUTES
                                     <SEQNBR>-<FILE NO>:<FILE LINE NO>
 __maxreclen         1-1:730         Type = unsigned long in struct __fileData at offset 12

 __memory            1-1:654         Class = enumeration constant: 8, Length = 4
                                     Type = int

 __modeflag          1-1:716         Type = unsigned int:4 in struct __fileData at offset 2(3)

 __msg               1-1:923         Type = struct with no tag in struct __amrctype at offset 12

 __msgfile           1-1:651         Class = enumeration constant: 7, Length = 4
                                     Type = int

 __msgrtn            1-1:659         Class = enumeration constant: 11, Length = 4
                                     Type = int

 __openmode          1-1:715         Type = unsigned int:2 in struct __fileData at offset 2(1)

 __other             1-1:662         Class = enumeration constant: 255, Length = 4
                                     Type = int

 __parmr0            1-1:918         Type = unsigned int in struct at offset 128

 __parmr1            1-1:919         Type = unsigned int in struct at offset 132

 __printer           1-1:645         Class = enumeration constant: 2, Length = 4
                                     Type = int

 __rc                1-1:894         Type = unsigned short in struct at offset 2

 __rc                1-1:898         Type = unsigned char in struct at offset 1

 __recfmASA          1-1:697         Type = unsigned int:1 in struct __fileData at offset 0(5)

 __recfmB            1-1:722         Type = unsigned int:1 in struct __fileData at offset 3(4)

 __recfmBlk          1-1:696         Type = unsigned int:1 in struct __fileData at offset 0(4)

 __recfmF            1-1:692         Type = unsigned int:1 in struct __fileData at offset 0(0)

 __recfmM            1-1:698         Type = unsigned int:1 in struct __fileData at offset 0(6)

 __recfmS            1-1:695         Type = unsigned int:1 in struct __fileData at offset 0(3)

 __recfmU            1-1:694         Type = unsigned int:1 in struct __fileData at offset 0(2)

 __recfmV            1-1:693         Type = unsigned int:1 in struct __fileData at offset 0(1)

 __recnum            1-1:865         Type = unsigned int in struct at offset 4

 __reserved          1-1:609         Type = unsigned char in struct __S99rbx at offset 16

 __reserved          1-1:956         Type = array[6] of int in struct __amrc2type at offset 8
```

Figure 11. Example of a C listing (Part 6 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                04/29/2009 04:55:12   Page     8


                                 * * * * *   C R O S S   R E F E R E N C E   L I S T I N G   * * * * *

 IDENTIFIER          DEFINITION       ATTRIBUTES
                                      <SEQNBR>-<FILE NO>:<FILE LINE NO>
 __reserve2          1-1:723          Type = unsigned int:3 in struct __fileData at offset 3(5)

 __reserve4          1-1:747          Type = pointer to void in struct __fileData at offset 32

 __reserv1           1-1:631          Type = int in struct __S99emparms at offset 20

 __reserv2           1-1:617          Type = int in struct __S99rbx at offset 32

 __reserv2           1-1:632          Type = int in struct __S99emparms at offset 24

 __rplfdbwd          1-1:926          Type = array[4] of unsigned char in struct __amrctype at offset 220

 __rrds_key_type     1-1:870          Class = typedef, Length = 8
                                      Type = struct with no tag

 __str               1-1:917          Type = array[120] of unsigned char in struct at offset 8

 __str2              1-1:921          Type = array[64] of unsigned char in struct at offset 144

 __svc99_error       1-1:904          Type = unsigned short in struct at offset 2

 __svc99_info        1-1:903          Type = unsigned short in struct at offset 0

 __syscode           1-1:893          Type = unsigned short in struct at offset 0

 __tape              1-1:646          Class = enumeration constant: 3, Length = 4
                                      Type = int

 __tdq               1-1:647          Class = enumeration constant: 5, Length = 4
                                      Type = int

 __terminal          1-1:644          Class = enumeration constant: 1, Length = 4
                                      Type = int

 __vsam              1-1:736          Type = struct with no tag in union at offset 0

 __vsam_keylen       1-1:734          Type = unsigned long in struct at offset 4

 __vsam_type         1-1:733          Type = unsigned short in struct at offset 0

 __vsam_RKP          1-1:735          Type = unsigned long in struct at offset 8

 __vsamEA            1-1:721          Type = unsigned int:1 in struct __fileData at offset 3(3)

 __vsamRLS           1-1:719          Type = unsigned int:3 in struct __fileData at offset 3(0)

 __EMBUFP            1-1:630          Type = pointer to void in struct __S99emparms at offset 16

 __EMCPPLP           1-1:629          Type = pointer to void in struct __S99emparms at offset 12

 __EMFUNCT           1-1:623          Type = unsigned char in struct __S99emparms at offset 0

 __EMIDNUM           1-1:624          Type = unsigned char in struct __S99emparms at offset 1
```

Figure 11. Example of a C listing (Part 7 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                04/29/2009 04:55:12   Page     9


                               * * * * *   C R O S S   R E F E R E N C E   L I S T I N G   * * * * *

 IDENTIFIER          DEFINITION      ATTRIBUTES
                                     <SEQNBR>-<FILE NO>:<FILE LINE NO>

 __EMNMSGBK          1-1:625         Type = unsigned char in struct __S99emparms at offset 2

 __EMRETCOD          1-1:628         Type = int in struct __S99emparms at offset 8

 __EMS99RBP          1-1:627         Type = pointer to void in struct __S99emparms at offset 4

 __FILEP             1-1:93          Class = typedef, Length = 4
                                     Type = pointer to struct __ffile

 __RBA               1-1:907         Type = unsigned int in struct __amrctype at offset 4

 __S99emparms        1-1:622         Class = struct tag
                                     1-1:635

 __S99emparms_t      1-1:635         Class = typedef, Length = 28
                                     Type = struct __S99emparms

 __S99parms          1-1:595         Class = typedef, Length = 20
                                     Type = struct __S99struc
                                     1-1:833

 __S99rbx            1-1:599         Class = struct tag
                                     1-1:620

 __S99rbx_t          1-1:620         Class = typedef, Length = 36
                                     Type = struct __S99rbx

 __S99struc          1-1:578         Class = struct tag
                                     1-1:595

 __S99ECPPL          1-1:608         Type = pointer to void in struct __S99rbx at offset 12

 __S99EERR           1-1:615         Type = unsigned short in struct __S99rbx at offset 28

 __S99EID            1-1:601         Type = array[6] of unsigned char in struct __S99rbx at offset 0

 __S99EINFO          1-1:616         Type = unsigned short in struct __S99rbx at offset 30

 __S99EKEY           1-1:605         Type = unsigned char in struct __S99rbx at offset 9

 __S99EMGSV          1-1:606         Type = unsigned char in struct __S99rbx at offset 10

 __S99EMSGP          1-1:614         Type = pointer to void in struct __S99rbx at offset 24

 __S99ENMSG          1-1:607         Type = unsigned char in struct __S99rbx at offset 11

 __S99EOPTS          1-1:603         Type = unsigned char in struct __S99rbx at offset 7

 __S99ERCF           1-1:612         Type = unsigned char in struct __S99rbx at offset 19

 __S99ERCO           1-1:611         Type = unsigned char in struct __S99rbx at offset 18
```

Figure 11. Example of a C listing (Part 8 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                04/29/2009 04:55:12   Page    10


                                 * * * * *   C R O S S   R E F E R E N C E   L I S T I N G   * * * * *

 IDENTIFIER          DEFINITION       ATTRIBUTES
                                      <SEQNBR>-<FILE NO>:<FILE LINE NO>
 __S99ERES           1-1:610          Type = unsigned char in struct __S99rbx at offset 17

 __S99ERROR          1-1:584          Type = unsigned short in struct __S99struc at offset 4

 __S99ESUBP          1-1:604          Type = unsigned char in struct __S99rbx at offset 8

 __S99EVER           1-1:602          Type = unsigned char in struct __S99rbx at offset 6

 __S99EWRC           1-1:613          Type = int in struct __S99rbx at offset 20

 __S99FLAG1          1-1:582          Type = unsigned short in struct __S99struc at offset 2

 __S99FLAG2          1-1:590          Type = unsigned int in struct __S99struc at offset 16

 __S99INFO           1-1:585          Type = unsigned short in struct __S99struc at offset 6

 __S99RBLN           1-1:580          Type = unsigned char in struct __S99struc at offset 0

 __S99S99X           1-1:588          Type = pointer to void in struct __S99struc at offset 12

 __S99TXTPP          1-1:586          Type = pointer to void in struct __S99struc at offset 8

 __S99VERB           1-1:581          Type = unsigned char in struct __S99struc at offset 1

 __XRBA              1-1:932          Type = unsigned long long in struct __amrctype at offset 224

 _gtca                                Class = extern
                                      Type = function returning pointer to const void
                                      1-1:166

 _Gtab                                Class = extern
                                      Type = function returning pointer to pointer to void
                                      1-1:156

 _GETCFUNC           1-1:74           Class = typedef
                                      Type = function returning int
                                      1-1:81

 _PUTCFUNC           1-1:75           Class = typedef
                                      Type = function returning int
                                      1-1:82

 argc                7-0:7            Class = parameter, Length = 4
                                      Type = int in function main
                                      11-0:11, 26-0:26, 26-0:26

 argv                7-0:7            Class = parameter, Length = 4
                                      Type = pointer to pointer to unsigned char in function main
                                      27-0:27, 28-0:28

 c_temp              36-0:36          Class = parameter, Length = 8
                                      Type = double in function convert
                                      37-0:37, 38-0:38
```

Figure 11. Example of a C listing (Part 9 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                04/29/2009 04:55:12   Page    11



                              * * * * *   C R O S S   R E F E R E N C E   L I S T I N G   * * * * *

 IDENTIFIER          DEFINITION       ATTRIBUTES
                                      <SEQNBR>-<FILE NO>:<FILE LINE NO>

 c_temp              9-0:9            Class = auto, Length = 8
                                      Type = double in function main
                                      16-0:16, 20-0:20, 27-0:27, 30-0:30

 ch                  12-0:12          Class = auto, Length = 4
                                      Type = int in function main

 clearerr                             Class = extern
                                      Type = function returning void
                                      1-1:406

 clrmemf                              Class = extern
                                      Type = function returning int
                                      1-1:837

 convert             36-0:36          Class = extern
                                      Type = function returning void
                                      5-0:5, 20-0:20, 30-0:30

 f_temp              37-0:37          Class = auto, Length = 8
                                      Type = double in function convert
                                      37-0:37, 38-0:38

 fclose                               Class = extern
                                      Type = function returning int
                                      1-1:407

 fdelrec                              Class = extern
                                      Type = function returning int
                                      1-1:835

 feof                                 Class = extern
                                      Type = function returning int
                                      1-1:408

 ferror                               Class = extern
                                      Type = function returning int
                                      1-1:409

 fflush                               Class = extern
                                      Type = function returning int
                                      1-1:410

 fgetc                                Class = extern
                                      Type = function returning int
                                      1-1:411

 fgetpos                              Class = extern
                                      Type = function returning int
                                      1-1:412

 fgets                                Class = extern
```

Figure 11. Example of a C listing (Part 10 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                04/29/2009 04:55:12   Page    12


                               * * * * *   C R O S S   R E F E R E N C E   L I S T I N G   * * * * *

 IDENTIFIER          DEFINITION      ATTRIBUTES
                                     <SEQNBR>-<FILE NO>:<FILE LINE NO>
                                     Type = function returning pointer to unsigned char
                                     1-1:413

 fldata                              Class = extern
                                     Type = function returning int
                                     1-1:838

 fldata_t            1-1:750         Class = typedef, Length = 36
                                     Type = struct __fileData
                                     1-1:838

 flocate                             Class = extern
                                     Type = function returning int
                                     1-1:834

 fopen                               Class = extern
                                     Type = function returning pointer to struct __ffile
                                     1-1:414

 fpos_t              1-1:107         Class = typedef, Length = 32
                                     Type = struct __fpos_t
                                     1-1:412, 1-1:427

 fprintf                             Class = extern
                                     Type = function returning int
                                     1-1:416

 fputc                               Class = extern
                                     Type = function returning int
                                     1-1:418

 fputs                               Class = extern
                                     Type = function returning int
                                     1-1:419

 fread                               Class = extern
                                     Type = function returning unsigned int
                                     1-1:420

 freopen                             Class = extern
                                     Type = function returning pointer to struct __ffile
                                     1-1:422

 fscanf                              Class = extern
                                     Type = function returning int
                                     1-1:424

 fseek                               Class = extern
                                     Type = function returning int
                                     1-1:426

 fsetpos                             Class = extern
                                     Type = function returning int
```

Figure 11. Example of a C listing (Part 11 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                04/29/2009 04:55:12   Page    13


                             * * * * *   C R O S S   R E F E R E N C E   L I S T I N G   * * * * *

 IDENTIFIER          DEFINITION       ATTRIBUTES
                                      <SEQNBR>-<FILE NO>:<FILE LINE NO>
                                      1-1:427

 ftell                                Class = extern
                                      Type = function returning long
                                      1-1:428

 fupdate                              Class = extern
                                      Type = function returning unsigned int
                                      1-1:836

 fwrite                               Class = extern
                                      Type = function returning unsigned int
                                      1-1:429

 getc                                 Class = extern
                                      Type = function returning int
                                      1-1:431

 getchar                              Class = extern
                                      Type = function returning int
                                      1-1:432

 gets                                 Class = extern
                                      Type = function returning pointer to unsigned char
                                      1-1:433

 i                   24-0:24          Class = auto, Length = 4
                                      Type = int in function main
                                      26-0:26, 26-0:26, 27-0:27, 28-0:28, 26-0:26, 26-0:26, 26-0:26

 main                7-0:7            Class = extern
                                      Type = function returning int

 perror                               Class = extern
                                      Type = function returning void
                                      1-1:434

 printf                               Class = extern
                                      Type = function returning int
                                      1-1:435, 14-0:14, 17-0:17, 28-0:28, 38-0:38

 putc                                 Class = extern
                                      Type = function returning int
                                      1-1:436

 putchar                              Class = extern
                                      Type = function returning int
                                      1-1:437

 puts                                 Class = extern
                                      Type = function returning int
                                      1-1:438
```

Figure 11. Example of a C listing (Part 12 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                04/29/2009 04:55:12   Page    14


                                 * * * * *   C R O S S   R E F E R E N C E   L I S T I N G   * * * * *

 IDENTIFIER          DEFINITION      ATTRIBUTES
                                     <SEQNBR>-<FILE NO>:<FILE LINE NO>
 remove                              Class = extern
                                     Type = function returning int
                                     1-1:439

 rename                              Class = extern
                                     Type = function returning int
                                     1-1:440

 rewind                              Class = extern
                                     Type = function returning void
                                     1-1:441

 scanf                               Class = extern
                                     Type = function returning int
                                     1-1:442, 16-0:16

 setbuf                              Class = extern
                                     Type = function returning void
                                     1-1:443

 setvbuf                             Class = extern
                                     Type = function returning int
                                     1-1:444

 size_t              1-1:58          Class = typedef, Length = 4
                                     Type = unsigned int
                                     1-1:420, 1-1:420, 1-1:421, 1-1:429, 1-1:429, 1-1:429, 1-1:445, 1-1:834, 1-
1:836, 1-1:836

 sprintf                             Class = extern
                                     Type = function returning int
                                     1-1:447

 sscanf                              Class = extern
                                     Type = function returning int
                                     1-1:450, 27-0:27

 ssize_t             1-1:67          Class = typedef, Length = 4
                                     Type = int

 svc99                               Class = extern
                                     Type = function returning int
                                     1-1:833

 tmpfile                             Class = extern
                                     Type = function returning pointer to struct __ffile
                                     1-1:452

 tmpnam                              Class = extern
                                     Type = function returning pointer to unsigned char
                                     1-1:453

 ungetc                              Class = extern
                                     Type = function returning int
```

Figure 11. Example of a C listing (Part 13 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                04/29/2009 04:55:12   Page    15


                                 * * * * *  C R O S S  R E F E R E N C E  L I S T I N G  * * * * *

 IDENTIFIER         DEFINITION      ATTRIBUTES
                                    <SEQNBR>-<FILE NO>:<FILE LINE NO>
                                    1-1:454

 va_list            1-1:145         Class = typedef, Length = 8
                                    Type = array[2] of pointer to unsigned char

 vfprintf                           Class = extern
                                    Type = function returning int
                                    1-1:455

 vprintf                            Class = extern
                                    Type = function returning int
                                    1-1:457

 vsprintf                           Class = extern
                                    Type = function returning int
                                    1-1:458

 FILE               1-1:99          Class = typedef, Length = 4
                                    Type = struct __ffile
                                    1-1:406, 1-1:407, 1-1:408, 1-1:409, 1-1:410, 1-1:411, 1-1:412, 1-1:413, 1-
1:414, 1-1:416,
                                    1-1:418, 1-1:419, 1-1:421, 1-1:422, 1-1:423, 1-1:424, 1-1:426, 1-1:427, 1-
1:428, 1-1:430,
                                    1-1:431, 1-1:436, 1-1:441, 1-1:443, 1-1:444, 1-1:452, 1-1:454, 1-1:455, 1-
1:834, 1-1:835,
                                    1-1:836, 1-1:838, 1-1:955

                          * * * * *  E N D  O F  C R O S S  R E F E R E N C E  L I S T I N G  * * * * *
```

Figure 11. Example of a C listing (Part 14 of 34)

```
                                  * * * * *   S T R U C T U R E   M A P S   * * * * *

 ===================================================================================================================================
 | Aggregate map for: union with no tag #1                                                                   Total size: 12 bytes  |
 |.................................................................................................................................|
 |__device_specific                                                                                                                |
 |=================================================================================================================================|
 |      Offset       |      Length       | Member Name                                                                             |
 |    Bytes(Bits)    |    Bytes(Bits)    |                                                                                         |
 |===================|===================|=========================================================================================|
 |        0          |        12         |   __vsam                                                                                |
 |        0          |         2         |     __vsam_type                                                                         |
 |        2          |         2         |     ***PADDING***                                                                       |
 |        4          |         4         |     __vsam_keylen                                                                       |
 |        8          |         4         |     __vsam_RKP                                                                          |
 |        0          |        12         |   __disk                                                                                |
 |        0          |         2         |     __disk_vsam_type                                                                    |
 |        2          |         1         |     __disk_access_method                                                                |
 |        3          |         1         |     __disk_noseek_to_seek                                                               |
 |        4          |         8         |     __disk_reserve[2]                                                                   |
 ===================================================================================================================================

 ===================================================================================================================================
 | Aggregate map for: struct with no tag #2                                                                  Total size: 12 bytes  |
 |.................................................................................................................................|
 |__vsam                                                                                                                           |
 |=================================================================================================================================|
 |      Offset       |      Length       | Member Name                                                                             |
 |    Bytes(Bits)    |    Bytes(Bits)    |                                                                                         |
 |===================|===================|=========================================================================================|
 |        0          |         2         |   __vsam_type                                                                           |
 |        2          |         2         |   ***PADDING***                                                                         |
 |        4          |         4         |   __vsam_keylen                                                                         |
 |        8          |         4         |   __vsam_RKP                                                                            |
 ===================================================================================================================================

 ===================================================================================================================================
 | Aggregate map for: struct with no tag #3                                                                  Total size: 12 bytes  |
 |.................................................................................................................................|
 |__disk                                                                                                                           |
 |=================================================================================================================================|
 |      Offset       |      Length       | Member Name                                                                             |
 |    Bytes(Bits)    |    Bytes(Bits)    |                                                                                         |
 |===================|===================|=========================================================================================|
 |        0          |         2         |   __disk_vsam_type                                                                      |
 |        2          |         1         |   __disk_access_method                                                                  |
 |        3          |         1         |   __disk_noseek_to_seek                                                                 |
 |        4          |         8         |   __disk_reserve[2]                                                                     |
 ===================================================================================================================================

 ===================================================================================================================================
 | Aggregate map for: struct with no tag #4                                                                  Total size: 8 bytes   |
 |.................................................................................................................................|
 |__rrds_key_type                                                                                                                  |
 |=================================================================================================================================|
 |      Offset       |      Length       | Member Name                                                                             |
 |    Bytes(Bits)    |    Bytes(Bits)    |
```

Figure 11. Example of a C listing (Part 15 of 34)

```
15694A01 V1.11 z/OS XL C              'CBC.SCCNSAM(CCNUAAM)'              04/29/2009 04:55:12   Page    17


                                  * * * * *   S T R U C T U R E   M A P S   * * * * *
 |==================|===================|==========================================================================================|
 |        0         |         4         |  __fill                                                                                  |
 |        4         |         4         |  __recnum                                                                                |
 ==================================================================================================================================

 ==================================================================================================================================
 | Aggregate map for: _Packed struct with no tag #4                                            Total size: 8 bytes              |
 |.................................................................................................................................|
 |_Packed __rrds_key_type                                                                                                          |
 |================================================================================================================================|
 |      Offset      |      Length       | Member Name                                                                              |
 |    Bytes(Bits)   |    Bytes(Bits)    |                                                                                          |
 |==================|===================|==========================================================================================|
 |        0         |         4         |  __fill                                                                                  |
 |        4         |         4         |  __recnum                                                                                |
 ==================================================================================================================================

 ==================================================================================================================================
 | Aggregate map for: union with no tag #5                                                     Total size: 4 bytes              |
 |.................................................................................................................................|
 |__code                                                                                                                           |
 |================================================================================================================================|
 |      Offset      |      Length       | Member Name                                                                              |
 |    Bytes(Bits)   |    Bytes(Bits)    |                                                                                          |
 |==================|===================|==========================================================================================|
 |        0         |         4         |  __error                                                                                 |
 |        0         |         4         |  __abend                                                                                 |
 |        0         |         2         |    __syscode                                                                             |
 |        2         |         2         |    __rc                                                                                  |
 |        0         |         4         |  __feedback                                                                              |
 |        0         |         1         |    __fdbk_fill                                                                           |
 |        1         |         1         |    __rc                                                                                  |
 |        2         |         1         |    __ftncd                                                                               |
 |        3         |         1         |    __fdbk                                                                                |
 |        0         |         4         |  __alloc                                                                                 |
 |        0         |         2         |    __svc99_info                                                                          |
 |        2         |         2         |    __svc99_error                                                                         |
 ==================================================================================================================================

 ==================================================================================================================================
 | Aggregate map for: struct with no tag #6                                                    Total size: 4 bytes              |
 |.................................................................................................................................|
 |__abend                                                                                                                          |
 |================================================================================================================================|
 |      Offset      |      Length       | Member Name                                                                              |
 |    Bytes(Bits)   |    Bytes(Bits)    |                                                                                          |
 |==================|===================|==========================================================================================|
 |        0         |         2         |  __syscode                                                                               |
 |        2         |         2         |  __rc                                                                                    |
 ==================================================================================================================================
```

Figure 11. Example of a C listing (Part 16 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                04/29/2009 04:55:12   Page    18


                                  * * * * *   S T R U C T U R E   M A P S   * * * * *

 =====================================================================================================================================
 | Aggregate map for: struct with no tag #7                                                          Total size: 4 bytes           |
 |...................................................................................................................................|
 |__feedback                                                                                                                         |
 |===================================================================================================================================|
 |      Offset        |      Length        | Member Name                                                                             |
 |    Bytes(Bits)     |    Bytes(Bits)     |                                                                                         |
 |====================|====================|=========================================================================================|
 |        0           |         1          |  __fdbk_fill                                                                            |
 |        1           |         1          |  __rc                                                                                   |
 |        2           |         1          |  __ftncd                                                                                |
 |        3           |         1          |  __fdbk                                                                                 |
 =====================================================================================================================================

 =====================================================================================================================================
 | Aggregate map for: struct with no tag #8                                                          Total size: 4 bytes           |
 |...................................................................................................................................|
 |__alloc                                                                                                                            |
 |===================================================================================================================================|
 |      Offset        |      Length        | Member Name                                                                             |
 |    Bytes(Bits)     |    Bytes(Bits)     |                                                                                         |
 |====================|====================|=========================================================================================|
 |        0           |         2          |  __svc99_info                                                                           |
 |        2           |         2          |  __svc99_error                                                                          |
 =====================================================================================================================================

 =====================================================================================================================================
 | Aggregate map for: struct with no tag #9                                                          Total size: 208 bytes         |
 |...................................................................................................................................|
 |__msg                                                                                                                              |
 |===================================================================================================================================|
 |      Offset        |      Length        | Member Name                                                                             |
 |    Bytes(Bits)     |    Bytes(Bits)     |                                                                                         |
 |====================|====================|=========================================================================================|
 |        0           |         4          |  __len_fill                                                                             |
 |        4           |         4          |  __len                                                                                  |
 |        8           |       120          |  __str[120]                                                                             |
 |      128           |         4          |  __parmr0                                                                               |
 |      132           |         4          |  __parmr1                                                                               |
 |      136           |         8          |  __fill2[2]                                                                             |
 |      144           |        64          |  __str2[64]                                                                             |
 =====================================================================================================================================

 =====================================================================================================================================
 | Aggregate map for: struct __amrctype                                                              Total size: 256 bytes         |
 |...................................................................................................................................|
 |__amrc_type                                                                                                                        |
 |*__amrc_ptr                                                                                                                        |
 |===================================================================================================================================|
 |      Offset        |      Length        | Member Name                                                                             |
 |    Bytes(Bits)     |    Bytes(Bits)     |                                                                                         |
 |====================|====================|=========================================================================================|
 |        0           |         4          |  __code                                                                                 |
 |        0           |         4          |    __error                                                                              |
 |        0           |         4          |    __abend                                                                              |
```

Figure 11. Example of a C listing (Part 17 of 34)

```
 15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                04/29/2009 04:55:12   Page    19


                                   * * * * *   S T R U C T U R E   M A P S   * * * * *
 |        0          |         2          |       __syscode                                                                      |
 |        2          |         2          |       __rc                                                                           |
 |        0          |         4          |     __feedback                                                                       |
 |        0          |         1          |       __fdbk_fill                                                                    |
 |        1          |         1          |       __rc                                                                           |
 |        2          |         1          |       __ftncd                                                                        |
 |        3          |         1          |       __fdbk                                                                         |
 |        0          |         4          |     __alloc                                                                          |
 |        0          |         2          |       __svc99_info                                                                   |
 |        2          |         2          |       __svc99_error                                                                  |
 |        4          |         4          |   __RBA                                                                              |
 |        8          |         4          |   __last_op                                                                          |
 |       12          |       208          |   __msg                                                                              |
 |       12          |         4          |     __len_fill                                                                       |
 |       16          |         4          |     __len                                                                            |
 |       20          |       120          |     __str[120]                                                                       |
 |      140          |         4          |     __parmr0                                                                         |
 |      144          |         4          |     __parmr1                                                                         |
 |      148          |         8          |     __fill2[2]                                                                       |
 |      156          |        64          |     __str2[64]                                                                       |
 |      220          |         4          |   __rplfdbwd[4]                                                                      |
 |      224          |         8          |   __XRBA                                                                             |
 |      232          |         1          |   __amrc_noseek_to_seek                                                              |
 |      233          |        23          |   __amrc_pad[23]                                                                     |
 ===================================================================================================================================

 ===================================================================================================================================
 | Aggregate map for: _Packed struct __amrctype                                                         Total size: 256 bytes      |
 |..................................................................................................................................|
 |_Packed __amrc_type                                                                                                               |
 |==================================================================================================================================|
 |      Offset       |      Length       | Member Name                                                                          |
 |    Bytes(Bits)    |    Bytes(Bits)    |                                                                                      |
 |===================|===================|======================================================================================|
 |        0          |         4          |   __code                                                                             |
 |        0          |         4          |     __error                                                                          |
 |        0          |         4          |     __abend                                                                          |
 |        0          |         2          |       __syscode                                                                      |
 |        2          |         2          |       __rc                                                                           |
 |        0          |         4          |     __feedback                                                                       |
 |        0          |         1          |       __fdbk_fill                                                                    |
 |        1          |         1          |       __rc                                                                           |
 |        2          |         1          |       __ftncd                                                                        |
 |        3          |         1          |       __fdbk                                                                         |
 |        0          |         4          |     __alloc                                                                          |
 |        0          |         2          |       __svc99_info                                                                   |
 |        2          |         2          |       __svc99_error                                                                  |
 |        4          |         4          |   __RBA                                                                              |
 |        8          |         4          |   __last_op                                                                          |
 |       12          |       208          |   __msg                                                                              |
 |       12          |         4          |     __len_fill                                                                       |
 |       16          |         4          |     __len                                                                            |
 |       20          |       120          |     __str[120]                                                                       |
 |      140          |         4          |     __parmr0                                                                         |
 |      144          |         4          |     __parmr1                                                                         |
 |      148          |         8          |     __fill2[2]                                                                       |
```

Figure 11. Example of a C listing (Part 18 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'               04/29/2009 04:55:12   Page    20


                                   * * * * *   S T R U C T U R E   M A P S   * * * * *
 |      156          |       64         |    __str2[64]                                                                  |
 |      220          |        4         |  __rplfdbwd[4]                                                                 |
 |      224          |        8         |  __XRBA                                                                        |
 |      232          |        1         |  __amrc_noseek_to_seek                                                         |
 |      233          |       23         |  __amrc_pad[23]                                                                |
 ===================================================================================================================================

 ===================================================================================================================================
 | Aggregate map for: struct __amrc2type                                                           Total size: 32 bytes         |
 |.................................................................................................................................|
 |__amrc2_type                                                                                                                    |
 |*__amrc2_ptr                                                                                                                    |
 |=================================================================================================================================|
 |       Offset      |       Length      | Member Name                                                                          |
 |     Bytes(Bits)   |     Bytes(Bits)   |                                                                                      |
 |===================|===================|======================================================================================|
 |        0          |        4          |  __error2                                                                            |
 |        4          |        4          |  __fileptr                                                                           |
 |        8          |       24          |  __reserved[6]                                                                       |
 ===================================================================================================================================

 ===================================================================================================================================
 | Aggregate map for: _Packed struct __amrc2type                                                   Total size: 32 bytes         |
 |.................................................................................................................................|
 |_Packed __amrc2_type                                                                                                            |
 |=================================================================================================================================|
 |       Offset      |       Length      | Member Name                                                                          |
 |     Bytes(Bits)   |     Bytes(Bits)   |                                                                                      |
 |===================|===================|======================================================================================|
 |        0          |        4          |  __error2                                                                            |
 |        4          |        4          |  __fileptr                                                                           |
 |        8          |       24          |  __reserved[6]                                                                       |
 ===================================================================================================================================

 ===================================================================================================================================
 | Aggregate map for: struct __ffile                                                               Total size: 4 bytes          |
 |.................................................................................................................................|
 |*__FILEP                                                                                                                        |
 |FILE                                                                                                                            |
 |=================================================================================================================================|
 |       Offset      |       Length      | Member Name                                                                          |
 |     Bytes(Bits)   |     Bytes(Bits)   |                                                                                      |
 |===================|===================|======================================================================================|
 |        0          |        4          |  __fp                                                                                |
 ===================================================================================================================================

 ===================================================================================================================================
 | Aggregate map for: _Packed struct __ffile                                                       Total size: 4 bytes          |
 |.................................................................................................................................|
 |_Packed FILE                                                                                                                    |
 |=================================================================================================================================|
 |       Offset      |       Length      | Member Name                                                                          |
 |     Bytes(Bits)   |     Bytes(Bits)   |                                                                                      |
 |===================|===================|======================================================================================|
 |        0          |        4          |  __fp                                                                                |
 ===================================================================================================================================
```

Figure 11. Example of a C listing (Part 19 of 34)

```
15694A01 V1.11 z/OS XL C               'CBC.SCCNSAM(CCNUAAM)'               04/29/2009 04:55:12   Page    21


                                  * * * * *   S T R U C T U R E   M A P S   * * * * *

 ===================================================================================================================================
 | Aggregate map for: struct __file                                                                          Total size: 24 bytes  |
 |=================================================================================================================================|
 |     Offset        |      Length       | Member Name                                                                             |
 |   Bytes(Bits)     |    Bytes(Bits)    |                                                                                         |
 |===================|===================|=========================================================================================|
 |       0           |       4           |  __bufPtr                                                                               |
 |       4           |       4           |  __countIn                                                                              |
 |       8           |       4           |  __countOut                                                                             |
 |      12           |       4           |  __fcbgetc                                                                              |
 |      16           |       4           |  __fcbputc                                                                              |
 |      20           |       0(1)        |  __cntlinterpret                                                                        |
 |      20(1)        |       0(1)        |  __fcb_ascii                                                                            |
 |      20(2)        |       0(2)        |  __fcb_orientation                                                                      |
 |      20(4)        |       3(4)        |  ***PADDING***                                                                          |
 ===================================================================================================================================

 ===================================================================================================================================
 | Aggregate map for: _Packed struct __file                                                                  Total size: 21 bytes  |
 |=================================================================================================================================|
 |     Offset        |      Length       | Member Name                                                                             |
 |   Bytes(Bits)     |    Bytes(Bits)    |                                                                                         |
 |===================|===================|=========================================================================================|
 |       0           |       4           |  __bufPtr                                                                               |
 |       4           |       4           |  __countIn                                                                              |
 |       8           |       4           |  __countOut                                                                             |
 |      12           |       4           |  __fcbgetc                                                                              |
 |      16           |       4           |  __fcbputc                                                                              |
 |      20           |       0(1)        |  __cntlinterpret                                                                        |
 |      20(1)        |       0(1)        |  __fcb_ascii                                                                            |
 |      20(2)        |       0(2)        |  __fcb_orientation                                                                      |
 |      20(4)        |       0(4)        |  ***PADDING***                                                                          |
 ===================================================================================================================================

 ===================================================================================================================================
 | Aggregate map for: struct __fileData                                                                      Total size: 36 bytes  |
 |.................................................................................................................................|
 |fldata_t                                                                                                                         |
 |=================================================================================================================================|
 |     Offset        |      Length       | Member Name                                                                             |
 |   Bytes(Bits)     |    Bytes(Bits)    |                                                                                         |
 |===================|===================|=========================================================================================|
 |       0           |       0(1)        |  __recfmF                                                                               |
 |       0(1)        |       0(1)        |  __recfmV                                                                               |
 |       0(2)        |       0(1)        |  __recfmU                                                                               |
 |       0(3)        |       0(1)        |  __recfmS                                                                               |
 |       0(4)        |       0(1)        |  __recfmBlk                                                                             |
 |       0(5)        |       0(1)        |  __recfmASA                                                                             |
 |       0(6)        |       0(1)        |  __recfmM                                                                               |
 |       0(7)        |       0(1)        |  __dsorgPO                                                                              |
 |       1           |       0(1)        |  __dsorgPDSmem                                                                          |
 |       1(1)        |       0(1)        |  __dsorgPDSdir                                                                          |
 |       1(2)        |       0(1)        |  __dsorgPS                                                                              |
 |       1(3)        |       0(1)        |  __dsorgConcat                                                                          |
 |       1(4)        |       0(1)        |  __dsorgMem                                                                             |
```

Figure 11. Example of a C listing (Part 20 of 34)

```
                              * * * * *   S T R U C T U R E   M A P S   * * * * *
 |       1(5)        |        0(1)        |  __dsorgHiper                                                                          |
 |       1(6)        |        0(1)        |  __dsorgTemp                                                                           |
 |       1(7)        |        0(1)        |  __dsorgVSAM                                                                           |
 |       2           |        0(1)        |  __dsorgHFS                                                                            |
 |       2(1)        |        0(2)        |  __openmode                                                                            |
 |       2(3)        |        0(4)        |  __modeflag                                                                            |
 |       2(7)        |        0(1)        |  __dsorgPDSE                                                                           |
 |       3           |        0(3)        |  __vsamRLS                                                                             |
 |       3(3)        |        0(1)        |  __vsamEA                                                                              |
 |       3(4)        |        0(1)        |  __recfmB                                                                              |
 |       3(5)        |        0(3)        |  __reserve2                                                                            |
 |       4           |        1           |  __device                                                                              |
 |       5           |        3           |  ***PADDING***                                                                         |
 |       8           |        4           |  __blksize                                                                             |
 |      12           |        4           |  __maxreclen                                                                           |
 |      16           |       12           |  __device_specific                                                                     |
 |      16           |       12           |    __vsam                                                                              |
 |      16           |        2           |      __vsam_type                                                                       |
 |      18           |        2           |      ***PADDING***                                                                     |
 |      20           |        4           |      __vsam_keylen                                                                     |
 |      24           |        4           |      __vsam_RKP                                                                        |
 |      16           |       12           |    __disk                                                                              |
 |      16           |        2           |      __disk_vsam_type                                                                  |
 |      18           |        1           |      __disk_access_method                                                              |
 |      19           |        1           |      __disk_noseek_to_seek                                                             |
 |      20           |        8           |      __disk_reserve[2]                                                                 |
 |      28           |        4           |  __dsname                                                                              |
 |      32           |        4           |  __reserve4                                                                            |
 ==================================================================================================================================

 ==================================================================================================================================
 | Aggregate map for: _Packed struct __fileData                                                           Total size: 33 bytes   |
 |................................................................................................................................|
 |_Packed fldata_t                                                                                                                |
 |================================================================================================================================|
 |      Offset       |       Length       | Member Name                                                                            |
 |    Bytes(Bits)    |    Bytes(Bits)     |                                                                                        |
 |===================|====================|========================================================================================|
 |       0           |        0(1)        |  __recfmF                                                                              |
 |       0(1)        |        0(1)        |  __recfmV                                                                              |
 |       0(2)        |        0(1)        |  __recfmU                                                                              |
 |       0(3)        |        0(1)        |  __recfmS                                                                              |
 |       0(4)        |        0(1)        |  __recfmBlk                                                                            |
 |       0(5)        |        0(1)        |  __recfmASA                                                                            |
 |       0(6)        |        0(1)        |  __recfmM                                                                              |
 |       0(7)        |        0(1)        |  __dsorgPO                                                                             |
 |       1           |        0(1)        |  __dsorgPDSmem                                                                         |
 |       1(1)        |        0(1)        |  __dsorgPDSdir                                                                         |
 |       1(2)        |        0(1)        |  __dsorgPS                                                                             |
 |       1(3)        |        0(1)        |  __dsorgConcat                                                                         |
 |       1(4)        |        0(1)        |  __dsorgMem                                                                            |
 |       1(5)        |        0(1)        |  __dsorgHiper                                                                          |
 |       1(6)        |        0(1)        |  __dsorgTemp                                                                           |
 |       1(7)        |        0(1)        |  __dsorgVSAM                                                                           |
 |       2           |        0(1)        |  __dsorgHFS                                                                            |
 |       2(1)        |        0(2)        |  __openmode                                                                            |
```

Figure 11. Example of a C listing (Part 21 of 34)

```
 15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                 04/29/2009 04:55:12   Page    23


                                   * * * * *   S T R U C T U R E   M A P S   * * * * *
|       2(3)        |       0(4)        |  __modeflag                                                                            |
|       2(7)        |       0(1)        |  __dsorgPDSE                                                                           |
|       3           |       0(3)        |  __vsamRLS                                                                             |
|       3(3)        |       0(1)        |  __vsamEA                                                                              |
|       3(4)        |       0(1)        |  __recfmB                                                                              |
|       3(5)        |       0(3)        |  __reserve2                                                                            |
|       4           |       1           |  __device                                                                              |
|       5           |       4           |  __blksize                                                                             |
|       9           |       4           |  __maxreclen                                                                           |
|      13           |      12           |  __device_specific                                                                     |
|      13           |      12           |    __vsam                                                                              |
|      13           |       2           |      __vsam_type                                                                       |
|      15           |       2           |      ***PADDING***                                                                     |
|      17           |       4           |      __vsam_keylen                                                                     |
|      21           |       4           |      __vsam_RKP                                                                        |
|      13           |      12           |    __disk                                                                              |
|      13           |       2           |      __disk_vsam_type                                                                  |
|      15           |       1           |      __disk_access_method                                                              |
|      16           |       1           |      __disk_noseek_to_seek                                                             |
|      17           |       8           |      __disk_reserve[2]                                                                 |
|      25           |       4           |  __dsname                                                                              |
|      29           |       4           |  __reserve4                                                                            |
=================================================================================================================================

=================================================================================================================================
| Aggregate map for: struct __fpos_t                                                                   Total size: 32 bytes     |
|...............................................................................................................................|
|fpos_t                                                                                                                         |
|===============================================================================================================================|
|      Offset       |      Length       | Member Name                                                                            |
|    Bytes(Bits)    |    Bytes(Bits)    |                                                                                        |
|===================|===================|========================================================================================|
|       0           |      32           |  __fpos_elem[8]                                                                        |
=================================================================================================================================

=================================================================================================================================
| Aggregate map for: _Packed struct __fpos_t                                                           Total size: 32 bytes     |
|...............................................................................................................................|
|_Packed fpos_t                                                                                                                 |
|===============================================================================================================================|
|      Offset       |      Length       | Member Name                                                                            |
|    Bytes(Bits)    |    Bytes(Bits)    |                                                                                        |
|===================|===================|========================================================================================|
|       0           |      32           |  __fpos_elem[8]                                                                        |
=================================================================================================================================

=================================================================================================================================
| Aggregate map for: struct __S99emparms                                                               Total size: 28 bytes     |
|...............................................................................................................................|
|__S99emparms_t                                                                                                                 |
|===============================================================================================================================|
|      Offset       |      Length       | Member Name                                                                            |
|    Bytes(Bits)    |    Bytes(Bits)    |                                                                                        |
|===================|===================|========================================================================================|
|       0           |       1           |  __EMFUNCT                                                                             |
|       1           |       1           |  __EMIDNUM                                                                             |
```

Figure 11. Example of a C listing (Part 22 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                04/29/2009 04:55:12   Page    24


                                   * * * * *   S T R U C T U R E   M A P S   * * * * *
|       2            |        1           |  __EMNMSGBK                                                                          |
|       3            |        1           |  __filler1                                                                           |
|       4            |        4           |  __EMS99RBP                                                                          |
|       8            |        4           |  __EMRETCOD                                                                          |
|      12            |        4           |  __EMCPPLP                                                                           |
|      16            |        4           |  __EMBUFP                                                                            |
|      20            |        4           |  __reserv1                                                                           |
|      24            |        4           |  __reserv2                                                                           |
===================================================================================================================================

===================================================================================================================================
| Aggregate map for: _Packed struct __S99emparms                                                   Total size: 28 bytes          |
|.................................................................................................................................|
|_Packed __S99emparms_t                                                                                                           |
|=================================================================================================================================|
|      Offset        |      Length        | Member Name                                                                           |
|    Bytes(Bits)     |    Bytes(Bits)     |                                                                                       |
|=================== |=================== |=======================================================================================|
|       0            |        1           |  __EMFUNCT                                                                            |
|       1            |        1           |  __EMIDNUM                                                                            |
|       2            |        1           |  __EMNMSGBK                                                                           |
|       3            |        1           |  __filler1                                                                            |
|       4            |        4           |  __EMS99RBP                                                                           |
|       8            |        4           |  __EMRETCOD                                                                           |
|      12            |        4           |  __EMCPPLP                                                                            |
|      16            |        4           |  __EMBUFP                                                                             |
|      20            |        4           |  __reserv1                                                                            |
|      24            |        4           |  __reserv2                                                                            |
===================================================================================================================================

===================================================================================================================================
| Aggregate map for: struct __S99rbx                                                               Total size: 36 bytes          |
|.................................................................................................................................|
|__S99rbx_t                                                                                                                       |
|=================================================================================================================================|
|      Offset        |      Length        | Member Name                                                                           |
|    Bytes(Bits)     |    Bytes(Bits)     |                                                                                       |
|=================== |=================== |=======================================================================================|
|       0            |        6           |  __S99EID[6]                                                                          |
|       6            |        1           |  __S99EVER                                                                            |
|       7            |        1           |  __S99EOPTS                                                                           |
|       8            |        1           |  __S99ESUBP                                                                           |
|       9            |        1           |  __S99EKEY                                                                            |
|      10            |        1           |  __S99EMGSV                                                                           |
|      11            |        1           |  __S99ENMSG                                                                           |
|      12            |        4           |  __S99ECPPL                                                                           |
|      16            |        1           |  __reserved                                                                           |
|      17            |        1           |  __S99ERES                                                                            |
|      18            |        1           |  __S99ERCO                                                                            |
|      19            |        1           |  __S99ERCF                                                                            |
|      20            |        4           |  __S99EWRC                                                                            |
|      24            |        4           |  __S99EMSGP                                                                           |
|      28            |        2           |  __S99EERR                                                                            |
|      30            |        2           |  __S99EINFO                                                                           |
|      32            |        4           |  __reserv2                                                                            |
===================================================================================================================================
```

Figure 11. Example of a C listing (Part 23 of 34)

```
15694A01 V1.11 z/OS XL C               'CBC.SCCNSAM(CCNUAAM)'               04/29/2009 04:55:12   Page    25


                                  * * * * *   S T R U C T U R E   M A P S   * * * * *

 ===================================================================================================================================
 | Aggregate map for: _Packed struct __S99rbx                                                              Total size: 36 bytes |
 |.................................................................................................................................|
 |_Packed __S99rbx_t                                                                                                               |
 |=================================================================================================================================|
 |      Offset       |      Length       | Member Name                                                                             |
 |    Bytes(Bits)    |    Bytes(Bits)    |                                                                                         |
 |===================|===================|=========================================================================================|
 |        0          |        6          |  __S99EID[6]                                                                            |
 |        6          |        1          |  __S99EVER                                                                              |
 |        7          |        1          |  __S99EOPTS                                                                             |
 |        8          |        1          |  __S99ESUBP                                                                             |
 |        9          |        1          |  __S99EKEY                                                                              |
 |       10          |        1          |  __S99EMGSV                                                                             |
 |       11          |        1          |  __S99ENMSG                                                                             |
 |       12          |        4          |  __S99ECPPL                                                                             |
 |       16          |        1          |  __reserved                                                                             |
 |       17          |        1          |  __S99ERES                                                                              |
 |       18          |        1          |  __S99ERCO                                                                              |
 |       19          |        1          |  __S99ERCF                                                                              |
 |       20          |        4          |  __S99EWRC                                                                              |
 |       24          |        4          |  __S99EMSGP                                                                             |
 |       28          |        2          |  __S99EERR                                                                              |
 |       30          |        2          |  __S99EINFO                                                                             |
 |       32          |        4          |  __reserv2                                                                              |
 ===================================================================================================================================

 ===================================================================================================================================
 | Aggregate map for: struct __S99struc                                                                    Total size: 20 bytes |
 |.................................................................................................................................|
 |__S99parms                                                                                                                       |
 |=================================================================================================================================|
 |      Offset       |      Length       | Member Name                                                                             |
 |    Bytes(Bits)    |    Bytes(Bits)    |                                                                                         |
 |===================|===================|=========================================================================================|
 |        0          |        1          |  __S99RBLN                                                                              |
 |        1          |        1          |  __S99VERB                                                                              |
 |        2          |        2          |  __S99FLAG1                                                                             |
 |        4          |        2          |  __S99ERROR                                                                             |
 |        6          |        2          |  __S99INFO                                                                              |
 |        8          |        4          |  __S99TXTPP                                                                             |
 |       12          |        4          |  __S99S99X                                                                              |
 |       16          |        4          |  __S99FLAG2                                                                             |
 ===================================================================================================================================

 ===================================================================================================================================
 | Aggregate map for: _Packed struct __S99struc                                                            Total size: 20 bytes |
 |.................................................................................................................................|
 |_Packed __S99parms                                                                                                               |
 |=================================================================================================================================|
 |      Offset       |      Length       | Member Name                                                                             |
 |    Bytes(Bits)    |    Bytes(Bits)    |                                                                                         |
 |===================|===================|=========================================================================================|
 |        0          |        1          |  __S99RBLN                                                                              |
 |        1          |        1          |  __S99VERB                                                                              |
```

Figure 11. Example of a C listing (Part 24 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                04/29/2009 04:55:12   Page    26


                                  * * * * *   S T R U C T U R E   M A P S   * * * * *
 |       2           |       2           |  __S99FLAG1                                                                       |
 |       4           |       2           |  __S99ERROR                                                                       |
 |       6           |       2           |  __S99INFO                                                                        |
 |       8           |       4           |  __S99TXTPP                                                                       |
 |      12           |       4           |  __S99S99X                                                                        |
 |      16           |       4           |  __S99FLAG2                                                                       |
 ===================================================================================================================================
                           * * * * *   E N D   O F   S T R U C T U R E   M A P S   * * * * *
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                04/29/2009 04:55:12   Page    27


                             * * * * *   M E S S A G E   S U M M A R Y   * * * * *

        Total            Informational(00)         Warning(10)         Error(30)           Severe Error(40)

          0                  0                        0                   0                        0
                        * * * * *   E N D   O F   M E S S A G E   S U M M A R Y   * * * * *
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                04/29/2009 04:55:12   Page    28

                      Inline Report (Summary)

   Reason:     P : noinline was specified for this routine
               F : inline was specified for this routine
               C : compact was specified for this routine
               M : This is an inline member routine
               A : Automatic inlining
               - : No reason
   Action:     I : Routine is inlined at least once
               L : Routine is initially too large to be inlined
               T : Routine expands too large to be inlined
               C : Candidate for inlining but not inlined
               N : No direct calls to routine are found in file (no action)
               U : Some calls not inlined due to recursion or parameter mismatch
               - : No action
   Status:     D : Internal routine is discarded
               R : A direct call remains to internal routine (cannot discard)
               A : Routine has its address taken (cannot discard)
               E : External routine (cannot discard)
               - : Status unchanged
   Calls/I       : Number of calls to defined routines / Number inline
   Called/I      : Number of times called / Number of times inlined

   Reason  Action  Status   Size (init)    Calls/I   Called/I     Name

      A      N       E      66    (52)       1/2       0/0       main
      A      I       E      9                0/0       1/2       convert

  Mode = AUTO    Inlining Threshold = 100    Expansion Limit = 1000
```

Figure 11. Example of a C listing (Part 25 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                  04/29/2009 04:55:12   Page    29

                      Inline Report (Call Structure)

   Defined Function     : main
       Calls To(1,2)    : convert (1,2)
    Called From         : 0

   Defined Function     : convert
       Calls To         : 0
    Called From(1,2)    : main (1,2)

15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                  04/29/2009 04:55:12   Page    30

 OFFSET OBJECT CODE        LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G

                           Timestamp and Version Information
 000000  F2F0  F0F9                                  =C'2009'           Compiled Year
 000004  F0F4  F2F9                                  =C'0429'           Compiled Date MMDD
 000008  F0F4  F5F5  F1F2                            =C'045512'         Compiled Time HHMMSS
 00000E  F0F1  F1F1  F0F0                            =C'011100'         Compiler Version

 000014  0074  ****                                  AL2(116),C'...'    Saved Options String
                           Timestamp and Version End

15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)': convert         04/29/2009 04:55:12   Page    31

 OFFSET OBJECT CODE        LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G

                           000001 |       *  #include <stdio.h>
                           000002 |       *
                           000003 |       *  #include "ccnuaan.h"
                           000004 |       *
                           000005 |       *  void convert(double);
                           000006 |       *
                           000007 |       *  int main(int argc, char **argv)
                           000008 |       *  {
                           000009 |       *      double c_temp;
                           000010 |       *
                           000011 |       *      if (argc == 1) {  /* get Celsius value from stdin */
                           000012 |       *         int ch;
                           000013 |       *
                           000014 |       *         printf("Enter Celsius temperature: \n");
```

Figure 11. Example of a C listing (Part 26 of 34)

```
 000015 |       *
                              000016 |       *        if (scanf("%f", &c_temp) != 1) {
                              000017 |       *           printf("You must enter a valid temperature\n");
                              000018 |       *        }
                              000019 |       *        else {
                              000020 |       *           convert(c_temp);
                              000021 |       *        }
                              000022 |       *     }
                              000023 |       *     else {  /* convert the command-line arguments to Fahrenheit */
                              000024 |       *        int i;
                              000025 |       *
                              000026 |       *        for (i = 1; i < argc; ++i) {
                              000027 |       *            if (sscanf(argv[i], "%f", &c_temp) != 1)
                              000028 |       *               printf("%s is not a valid temperature\n",argv[i]);
                              000029 |       *            else
                              000030 |       *               convert(c_temp);
                              000031 |       *        }
                              000032 |       *     }
                              000033 |       *     return 0;
                              000034 |       *  }
                              000035 |       *
                              000036 |       *  void convert(double c_temp) {
  000098                      000036 |        convert  DS       0D
  000098  47F0  F022          000036 |                 B        34(,r15)
  00009C  01C3C5C5                                              CEE eyecatcher
  0000A0  00000100                                              DSA size
  0000A4  00000320                                              =A(PPA1-convert)
  0000A8  47F0  F001          000036 |                 B        1(,r15)
  0000AC  58F0  C31C          000036 |                 L        r15,796(,r12)
  0000B0  184E                000036 |                 LR       r4,r14
  0000B2  05EF                000036 |                 BALR     r14,r15
  0000B4  00000000                                              =F'0'
  0000B8  07F3                000036 |                 BR       r3
  0000BA  90E5  D00C          000036 |                 STM      r14,r5,12(r13)
  0000BE  58E0  D04C          000036 |                 L        r14,76(,r13)
  0000C2  4100  E100          000036 |                 LA       r0,256(,r14)
  0000C6  5500  C314          000036 |                 CL       r0,788(,r12)
  0000CA  4130  F03A          000036 |                 LA       r3,58(,r15)
  0000CE  4720  F014          000036 |                 BH       20(,r15)
  0000D2  58F0  C280          000036 |                 L        r15,640(,r12)
  0000D6  90F0  E048          000036 |                 STM      r15,r0,72(r14)
  0000DA  9210  E000          000036 |                 MVI      0(r14),16
  00011E  5020  D098          000016 |                 ST       r2,#MX_TEMP1(,r13,152)
```

Figure 11. Example of a C listing (Part 27 of 34)

```
 15694A01 V1.11 z/OS XL C                 'CBC.SCCNSAM(CCNUAAM)': convert          04/29/2009 04:55:12   Page    32

  OFFSET OBJECT CODE        LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G

  0000DE  50D0  E004        000036 |                 ST       r13,4(,r14)
  0000E2  18DE              000036 |                 LR       r13,r14
  0000E4                    End of Prolog

  0000E4  C050  0000  011E  000000 |                 LARL     r5,F'286'
  0000EA  5010  D0B8        000036 |                 ST       r1,#SR_PARM_2(,r13,184)
                            000037 |        *     double f_temp = (c_temp * CONV + OFFSET);
  0000EE  5810  D0B8        000037 |                 L        r1,#SR_PARM_2(,r13,184)
  0000F2  6800  1000        000037 |                 LD       f0,c_temp(,r1,0)
  0000F6  6820  5000        000037 |                 LD       f2,+CONSTANT_AREA(,r5,0)
  0000FA  2C02              000037 |                 MDR      f0,f2
  0000FC  6820  5008        000037 |                 LD       f2,+CONSTANT_AREA(,r5,8)
  000100  2A02              000037 |                 ADR      f0,f2
  000102  6000  D0B0        000037 |                 STD      f0,f_temp(,r13,176)
                            000038 |        *     printf("%5.2f Celsius is %5.2f Fahrenheit•n",c_temp, f_temp);
  000106  5810  D0B8        000038 |                 L        r1,#SR_PARM_2(,r13,184)
  00010A  6820  1000        000038 |                 LD       f2,c_temp(,r1,0)
  00010E  6020  D09C        000038 |                 STD      f2,#MX_TEMP2(,r13,156)
  000112  6000  D0A4        000038 |                 STD      f0,#MX_TEMP2(,r13,164)
  000116  58F0  3066        000038 |                 L        r15,=V(PRINTF)(,r3,102)
  00011A  4100  5010        000038 |                 LA       r0,+CONSTANT_AREA(,r5,16)
  00011E  4110  D098        000038 |                 LA       r1,#MX_TEMP2(,r13,152)
  000122  5000  D098        000038 |                 ST       r0,#MX_TEMP2(,r13,152)
  000126  0DEF              000038 |                 BASR     r14,r15
                            000039 |        *  }
  000128                    000039 |         @2L11    DS       0H

  000128                    Start of Epilog
  000128  58D0  D004        000039 |                 L        r13,4(,r13)
  00012C  58E0  D00C        000039 |                 L        r14,12(,r13)
  000130  9825  D01C        000039 |                 LM       r2,r5,28(r13)
  000134  051E              000039 |                 BALR     r1,r14
  000136  0707              000039 |                 NOPR     7

  000138                    Start of Literals
  000138  00000000                                            =V(PRINTF)
  00013C                    End of Literals

                            ***   General purpose registers used: 1101110000001111
                            ***   Floating point  registers used: 1111111100000000
                            ***   Size of register spill area: 128(max) 0(used)
                            ***   Size of dynamic storage: 256
                            ***   Size of executable code: 160
```

Figure 11. Example of a C listing (Part 28 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)': main            04/29/2009 04:55:12   Page    33

 OFFSET OBJECT CODE        LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G

                           000001 |       *  #include <stdio.h>
                           000002 |       *
                           000003 |       *  #include "ccnuaan.h"
                           000004 |       *
                           000005 |       *  void convert(double);
                           000006 |       *
                           000007 |       *  int main(int argc, char **argv)
 000140                    000007 |        main     DS       0D
 000140  47F0  F022        000007 |                 B        34(,r15)
 000144  01C3C5C5                                            CEE eyecatcher
 000148  00000110                                            DSA size
 00014C  000002C0                                            =A(PPA1-main)
 000150  47F0  F001        000007 |                 B        1(,r15)
 000154  58F0  C31C        000007 |                 L        r15,796(,r12)
 000158  184E              000007 |                 LR       r4,r14
 00015A  05EF              000007 |                 BALR     r14,r15
 00015C  00000000                                            =F'0'
 000160  07F3              000007 |                 BR       r3
 000162  90E5  D00C        000007 |                 STM      r14,r5,12(r13)
 000166  58E0  D04C        000007 |                 L        r14,76(,r13)
 00016A  4100  E110        000007 |                 LA       r0,272(,r14)
 00016E  5500  C314        000007 |                 CL       r0,788(,r12)
 000172  4130  F03A        000007 |                 LA       r3,58(,r15)
 000176  4720  F014        000007 |                 BH       20(,r15)
 00017A  58F0  C280        000007 |                 L        r15,640(,r12)
 00017E  90F0  E048        000007 |                 STM      r15,r0,72(r14)
 000182  9210  E000        000007 |                 MVI      0(r14),16
 000186  50D0  E004        000007 |                 ST       r13,4(,r14)
 00018A  18DE              000007 |                 LR       r13,r14
 00018C                    End of Prolog

 00018C  C050  0000  00CA  000000 |                 LARL     r5,F'202'
 000192  5010  D0BC        000007 |                 ST       r1,#SR_PARM_1(,r13,188)
                           000008 |       *  {
                           000009 |       *      double c_temp;
                           000010 |       *
                           000011 |       *      if (argc == 1) {  /* get Celsius value from stdin */
 000196  5810  D0BC        000011 |                 L        r1,#SR_PARM_1(,r13,188)
 00019A  5800  1000        000011 |                 L        r0,argc(,r1,0)
 00019E  A70E  0001        000011 |                 CHI      r0,H'1'
 0001A2  4770  30B2        000011 |                 BNE      @1L1
                           000012 |       *         int ch;
                           000013 |       *
                           000014 |       *         printf("Enter Celsius temperature: \n");
 0001A6  58F0  319A        000014 |                 L        r15,=V(PRINTF)(,r3,410)
 0001AA  4100  5033        000014 |                 LA       r0,+CONSTANT_AREA(,r5,51)
 0001AE  4110  D098        000014 |                 LA       r1,#MX_TEMP1(,r13,152)
 0001B2  5000  D098        000014 |                 ST       r0,#MX_TEMP1(,r13,152)
 0001B6  0DEF              000014 |                 BASR     r14,r15
                           000015 |       *
                           000016 |       *         if (scanf("%f", &c_temp) != 1) {
 0001B8  4100  D0B0        000016 |                 LA       r0,c_temp(,r13,176)
 0001BC  58F0  319E        000016 |                 L        r15,=V(SCANF)(,r3,414)
 0001C0  4120  5050        000016 |                 LA       r2,+CONSTANT_AREA(,r5,80)
 0001C4  4110  D098        000016 |                 LA       r1,#MX_TEMP1(,r13,152)
 0001C8  5020  D098        000016 |                 ST       r2,#MX_TEMP1(,r13,152)
```

Figure 11. Example of a C listing (Part 29 of 34)

```
 OFFSET OBJECT CODE        LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G

 0001CC  5000  D09C        000016 |                 ST       r0,#MX_TEMP1(,r13,156)
 0001D0  0DEF              000016 |                 BASR     r14,r15
 0001D2  180F              000016 |                 LR       r0,r15
 0001D4  A70E  0001        000016 |                 CHI      r0,H'1'
 0001D8  4780  3078        000016 |                 BE       @1L2
                           000017 |        *           printf("You must enter a valid temperature\n");
 0001DC  58F0  319A        000017 |                 L        r15,=V(PRINTF)(,r3,410)
 0001E0  4100  5053        000017 |                 LA       r0,+CONSTANT_AREA(,r5,83)
 0001E4  4110  D098        000017 |                 LA       r1,#MX_TEMP1(,r13,152)
 0001E8  5000  D098        000017 |                 ST       r0,#MX_TEMP1(,r13,152)
 0001EC  0DEF              000017 |                 BASR     r14,r15
 0001EE  47F0  30AE        000017 |                 B        @1L3
 0001F2                    000017 |         @1L2     DS       0H
                           000018 |        *        }
                           000019 |        *        else {
                           000020 |        *           convert(c_temp);
 0001F2  6800  D0B0        000020 |                 LD       f0,c_temp(,r13,176)
                           000021 |        *        }
                           000022 |        *     }
                           000023 |        *     else {  /* convert the command-line arguments to Fahrenheit */
                           000024 |        *        int i;
                           000025 |        *
                           000026 |        *        for (i = 1; i < argc; ++i) {
                           000027 |        *            if (sscanf(argv[i], "%f", &c_temp) != 1)
                           000028 |        *               printf("%s is not a valid temperature.",argv[i]);
                           000029 |        *            else
                           000030 |        *               convert(c_temp);
                           000031 |        *        }
                           000032 |        *     }
                           000033 |        *     return 0;
                           000034 |        *  }
                           000035 |        *
                           000036 |        *  void convert(double c_temp) {
 0001F6  6000  D0C0        000036 |                 STD      f0,c_temp%2=>1(,r13,192)
 0001FA  6820  5000        000037 |        +        LD       f2,+CONSTANT_AREA(,r5,0)
 0001FE  2C02              000037 |        +        MDR      f0,f2
 000200  6820  5008        000037 |        +        LD       f2,+CONSTANT_AREA(,r5,8)
 000204  2A02              000037 |        +        ADR      f0,f2
 000206  6000  D0C8        000037 |        +        STD      f0,f_temp%2=>1(,r13,200)
 00020A  6820  D0C0        000038 |        +        LD       f2,c_temp%2=>1(,r13,192)
 00020E  6020  D09C        000038 |        +        STD      f2,#MX_TEMP1(,r13,156)
 000212  6000  D0A4        000038 |        +        STD      f0,#MX_TEMP1(,r13,164)
 000216  58F0  319A        000038 |        +        L        r15,=V(PRINTF)(,r3,410)
 00021A  4100  5010        000038 |        +        LA       r0,+CONSTANT_AREA(,r5,16)
 00021E  4110  D098        000038 |        +        LA       r1,#MX_TEMP1(,r13,152)
 000222  5000  D098        000038 |        +        ST       r0,#MX_TEMP1(,r13,152)
 000226  0DEF              000038 |        +        BASR     r14,r15
 000228                    000039 |        +@1L12    DS       0H
 000228                    000020 |         @1L3     DS       0H
 000228  47F0  3182        000020 |                 B        @1L4
 00022C                    000020 |         @1L1     DS       0H
 00022C  4100  0001        000026 |                 LA       r0,1
 000230  5000  D0B8        000026 |                 ST       r0,i(,r13,184)
 000234  5810  D0BC        000026 |                 L        r1,#SR_PARM_1(,r13,188)
 000238  5810  1000        000026 |                 L        r1,argc(,r1,0)
 00023C  1901              000026 |                 CR       r0,r1
```

Figure 11. Example of a C listing (Part 30 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)': main            04/29/2009 04:55:12   Page    35

 OFFSET OBJECT CODE       LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G

 00023E  47B0  3182       000026 |                 BNL      @1L6
 000242                   000026 |        @1L5     DS       0H
 000242  5810  D0BC       000027 |                 L        r1,#SR_PARM_1(,r13,188)
 000246  5810  1004       000027 |                 L        r1,argv(,r1,4)
 00024A  5820  D0B8       000027 |                 L        r2,i(,r13,184)
 00024E  8920  0002       000027 |                 SLL      r2,2
 000252  5822  1000       000027 |                 L        r2,(*)uchar*(r2,r1,0)
 000256  4100  D0B0       000027 |                 LA       r0,c_temp(,r13,176)
 00025A  58F0  31A2       000027 |                 L        r15,=V(SSCANF)(,r3,418)
 00025E  4110  D098       000027 |                 LA       r1,#MX_TEMP1(,r13,152)
 000262  5020  D098       000027 |                 ST       r2,#MX_TEMP1(,r13,152)
 000266  4120  5050       000027 |                 LA       r2,+CONSTANT_AREA(,r5,80)
 00026A  5020  D09C       000027 |                 ST       r2,#MX_TEMP1(,r13,156)
 00026E  5000  D0A0       000027 |                 ST       r0,#MX_TEMP1(,r13,160)
 000272  0DEF             000027 |                 BASR     r14,r15
 000274  180F             000027 |                 LR       r0,r15
 000276  A70E  0001       000027 |                 CHI      r0,H'1'
 00027A  4780  3132       000027 |                 BE       @1L7
 00027E  5810  D0BC       000028 |                 L        r1,#SR_PARM_1(,r13,188)
 000282  5810  1004       000028 |                 L        r1,argv(,r1,4)
 000286  5820  D0B8       000028 |                 L        r2,i(,r13,184)
 00028A  8920  0002       000028 |                 SLL      r2,2
 00028E  5802  1000       000028 |                 L        r0,(*)uchar*(r2,r1,0)
 000292  58F0  319A       000028 |                 L        r15,=V(PRINTF)(,r3,410)
 000296  4120  5077       000028 |                 LA       r2,+CONSTANT_AREA(,r5,119)
 00029A  4110  D098       000028 |                 LA       r1,#MX_TEMP1(,r13,152)
 00029E  5020  D098       000028 |                 ST       r2,#MX_TEMP1(,r13,152)
 0002A2  5000  D09C       000028 |                 ST       r0,#MX_TEMP1(,r13,156)
 0002A6  0DEF             000028 |                 BASR     r14,r15
 0002A8  47F0  3168       000028 |                 B        @1L8
 0002AC                   000028 |        @1L7     DS       0H
 0002AC  6800  D0B0       000030 |                 LD       f0,c_temp(,r13,176)
 0002B0  6000  D0C0       000036 |                 STD      f0,c_temp%2=>1(,r13,192)
 0002B4  6820  5000       000037 |       +         LD       f2,+CONSTANT_AREA(,r5,0)
 0002B8  2C02             000037 |       +         MDR      f0,f2
 0002BA  6820  5008       000037 |       +         LD       f2,+CONSTANT_AREA(,r5,8)
 0002BE  2A02             000037 |       +         ADR      f0,f2
 0002C0  6000  D0C8       000037 |       +         STD      f0,f_temp%2=>1(,r13,200)
 0002C4  6820  D0C0       000038 |       +         LD       f2,c_temp%2=>1(,r13,192)
 0002C8  6020  D09C       000038 |       +         STD      f2,#MX_TEMP1(,r13,156)
 0002CC  6000  D0A4       000038 |       +         STD      f0,#MX_TEMP1(,r13,164)
 0002D0  58F0  319A       000038 |       +         L        r15,=V(PRINTF)(,r3,410)
 0002D4  4100  5010       000038 |       +         LA       r0,+CONSTANT_AREA(,r5,16)
 0002D8  4110  D098       000038 |       +         LA       r1,#MX_TEMP1(,r13,152)
 0002DC  5000  D098       000038 |       +         ST       r0,#MX_TEMP1(,r13,152)
 0002E0  0DEF             000038 |       +         BASR     r14,r15
 0002E2                   000039 |       +@1L13    DS       0H
 0002E2                   000030 |        @1L8     DS       0H
 0002E2  5800  D0B8       000026 |                 L        r0,i(,r13,184)
 0002E6  A70A  0001       000026 |                 AHI      r0,H'1'
 0002EA  5000  D0B8       000026 |                 ST       r0,i(,r13,184)
 0002EE  5810  D0BC       000026 |                 L        r1,#SR_PARM_1(,r13,188)
 0002F2  5810  1000       000026 |                 L        r1,argc(,r1,0)
 0002F6  1901             000026 |                 CR       r0,r1
 0002F8  4740  30C8       000026 |                 BL       @1L5
 0002FC                   000026 |        @1L6     DS       0H
```

Figure 11. Example of a C listing (Part 31 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)': main            04/29/2009 04:55:12   Page    36

 OFFSET OBJECT CODE       LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G

 0002FC                   000026 |        @1L4     DS       0H
 0002FC  41F0  0000       000033 |                 LA       r15,0
 000300                   000034 |        @1L10    DS       0H

 000300                   Start of Epilog
 000300  180D             000034 |                 LR       r0,r13
 000302  58D0  D004       000034 |                 L        r13,4(,r13)
 000306  58E0  D00C       000034 |                 L        r14,12(,r13)
 00030A  9825  D01C       000034 |                 LM       r2,r5,28(r13)
 00030E  051E             000034 |                 BALR     r1,r14
 000310  0707             000034 |                 NOPR     7
 000312  0000

 000314                   Start of Literals
 000314  00000000                                           =V(PRINTF)
 000318  00000000                                           =V(SCANF)
 00031C  00000000                                           =V(SSCANF)
 000320                   End of Literals

                          ***   General purpose registers used: 1111110000001111
                          ***   Floating point  registers used: 1111111100000000
                          ***   Size of register spill area: 128(max) 0(used)
                          ***   Size of dynamic storage: 272
                          ***   Size of executable code: 466

                          Constant Area
 000320  411CCCCC CCCCCCCC 42200000 00000000    |................|
 000330  6CF54BF2 8640C385 93A289A4 A24089A2    |%5.2f Celsius is|
 000340  406CF54B F28640C6 81889985 95888589    | %5.2f Fahrenhei|
 000350  A31500C5 95A38599 40C38593 A289A4A2    |t..Enter Celsius|
 000360  40A38594 97859981 A3A49985 7A401500    | temperature: ..|
 000370  6C8600E8 96A44094 A4A2A340 8595A385    |%f.You must ente|
 000380  99408140 A5819389 8440A385 94978599    |r a valid temper|
 000390  81A3A499 8515006C A24089A2 409596A3    |ature..%s is not|
 0003A0  408140A5 81938984 40A38594 97859981    | a valid tempera|
 0003B0  A3A49985 1500                          |ture..          |
```

Figure 11. Example of a C listing (Part 32 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                04/29/2009 04:55:12   Page    37

 OFFSET OBJECT CODE        LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G

                           PPA1: Entry Point Constants
 0003B8  1CCEA106                                           =F'483303686'      Flags
 0003BC  000003A8                                           =A(PPA2-convert)
 0003C0  00000000                                           =F'0'              No PPA3
 0003C4  00000000                                           =F'0'              No EPD
 0003C8  FF000000                                           =F'-16777216'      Register save mask
 0003CC  00000000                                           =F'0'              Member flags
 0003D0  90                                                 =AL1(144)          Flags
 0003D1  000000                                             =AL3(0)            Callee's DSA use/8
 0003D4  0040                                               =H'64'             Flags
 0003D6  0012                                               =H'18'             Offset/2 to CDL
 0003D8  00000000                                           =F'0'              Reserved
 0003DC  50000050                                           =F'1342177360'     CDL function length/2
 0003E0  FFFFFCE0                                           =F'-800'           CDL function EP offset
 0003E4  38260000                                           =F'942014464'      CDL prolog
 0003E8  40080048                                           =F'1074266184'     CDL epilog
 0003EC  00000000                                           =F'0'              CDL end
 0003F0  0007  ****                                         AL2(7),C'convert'
                           PPA1 End

                           PPA1: Entry Point Constants
 000400  1CCEA106                                           =F'483303686'      Flags
 000404  00000300                                           =A(PPA2-main)
 000408  00000000                                           =F'0'              No PPA3
 00040C  00000000                                           =F'0'              No EPD
 000410  FF000000                                           =F'-16777216'      Register save mask
 000414  00000000                                           =F'0'              Member flags
 000418  90                                                 =AL1(144)          Flags
 000419  000000                                             =AL3(0)            Callee's DSA use/8
 00041C  0040                                               =H'64'             Flags
 00041E  0012                                               =H'18'             Offset/2 to CDL
 000420  00000000                                           =F'0'              Reserved
 000424  500000E9                                           =F'1342177513'     CDL function length/2
 000428  FFFFFD40                                           =F'-704'           CDL function EP offset
 00042C  38260000                                           =F'942014464'      CDL prolog
 000430  400900E0                                           =F'1074331872'     CDL epilog
 000434  00000000                                           =F'0'              CDL end
 000438  0004  ****                                         AL2(4),C'main'
                           PPA1 End

                           PPA2: Compile Unit Block
 000440  0300  2203                                         =F'50340355'       Flags
 000444  FFFF  FBC0                                         =A(CEESTART-PPA2)
 000448  0000  0000                                         =F'0'              No PPA4
 00044C  FFFF  FBC0                                         =A(TIMESTMP-PPA2)
 000450  0000  0000                                         =F'0'              No primary
 000454  0200  0000                                         =F'33554432'       Flags
                           PPA2 End
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                04/29/2009 04:55:12   Page    38

                                   E X T E R N A L   S Y M B O L   D I C T I O N A R Y

           NAME        TYPE  ID  ADDR    LENGTH            NAME        TYPE  ID  ADDR    LENGTH

                        PC    1 000000   000458            CONVERT      LD    0 000098   000001
           MAIN         LD    0 000140   000001            CEESG003     ER    2 000000
           PRINTF       ER    3 000000                     SCANF        ER    4 000000
           SSCANF       ER    5 000000                     CEESTART     ER    6 000000
           CEEMAIN      SD    7 000000   00000C            EDCINPL      ER    8 000000
           MAIN         ER    9 000000
```

Figure 11. Example of a C listing (Part 33 of 34)

```
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                  04/29/2009 04:55:12   Page    39

                            E X T E R N A L   S Y M B O L   C R O S S   R E F E R E N C E

           ORIGINAL NAME                                   EXTERNAL SYMBOL NAME

           convert                                         CONVERT
           main                                            MAIN
           CEESG003                                        CEESG003
           printf                                          PRINTF
           scanf                                           SCANF
           sscanf                                          SSCANF
           CEESTART                                        CEESTART
           CEEMAIN                                         CEEMAIN
           EDCINPL                                         EDCINPL
15694A01 V1.11 z/OS XL C                'CBC.SCCNSAM(CCNUAAM)'                  04/29/2009 04:55:12   Page    40

                               * * * * *   S T O R A G E   O F F S E T   L I S T I N G   * * * * *

 IDENTIFIER          DEFINITION      ATTRIBUTES

 argc                7-0:7           Class = parameter,          Location = 0(r1),                              Length = 4

 argv                7-0:7           Class = parameter,          Location = 4(r1),                              Length = 4

 c_temp              9-0:9           Class = automatic,          Location = 176(r13),                           Length = 8

 i                   24-0:24         Class = automatic,          Location = 184(r13),                           Length = 4

 c_temp              36-0:36         Class = parameter,          Location = 0(r1),                              Length = 8

 f_temp              37-0:37         Class = automatic,          Location = 176(r13),                           Length = 8

                         * * * * *   E N D   O F   S T O R A G E   O F F S E T   L I S T I N G   * * * * *

                             * * * * *   E N D   O F   C O M P I L A T I O N   * * * * *
```

Figure 11. Example of a C listing (Part 34 of 34)

z/OS XL C compiler listing components

The following information describes the components of a C compiler listing. These are available for regular and IPA compilations. Differences in the IPA versions of the listings are noted. "Using the IPA Link step listing" on page 317 describes IPA-specific listings.

Heading information

The first page of the listing is identified by the product number, the compiler version and release numbers, the name of the data set or z/OS UNIX System Services file containing the source code, the date and time compilation began (formatted according to the current locale), and the page number.

Note: If the name of the data set or z/OS UNIX file that contains the source code is greater than 32 characters, it is truncated. Only the right-most 32 characters appear in the listing.

Prolog section

The Prolog section provides information about the compile-time library, file identifiers, compiler options, and other items in effect when the compiler was invoked.

All options except those with no default (for example, DEFINE) are shown in the listing. Any problems with the compiler options appear after the body of the Prolog section.

IPA considerations: If you specify IPA suboptions that are irrelevant to the IPA Compile step, the Prolog does not display them. If IPA processing is not active, IPA suboptions do not appear in the Prolog.

The following information describes the optional parts of the listing and the compiler options that generate them.

Source program

If you specify the SOURCE option, the listing file includes input to the compiler.

Note: If you specify the SHOWINC option, the source listing shows the included text after the `#include` directives.

Includes section

The compiler generates the Includes section when you use include files, and specify the options SOURCE, LIST, or INLRPT.

Cross-Reference Listing

The XREF option generates a cross-reference table that contains a list of the identifiers from the source program and the line numbers in which they appear.

Structure and Union Maps

You obtain structure and union maps by using the AGGREGATE option. The table shows how each structure and union in the program is mapped. It contains the following:

- Name of the structure or union and the elements within the structure or union
- Byte offset of each element from the beginning of the structure or union, and the bit offset for unaligned bit data
- Length of each element
- Total length of each structure, union, and substructure

Messages

If the preprocessor or the compiler detects an error, or the possibility of an error, it generates messages. If you specify the SOURCE compiler option, preprocessor error messages appear immediately after the source statement in error. You can generate your own messages in the preprocessing stage by using the **#error** preprocessor directive. For information on **#error**, see the *z/OS XL C/C++ Language Reference*.

If you specify the compiler options CHECKOUT or INFO(), the compiler will generate informational diagnostic messages.

For more information on the compiler messages, see “FLAG | NOFLAG” on page 113, and *z/OS XL C/C++ Messages*.

Message Summary

This listing section displays the total number of messages and the number of messages for each severity level.

Inline Report

If you specify the OPTIMIZE and INLINE(,REPORT,,) options, or the OPTIMIZE and INLRPT options, an Inline Report is included in the listing. This report contains an inline summary and a detailed call structure.

Note: No report is produced when your source file contains only one defined subprogram.

The summary contains information such as:

| • Name of each defined subprogram.
- Reason for action on a subprogram:
  – The P indicates that **#pragma noinline** and the COMPACT compiler option are not in effect.
  – The F indicates that the subprogram was declared inline, either by **#pragma inline** for C or the `inline` keyword for C++.
  – The C indicates that the COMPACT compiler option is specified for **#pragma_override(FuncName,″OPT(COMPACT,yes)**″ is specified in the source code.
  – The M indicates that C++ routine is an inline member routine.
  – The A indicates automatic inlining acted on the subprogram.
  – The - indicates there was no reason to inline the subprogram.
- Action on a subprogram:
  – Subprogram was inlined at least once.
  – Subprogram was not inlined because of initial size constraints.
  – Subprogram was not inlined because of expansion beyond size constraint.
  – Subprogram was a candidate for inlining, but was not inlined.
  – Subprogram was a candidate for inlining, but was not referenced.
  – The subprogram is directly recursive, or some calls have mismatching parameters.

  Note: ″Called″ and ″Calls″ in the actions section of the inline report indicate how many times a function has been called or has called other functions, regardless of whether or not the callers or callees have been inlined.
- Status of original subprogram after inlining:
  – Subprogram is discarded because it is no longer referenced and is defined as `static` internal.
  – Subprogram was not discarded for various reasons :
    - Subprogram is external. (It can be called from outside the compilation unit.)
    - A call to this subprogram remains.
    - Subprogram has its address taken.
- Initial relative size of subprogram (in Abstract Code Units (ACU)).
- Final relative size of subprogram (in ACUs) after inlining.
- Number of calls within the subprogram and the number of these calls that were inlined into subprogram.
- Number of times the subprogram is called by others in the compile unit and the number of times the subprogram was inlined.
- Mode that is selected and the value of *threshold* and *limit* specified for the compilation.

The detailed call structure contains specific information of each subprogram such as:
- Subprograms that it calls
- Subprograms that call it
- Subprograms in which it is inlined

The information can help you to better analyze your program if you want to use the inliner in selective mode.

Inlining may result in additional messages. For example, if inlining a subprogram with automatic storage increases the automatic storage of the subprogram it is being inlined into by more than 4K, a message is generated.

Pseudo Assembly Listing

The LIST compiler option generates a listing of the machine instructions in the object module in a form similar to assembler language.

This Pseudo Assembly listing displays the source statement line numbers and the line number of inlined code to aid you in debugging inlined code.

External Symbol Dictionary

The LIST compiler option generates the External Symbol Dictionary. The External Symbol Dictionary lists the names that the compiler generates for the output object module. It includes address information and size information about each symbol.

External Symbol Cross Reference

The XREF compiler option generates the External Symbol Cross Reference section. It shows the original name and corresponding mangled name for each symbol.

Storage Offset Listing

If you specify the XREF option, the listing file includes offset information on identifiers.

Static Map

Static Map displays the contents of the @STATIC data area, which holds the file scope read/write static variables. It displays the offset (as a hexadecimal number), the length (as a hexadecimal number), and the names of the objects mapped to @STATIC. Under certain circumstances, the compiler may decide to map other objects to @STATIC. In the example of the listing, the unnamed string ″Enter Celsius temperature: \n″ is stored in the @STATIC area at offset 48 and its length is 23 (both numbers are in hexadecimal notation), under the name ″″12.

If you specify the XREF, IPA (ATTRIBUTE), or IPA (XREF) options, the listing file includes offset information for file scope read/write static variables.

Using the z/OS XL C++ compiler listing

If you select the SOURCE, INLRPT, or LIST option, the compiler creates a listing that contains information about the source program and the compilation. If the compilation terminates before reaching a particular stage of processing, the compiler does not generate corresponding parts of the listing. The listing contains standard information that always appears, together with optional information that is supplied by default or specified through compiler options.

In an interactive environment you can also use the TERMINAL option to direct all compiler diagnostic messages to your terminal. The TERMINAL option directs only the diagnostic messages part of the compiler listing to your terminal.

Notes:

1. Although the compiler listing is for your use, it is not a programming interface and is subject to change.
2. The compiler always attempts to put diagnostic messages in the listing, as close as possible to the location where the condition occurred. The exact location or line number within the listing may not be the same from release to release.

IPA considerations

The listings that the IPA Compile step produces are basically the same as those that a regular compilation produces. Any differences are noted throughout this section.

The IPA Link step listing has a separate format from the other compiler listings. Many listing sections are similar to those that are produced by a regular compilation or the IPA Compile step with the IPA(OBJECT) option specified. Refer to “Using the IPA Link step listing” on page 317 for information about IPA Link step listings.

Example of a C++ compiler listing

Figure 12 shows an example of a z/OS XL C++ compiler listing. Vertical ellipses indicate sections that have been truncated.

```
15694A01 V1.11 z/OS XL C++        //'CBC.SCCNSAM(CCNUBRC)'                    04/29/2009 04:55:13

                                      * * * * *   P R O L O G   * * * * *
   Compiler options. . . . . . . :AGGRCOPY(NOOVERLAP)             ANSIALIAS       ARCH(5)         ARGPARSE        NOASCII
                                 :NOATTRIBUTE     ASSERT(RESTRICT)                BITFIELD(UNSIGNED)              CHARS(UNSIGNED)
                                 :NOCOMPACT       NOCOMPRESS      CVFT            NODFP           DIGRAPH
                                 :DLL(NOCALLBACKANY)              ENUMSIZE(SMALL) NOEVENTS        EXECOPS         EXH
                                 :NOEXPMAC        NOEXPORTALL     NOFASTTEMPINC   FLAG(I)         NOGOFF          NOGONUMBER
                                 :HALT(16)        NOHGPR(NOPRESERVE)              NOHOT           NOIGNERRNO      ILP32
                                 :NOINITAUTO      INLRPT          NOLIBANSI       LIST            LONGNAME        LONGLONG
                                 :NOMAKEDEP       NOMARGINS       MAXMEM(2097152) MEMORY          NAMEMANGLING(zOSV1R2)
                                 :NESTINC(255)    OBJECT          OBJECTMODEL(COMPAT)             NOOE            NOOFFSET
                                 :NOOPTIMIZE      PLIST(HOST)     NOPORT          NOPPONLY        PREFETCH        REDIR
                                 :NOREPORT        ROSTRING        ROCONST         NORTTI          NOSEQUENCE      NOSHOWINC
                                 :NOSHOWMACROS    SOURCE          SKIPSRC(SHOW)   SPILL(128)      START           NOSTATICINLINE
                                 :STRICT          NOSTRICT_INDUCTION              TARGET(LE,CURRENT)
                                 :NOTEMPLATEREGISTRY              TEMPLATERECOMPILE               TERMINAL        NOTEST(HOOK)
                                 :TMPLPARSE(NO)   TUNE(5)         UNROLL(AUTO)    UTF             NOWARN0X        NOWARN64
                                 :NOWSIZEOF       XREF
                                 :NOCICS
                                 :NOCONVLIT
                                 :NOCSECT
                                 :NODEBUG
                                 :FLOAT(HEX,FOLD,AFP(NOVOLATILE)) ROUND(Z)
                                 :NOHALTONMSG
                                 :INFO(LAN)
                                 :INLINE(AUTO,REPORT,100,1000)
                                 :NOIPA
                                 :LANGLVL(ANONSTRUCT,ANONUNION,ANSIFOR,ANSISINIT,C99VLA,C99__FUNC__,NODBCS,DEPENDENTBASELOOKUP,
                                  NODOLLARINNAMES,EMPTYSTRUCT,NOEXTENDEDFRIEND,EXTERNTEMPLATE,ILLPTOM,IMPLICITINT,LIBEXT,LONGLONG,
                                  NONEWEXCP,OFFSETNONPOD,NOOLDDIGRAPH,OLDFRIEND,NOOLDMATH,NOOLDSTR,OLDTEMPACC,NOOLDTMPLALIGN,
                                  OLDTMPLSPEC,TRAILENUM,TYPEDEFCLASS,NOUCS,GNU_INCLUDE_NEXT,ZEROEXTARRAY)
                                 :NOLOCALE
                                 :NOLSEARCH
                                 :OPTFILE(DD:OPTS)
                                 :NORTCHECK
                                 :SEARCH(//'CEE.SCEEH.+', //'CBC.SCLBH.+')
                                 :NOSERVICE
                                 :NOSQL
                                 :NOSUPPRESS
                                 :TEMPINC(./tempinc)
                                 :NOXPLINK(NOBACKCHAIN,NOCALLBACK,GUARD,OSCALL(UPSTACK),NOSTOREARGS)
   Version Macros. . . . . . . . : __COMPILER_VER__=0x410B0000
                                 : __LIBREL__=0x410B0000
                                 : __TARGET_LIB__=0x410B0000
   Source margins. . . . . . . . :
     Varying length. . . . . . . : 1 - 32760
     Fixed length. . . . . . . . : 1 - 32760
   Sequence columns. . . . . . . :
     Varying length. . . . . . . : none
     Fixed length. . . . . . . . : none
   Listing name. . . . . . . . . : DD:SYSCPRT

                                 * * * * *   E N D   O F   P R O L O G   * * * * *
```

Figure 12. Example of a C++ compiler listing (Part 1 of 15)

```
15694A01 V1.11 z/OS XL C++        //'CBC.SCCNSAM(CCNUBRC)'                  04/30/2009 02:04:27



                                  * * * * *   S O U R C E   * * * * *

           1 | //
           2 | // Sample Program: Biorhythm
           3 | // Description   : Calculates biorhythm based on the current
           4 | //                 system date and birth date entered
           5 | //
           6 | // File 2 of 2-other file is CCNUBRH
           7 |
           8 | #include <stdio.h>
           9 | #include <string.h>
          10 | #include <math.h>
          11 | #include <time.h>
          12 | #include <iostream>
          13 | #include <iomanip>
          14 |
          15 | #include "ccnubrh.h"  //BioRhythm class and Date class
          16 | using namespace std;
          17 | static ostream& operator << (ostream&, BioRhythm&);
          18 |
          19 |
          20 | int main(void) {
          21 |
          22 |   BioRhythm bio;
          23 |   int code;
          24 |
          25 |   if (!bio.ok()) {
          26 |     cerr << "Error in birthdate specification - format is yyyy/mm/dd";
          27 |     code = 8;
          28 |   }
          29 |   else {
          30 |     cout << bio;  // write out birthdate for bio
          31 |     code = 0;
          32 |   }
          33 |   return(code);
          34 | }
          35 |
          36 | const int Date::dateLen ;
          37 | const int Date::numMonths;
          38 | const int Date::numDays[Date::numMonths] = {
          39 |   31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
          40 |     };
          41 |
          42 | const int BioRhythm::pCycle;
          43 | const int BioRhythm::eCycle;
          44 | const int BioRhythm::iCycle;
          45 |
          46 | ostream& operator<<(ostream& os, BioRhythm& bio) {
          47 |   os << "Total Days  : " << bio.AgeInDays() << "\n";
          48 |   os << "Physical    : " << bio.Physical() << "\n";
          49 |   os << "Emotional   : " << bio.Emotional() << "\n";
```

Figure 12. Example of a C++ compiler listing (Part 2 of 15)

```
     50 |   os << "Intellectual: " << bio.Intellectual() << "\n";
     51 |
     52 | return(os);
     53 | }
     54 |
     55 | Date::Date() {
     56 |   time_t lTime;
     57 |   struct tm *newTime;
     58 |
     59 |   time(&lTime);
     60 |   newTime = localtime(&lTime);
     61 |   cout << "local time is " << asctime(newTime) << endl;
     62 |
     63 |   curYear = newTime->tm_year + 1900;
     64 |   curDay  = newTime->tm_yday + 1;
     65 | }
     66 |
     67 | BirthDate::BirthDate(const char *birthText) {
     68 |   strcpy(text, birthText);
     69 | }
     70 |
     71 | BirthDate::BirthDate() {
     72 |   cout << "Please enter your birthdate in the form yyyy/mm/dd\n";
     73 |   cin >> setw(dateLen+1) >> text;
     74 | }
     75 |
     76 | Date::DaysSince(const char *text) {
     77 |
     78 |   int year, month, day, totDays, delim;
     79 |   int daysInYear = 0;
     80 |   int i;
     81 |   int leap = 0;
     82 |
     83 |   int rc = sscanf(text, "%4d%c%2d%c%2d",
     84 |                   &year, &delim, &month, &delim, &day);
     85 |   --month;
     86 |   if (rc != 5 || year  < 0 || year  > 9999 ||
     87 |                 month < 0 || month >   11 ||
     88 |                 day   < 1 || day   >   31 ||
     89 |                (day   > numDays[month]&& month != 1)) {
     90 |     return(-1);
     91 |   }
     92 |   if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
     93 |      leap = 1;
     94 |
     95 |   if (month == 1 && day > numDays[month]) {
     96 |     if (day > 29)
     97 |       return(-1);
     98 |     else if (!leap)
     99 |       return (-1);
    100 |   }
    101 |
    102 |   for (i=0;i<month;++i) {
    103 |     daysInYear += numDays[i];
    104 |   }
    105 |   daysInYear += day;
    106 |
    107 |   // correct for leap year
    108 |   if (leap == 1 &&
    109 |      (month > 1 || (month == 1 && day == 29)))
    110 |      ++daysInYear;
    111 |
    112 |   totDays = (curDay - daysInYear) + (curYear - year)*365;
    113 |
    114 |   // now, correct for leap year
    115 |   for (i=year+1; i < curYear; ++i) {
    116 |     if ((i % 4 == 0 && i % 100 != 0) || i % 400 == 0) {
    117 |       ++totDays;
    118 |     }
    119 |   }
    120 |   return(totDays);
    121 | }

                          * * * * *   E N D   O F   S O U R C E   * * * * *
```

Figure 12. Example of a C++ compiler listing (Part 3 of 15)

```
15694A01 V1.11 z/OS XL C++       //'CBC.SCCNSAM(CCNUBRC)'                  04/29/2009 04:55:13

                                   * * * * *   I N C L U D E S   * * * * *
     1  =  //'CEE.SCEEH.H(STDIO)'
     2  =  //'CEE.SCEEH.H(FEATURES)'
     3  =  //'CEE.SCEEH.SYS.H(TYPES)'
     4  =  //'CEE.SCEEH.H(STRING)'
     5  =  //'CEE.SCEEH.H(MATH)'
     6  =  //'CEE.SCEEH.H(BUILTINS)'
     7  =  //'CEE.SCEEH.H(STDDEF)'
     8  =  //'CEE.SCEEH.H(TIME)'
     9  =  //'CEE.SCEEH(IOSTREAM)'
    10  =  //'CEE.SCEEH(ISTREAM)'
    11  =  //'CEE.SCEEH(OSTREAM)'
    12  =  //'CEE.SCEEH.H(YVALS)'
    13  =  //'CEE.SCEEH(IOS)'
    14  =  //'CEE.SCEEH(XLOCNUM)'
    15  =  //'CEE.SCEEH(CERRNO)'
    16  =  //'CEE.SCEEH.H(ERRNO)'
    17  =  //'CEE.SCEEH(CLIMITS)'
    18  =  //'CEE.SCEEH.H(LIMITS)'
    19  =  //'CEE.SCEEH(CSTDIO)'
    20  =  //'CEE.SCEEH(CSTDLIB)'
    21  =  //'CEE.SCEEH.H(STDLIB)'
    22  =  //'CEE.SCEEH(STREAMBU)'

 ⋮
                                * * * * *   E N D   O F   I N C L U D E S   * * * * *
```

Figure 12. Example of a C++ compiler listing (Part 4 of 15)

```
15694A01 V1.11 z/OS XL C++        //'CBC.SCCNSAM(CCNUBRC)'                    04/29/2009 04:55:13

                       * * * * *  C R O S S   R E F E R E N C E   L I S T I N G   * * * * *

   ___valist                          :
             1:142 (D)    1:145 (R)


   __abs                              :
            21:400 (R)   21:481 (R)


   __absd                             :
             5:1162(R)    5:1242(R)


   __acos                             :
             5:1148(R)    5:1245(R)


   __acosf                            :
             5:1163(R)    5:1244(R)


   __acosl                            :
             5:1164(R)    5:1246(R)


   __amrc_type                        :
             1:943 (D)    1:947 (R)


   __amrctype                         :
             1:877 (D)    1:943 (R)


   __amrc2_type                       :
             1:957 (D)    1:961 (R)


   __amrc2type                        :
             1:952 (D)    1:957 (R)
⋮
             * * * * *  E N D   O F   C R O S S   R E F E R E N C E   L I S T I N G   * * * * *
```

Figure 12. Example of a C++ compiler listing (Part 5 of 15)

```
15694A01 V1.11 z/OS XL C++        //'CBC.SCCNSAM(CCNUBRC)'                    04/29/2009 04:55:13


                           * * * * *  M E S S A G E   S U M M A R Y   * * * * *

TOTAL   UNRECOVERABLE  SEVERE       ERROR     WARNING    INFORMATIONAL
                (U)       (S)         (E)        (W)          (I)
    0           0         0           0          0            0

                    * * * * *  E N D   O F   M E S S A G E   S U M M A R Y   * * * * *
```

Figure 12. Example of a C++ compiler listing (Part 6 of 15)

```
15694A01 V1.11 z/OS XL C++                                          CCNUBRC                 04/29/09 04:55:13

                      Inline Report (Summary)

   Reason:    P : noinline was specified for this routine
              F : inline was specified for this routine
              C : compact was specified for this routine
              M : This is an inline member routine
              A : Automatic inlining
              - : No reason
   Action:    I : Routine is inlined at least once
              L : Routine is initially too large to be inlined
              T : Routine expands too large to be inlined
              C : Candidate for inlining but not inlined
              N : No direct calls to routine are found in file (no action)
              U : Some calls not inlined due to recursion or parameter mismatch
              - : No action
   Status:    D : Internal routine is discarded
              R : A direct call remains to internal routine (cannot discard)
              A : Routine has its address taken (cannot discard)
              E : External routine (cannot discard)
              - : Status unchanged
   Calls/I      : Number of calls to defined routines / Number inline
   Called/I     : Number of times called / Number of times inlined

   Reason  Action  Status   Size (init)    Calls/I   Called/I     Name

      A      N       E      224   (60)       4/2       0/0        main
      A      I       E      88               2/0       1/1        BirthDate::BirthDate()
      A      L       R      167              0/0       1/0        Date::DaysSince(const char*)
      P      -       -      670   (380)      4/5       5/0
⋮
  Mode = AUTO    Inlining Threshold = 100    Expansion Limit = 1000
```

Figure 12. Example of a C++ compiler listing (Part 7 of 15)

```
15694A01 V1.11 z/OS XL C++                                          CCNUBRC                 04/29/09 04:55:13

                      Inline Report (Call Structure)

   Defined Function     : main
       Calls To(4,2)    : BirthDate::BirthDate() (1,1)
                        : Date::DaysSince(const char*) (1,0)
                        : std::_EBCDIC::_LFS_OFF::basic_ostream<char,std::char_traits<char> >& std::_EBCDIC::_LFS_OFF::operator<
                          <<std::char_traits<char> >(std::_EBCDIC::_LFS_OFF::basic_ostream<char,std::char_traits<char> >&,const
                          char*) (1,0)
                        : operator<<(std::_EBCDIC::_LFS_OFF::basic_ostream<char,std::char_traits<char> >&,BioRhythm&) (1,1)
    Called From         : 0

   Defined Function     : BirthDate::BirthDate()
       Calls To(2,0)    : std::_EBCDIC::_LFS_OFF::basic_ostream<char,std::char_traits<char> >& std::_EBCDIC::_LFS_OFF::operator<
                          <<std::char_traits<char> >(std::_EBCDIC::_LFS_OFF::basic_ostream<char,std::char_traits<char> >&,const
                          char*) (1,0)
                        : std::_EBCDIC::_LFS_OFF::basic_istream<char,std::char_traits<char> >& std::_EBCDIC::_LFS_OFF::operator>
                          ><char,std::char_traits<char> >(std::_EBCDIC::_LFS_OFF::basic_istream<char,std::char_traits<char> >&,c
                          har*) (1,0)
    Called From(1,1)    : main (1,1)
⋮
```

Figure 12. Example of a C++ compiler listing (Part 8 of 15)

```
15694A01 V1.11 z/OS XL C++                                          CCNUBRC                   04/29/09 04:55:13

                      Inline Report (Additional Information)

 INFORMATIONAL CCN1051: Function Date::DaysSince(const char*) exceeds size limit.

 INFORMATIONAL CCN1051: Function std::_EBCDIC::_LFS_OFF::basic_ exceeds size limit.

 INFORMATIONAL CCN1051: Function std::_EBCDIC::_LFS_OFF::basic_ exceeds size limit.

 INFORMATIONAL CCN1051: Function std::ostreambuf_iterator<char, exceeds size limit.

 INFORMATIONAL CCN1051: Function std::ostreambuf_iterator<char, exceeds size limit.

 INFORMATIONAL CCN1051: Function std::_EBCDIC::_LFS_OFF::num_pu exceeds size limit.

 INFORMATIONAL CCN1051: Function std::_EBCDIC::_LFS_OFF::num_pu exceeds size limit.

 INFORMATIONAL CCN1051: Function std::_EBCDIC::_LFS_OFF::num_pu exceeds size limit.
⋮
```

Figure 12. Example of a C++ compiler listing (Part 9 of 15)

```
15694A01 V1.11 z/OS XL C++                                          CCNUBRC                   04/29/09 04:55:13

 OFFSET OBJECT CODE        LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G

                           Timestamp and Version Information
 000000  F2F0  F0F9                                          =C'2009'           Compiled Year
 000004  F0F4  F2F9                                          =C'0429'           Compiled Date MMDD
 000008  F0F4  F5F5  F1F4                                    =C'045514'         Compiled Time HHMMSS
 00000E  F0F1  F1F0  F0F0                                    =C'011000'         Compiler Version

 000014  0074  ****                                          AL2(116),C'...'    Saved Options String
                           Timestamp and Version End
```

Figure 12. Example of a C++ compiler listing (Part 10 of 15)

```
15694A01 V1.11 z/OS XL C++                                          CCNUBRC: std::_EBCDIC:...)   04/29/09 04:55:13

 OFFSET OBJECT CODE       LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G

                                         std::_EBCDIC::_LFS_OFF::basic_ostrea
                                         m<char,std::char_traits<char> >& std
                                         ::_EBCDIC::_LFS_OFF::operator<<<std:
                                         :char_traits<char> >(std::_EBCDIC::_
                                         LFS_OFF::basic_ostream<char,std::cha
                                         r_traits<char> >&,const char*)
 000098                   000400 |    49           DS       0D
 000098  47F0  F001       000400 |    49           B        1(,r15)
 00009C  01C3C5C5                                           CEE eyecatcher
 0000A0  00000270                                           DSA size
 0000A4  0000DBF8                                           =A(PPA1-std::_EBCDIC::_LFS_OFF::basic_ostream<char,std::char_t...))
 0000A8  90E7  D00C       000400 |    49           STM      r14,r7,12(r13)
 0000AC  58E0  D04C       000400 |    49           L        r14,76(,r13)
 0000B0  4100  E270       000400 |    49           LA       r0,624(,r14)
 0000B4  5500  C314       000400 |    49           CL       r0,788(,r12)
 0000B8  4140  F050       000400 |    49           LA       r4,80(,r15)
 0000BC  47D0  F03A       000400 |    49           BNH      58(,r15)
 0000C0  58F0  C31C       000400 |    49           L        r15,796(,r12)
 0000C4  184E             000400 |    49           LR       r4,r14
 0000C6  05EF             000400 |    49           BALR     r14,r15
 0000C8  00000008                                           =F'8'
 0000CC  0540             000400 |    49           BALR     r4,0
 0000CE  4140  401A       000400 |    49           LA       r4,26(,r4)
 0000D2  58F0  C280       000400 |    49           L        r15,640(,r12)
 0000D6  90F0  E048       000400 |    49           STM      r15,r0,72(r14)
 0000DA  9210  E000       000400 |    49           MVI      0(r14),16
 0000DE  50D0  E004       000400 |    49           ST       r13,4(,r14)
 0000E2  5800  D014       000400 |    49           L        r0,20(,r13)
 0000E6  18DE             000400 |    49           LR       r13,r14
 0000E8                   End of Prolog

 0000E8  5000  D224       000000 |                 ST       r0,#CEECAACRENT_5(,r13,548)
 0000EC  5830  D224       000000 |                 L        r3,#CEECAACRENT_5(,r13,548)
 0000F0  5860  4AC4       000000 |                 L        r6,=Q(@STATIC)(,r4,2756)
 0000F4  4156  3000       000000 |                 LA       r5,=Q(@STATIC)(r6,r3,0)
 0000F8  5010  D0BC       000400 |    49           ST       r1,_O(,r13,188)
 0000FC  5020  D0C0       000400 |    49           ST       r2,_X(,r13,192)
 000100  4130  0000       000402 |    49           LA       r3,0
 000104  5030  D0D0       000402 |    49           ST       r3,__es.__l@0(,r13,208)
 000108  4100  52E0       000402 |    49           LA       r0,__fsm_tab(,r5,736)
 00010C  5000  D0D4       000402 |    49           ST       r0,__es.__t@4(,r13,212)
 000110  5030  D0D8       000402 |    49           ST       r3,__es.__this@8(,r13,216)
 000114  5030  D0DC       000402 |    49           ST       r3,__es.__i@c(,r13,220)
 000118  5030  D0EC       000404 |    49           ST       r3,_St(,r13,236)
 00011C  5820  D0C0       000405 |    49           L        r2,_X(,r13,192)
 000120  A526  7FFF       000147 |     4           NILH     r2,H'32767'
 000124  1812             000147 |     4           LR       r1,r2
 000126  4100  0000       000147 |     4           LA       r0,0
 00012A  B25E  0001       000147 |     4           SRST     r0,r1
 00012E  A714  FFFE       000147 |     4           JO       *-4
 000132  1810             000147 |     4           LR       r1,r0
 000134  1803             000147 |     4           LR       r0,r3
 000136  1F12             000147 |     4           SLR      r1,r2
 000138  5010  D0F0       000147 |     4           ST       r1,_N(,r13,240)
 00013C  5810  D0BC       000406 |    49           L        r1,_O(,r13,188)
 000140  1211             000406 |    49           LTR      r1,r1
⋮
```

Figure 12. Example of a C++ compiler listing (Part 11 of 15)

```
15694A01 V1.11 z/OS XL C++                                   CCNUBRC                  04/29/09 04:55:13

                                   E X T E R N A L   S Y M B O L   D I C T I O N A R Y

                       TYPE  ID  ADDR    LENGTH            NAME

                        SD    1 000000   0107B8            @STATICP
                        PR    2 000000   000660            @STATIC
                        PR    3 000000   000004            dateLen__4Date
                        PR    4 000000   000004            numMonths__4Date
                        PR    5 000000   000004            pCycle__9BioRhythm
                        PR    6 000000   000004            eCycle__9BioRhythm
                        PR    7 000000   000004            iCycle__9BioRhythm
                        PR    8 000000   000030            numDays__4Date
                        PR    9 000000   000004            _Psave__use_facet__Q3_3std7_EBCDIC8_
                                                           LFS_OFFHQ4_3std7_EBCDIC8_LFS_OFF7num
                                                           _putXTcTQ2_3std19ostreambuf_iterator
                                                           XTcTQ2_3std11char_traitsXTc____RCQ4_
                                                           3std7_EBCDIC8_LFS_OFF6locale_RCQ4_3s
                                                           td7_EBCDIC8_LFS_OFF7num_putXTcTQ2_3s
                                                           td19ostreambuf_iteratorXTcTQ2_3std11
                                                           char_traitsXTc_____2
⋮
```

Figure 12. Example of a C++ compiler listing (Part 12 of 15)

```
15694A01 V1.11 z/OS XL C++                                   CCNUBRC                  04/29/09 04:55:13

                             E X T E R N A L   S Y M B O L   C R O S S   R E F E R E N C E

           ORIGINAL NAME                                   EXTERNAL SYMBOL NAME

           @STATICP                                        @STATICP
           @STATIC                                         @STATIC
           Date::dateLen                                   dateLen__4Date
           Date::numMonths                                 numMonths__4Date
           BioRhythm::pCycle                               pCycle__9BioRhythm
           BioRhythm::eCycle                               eCycle__9BioRhythm
           BioRhythm::iCycle                               iCycle__9BioRhythm
           Date::numDays                                   numDays__4Date
           const std::_EBCDIC::_LFS_OFF                    _Psave__use_facet__Q3_3std7_EBCDIC8_
⋮
```

Figure 12. Example of a C++ compiler listing (Part 13 of 15)

```
15694A01 V1.11 z/OS XL C++                                          CCNUBRC                 04/29/09 04:55:13
                                * * * * *   S T O R A G E   O F F S E T   L I S T I N G   * * * * *

 IDENTIFIER          DEFINITION      ATTRIBUTES

 code                23-0:23         Class = automatic,           Location = 228(r13),
Length = 4

 bio                 22-0:22         Class = automatic,           Location = 192(r13),
Length = 24

 newTime             57-0:57         Class = automatic,           Location = 204(r13),
Length = 4

 lTime               56-0:56         Class = automatic,           Location = 200(r13),
Length = 4

 newTime             57-0:57         Class = automatic,           Location = 208(r13),
Length = 4

 lTime               56-0:56         Class = automatic,           Location = 204(r13),
Length = 4

 birthText           67-0:67         Class = parameter,           Location = 192(r13),
Length = 4

 newTime             57-0:57         Class = automatic,           Location = 212(r13),
Length = 4

 lTime               56-0:56         Class = automatic,           Location = 208(r13),
Length = 4

 totDays             78-0:78         Class = automatic,           Location = 260(r13),
Length = 4

 i                   80-0:80         Class = automatic,           Location = 256(r13),
Length = 4
⋮
```

Figure 12. Example of a C++ compiler listing (Part 14 of 15)

```
15694A01 V1.11 z/OS XL C++                                                  CCNUBRC                 04/29/09 04:55:13

                                               * * * * *   S T A T I C     M A P   * * * * *

 OFFSET (HEX)   LENGTH (HEX)   NAME

          0              C     __td__Q2_3std8bad_cast
         10              C     __td__Q2_3std9bad_alloc
         20             1C     __fsm_tab
         40             1C     __fsm_tab
         60             1C     __fsm_tab
         80             1C     __fsm_tab
         A0             1C     __fsm_tab
         C0             1C     __fsm_tab
         E0             1C     __fsm_tab
        100             1C     __fsm_tab
        120             1C     __fsm_tab
        140             1C     __fsm_tab
        160             1C     __fsm_tab
        180             1C     __fsm_tab
        1A0             1C     __fsm_tab
        1C0             30     __fsm_tab
        1F0             30     __fsm_tab
        220             30     __fsm_tab
        250             30     __fsm_tab
        280             30     __fsm_tab
        2B0             30     __fsm_tab
        2E0             44     __fsm_tab
        328             44     __fsm_tab
        370             44     __fsm_tab
        3B8             44     __fsm_tab
        400             44     __fsm_tab
        448             58     __fsm_tab
        4A0             58     __fsm_tab
        4F8             58     __fsm_tab
        550             58     __fsm_tab
        5A8              4     cerr__Q3_3std7_EBCDIC8_LFS_OFF_ptr
        5AC              4     cout__Q3_3std7_EBCDIC8_LFS_OFF_ptr
        5B0              4     time_ptr
        5B4              4     localtime_ptr
        5B8              4     asctime_ptr
        5BC              4     cin__Q3_3std7_EBCDIC8_LFS_OFF_ptr
        5C0              4     setw__Q3_3std7_EBCDIC8_LFS_OFFFi_ptr
        5C4              4     sscanf_ptr
        5C8              4     __ct__Q2_3std7_LockitFi_ptr
        5CC              4     _Id_cnt__Q5_3std7_EBCDIC8_LFS_OFF6locale2id_ptr
        5D0              4     __dt__Q2_3std7_LockitFv_ptr
⋮
```

| *Figure 12. Example of a C++ compiler listing (Part 15 of 15)*

z/OS XL C++ compiler listing components

The following information describes the components of a C++ compiler listing.These are available for regular and IPA compilations. Differences in the IPA versions of the listings are noted. “Using the IPA Link step listing” on page 317 describes IPA-specific listings.

Heading information

The first page of the listing is identified by the product number, the compiler version and release numbers, the name of the data set or z/OS UNIX System Services file containing the source code, the date and time compilation began (formatted according to the current locale), and the page number.

Note: If the name of the data set or z/OS UNIX file that contains the source code is greater than 32 characters, it is truncated. Only the right-most 32 characters appear in the listing.

Prolog section

The Prolog section provides information about the compile-time library, file identifiers, compiler options, and other items in effect when the compiler was invoked.

All options except those with no default (for example, DEFINE) are shown in the listing. Any problems with the compiler options appear after the body of the Prolog section.

IPA considerations: If you specify IPA suboptions that are irrelevant to the IPA Compile step, the Prolog does not display them. If IPA processing is not active, IPA suboptions do not appear in the Prolog.

The following information describes the optional parts of the listing and the compiler options that generate them.

Source Program

If you specify the SOURCE option, the listing file includes input to the compiler.

Note: If you specify the SHOWINC option, the source listing shows the included text after the `#include` directives.

Cross-Reference Listing

The XREF option generates a cross-reference table that contains a list of the identifiers from the source program. The table also displays a list of reference, modification, and definition information for each identifier.

The ATTR option generates a cross-reference table that contains a list of the identifiers from the source program, with a list of attributes for each identifier.

If you specify both ATTR and XREF, the cross-reference listing is a composite of the two forms. It contains the list of identifiers, as well as the attribute and reference, modification, and definition information for each identifier. The list is in the form:

```
identifier : attribute
       n:m (x)
```

where:

`n` corresponds to the file number from the INCLUDE LIST. If the identifier is from the main program, `n` is `0`.

`m` corresponds to the line number in the file *n*.

`x` is the cross reference code. It takes one of the following values:
R - referenced
D - defined
M - modified

together with the line numbers in which they appear.

Includes section

The compiler generates the Includes section when you use include files, and specify the options SOURCE, LIST, or INLRPT.

Messages

If the preprocessor or the compiler detects an error, or the possibility of an error, it generates messages. If you specify the SOURCE compiler option, preprocessor error messages appear immediately after the source statement in error. You can generate your own messages in the preprocessing stage by using `#error`. For information on `#error`, see the *z/OS XL C/C++ Language Reference*.

If you specify the compiler options FLAG(I), CHECKOUT or INFO(), the compiler will generate informational diagnostic messages.

For a description of compiler messages, see *z/OS XL C/C++ Messages*.

Message Summary

This listing section displays the total number of messages and the number of messages for each severity level.

Inline Report

If the OPTIMIZE and INLRPT options are specified, an Inline Report will be included in the listing. This report contains an inline summary and a detailed call structure.

Note: No report is produced when your source file contains only one defined subprogram.

The summary contains information such as:

- Name of each defined subprogram. Subprogram names appear in alphabetical order.
- Reason for action on a subprogram:
  - The P indicates that **#pragma noinline** and the COMPACT compiler option are not in effect.
  - The F indicates that the subprogram was declared inline, either by **#pragma inline** for C or the `inline` keyword for C++.
  - The C indicates that the COMPACT compiler option is specified for **#pragma_override(FuncName,″OPT(COMPACT,yes)**″ is specified in the source code.
  - The M indicates that C++ routine is an inline member routine.
  - The A indicates automatic inlining acted on the subprogram.
  - The - indicates there was no reason to inline the subprogram.
- Action on a subprogram:
  - Subprogram was inlined at least once.
  - Subprogram was not inlined because of initial size constraints.
  - Subprogram was not inlined because of expansion beyond size constraint.
  - Subprogram was a candidate for inlining, but was not inlined.
  - Subprogram was a candidate for inlining, but was not referenced.
  - This subprogram is directly recursive, or some calls have mismatching parameters

  Note: The ″Called″ and ″Calls″ in the actions section of the inline report, indicate how many times a function has been called or has called other functions, despite whether or not the callers or callees have been inlined.
- Status of original subprogram after inlining:
  - Subprogram is discarded because it is no longer referenced and is defined as `static` internal.
  - Subprogram was not discarded for various reasons :
    - Subprogram is external. (It can be called from outside the compilation unit.)
    - Some call to this subprogram remains.
    - Subprogram has its address taken.
- Initial relative size of subprogram (in Abstract Code Units (ACU)).

- Final relative size of subprogram (in ACUs) after inlining.
- Number of calls within the subprogram and the number of these calls that were inlined into the subprogram.
- Number of times the subprogram is called by others in the compile unit and the number of times this subprogram was inlined.
- Mode that is selected and the value of *threshold* and *limit* specified for this compilation.

The detailed call structure contains specific information of each subprogram such as:
- What subprograms it calls
- What subprograms call it
- In which subprograms it is inlined.

The information can help you to better analyze your program if you want to use the inliner in selective mode.

There may be additional messages as a result of the inlining. For example, if inlining a subprogram with automatic storage increases the automatic storage of the subprogram it is being inlined into by more than 4K, a message is emitted.

Pseudo Assembly Listing

The LIST compiler option generates a listing of the machine instructions in the object module in a form similar to assembler language.

This Pseudo Assembly listing displays the source statement line numbers and the line number of any inlined code to aid you in debugging inlined code.

External Symbol Dictionary

The LIST compiler option generates the External Symbol Dictionary. The External Symbol Dictionary lists the names that the compiler generates for the output object module. It includes address information and size information about each symbol.

External Symbol Cross Reference

The ATTR or XREF compiler options generate the External Symbol Cross Reference section. It shows the original name and corresponding mangled name for each symbol. For additional information on mangled names, see Chapter 14, “Filter utility,” on page 493.

Storage Offset Listing

If you specify the XREF option, the listing file includes offset information on identifiers.

Static Map

Static Map displays the contents of the @STATIC data area, which holds the file scope read/write static variables. It displays the offset (as a hexadecimal number), the length (as a hexadecimal number), and the names of the objects mapped to @STATIC. Under certain circumstances, the compiler may decide to map other objects to @STATIC.

If you specify the ATTR or XREF option, the listing file includes offset information for file scope read/write static variables.

Using the IPA Link step listing

The IPA Link step generates a listing file if you specify any of the following options:

- ATTR
- INLINE(,REPORT,,)
- INLRPT
- IPA(MAP)
- LIST
- XREF

Note: IPA does not support source listings or source annotations within Pseudo Assembly listings. The Pseudo Assembly listings do display the file and line number of the source code that contributed to a segment of pseudo assembly code.

Example of an IPA Link step listing

Figure 13 on page 318 shows an example of an IPA Link step listing.

```
5694-A01 V1.11 z/OS XL C/C++ IPA                                   DD:SYSIN                    04/29/2009 04:55:27   Page     1
                                      * * * * *   P R O L O G   * * * * *

  Compile Time Library . . . . . . : 410B0000
  Command options:
     Primary input name. . . . . . : DD:SYSIN
     Compiler options. . . . . . . : *IPA(LINK,MAP,LEVEL(1),DUP,ER,NONCAL,NOUPCASE,NOPDF1,NOPDF2,NOPDFNAME,NOCONTROL)
                                   : *NOGONUMBER  *NOHOT      *NOALIAS     *TERMINAL    *LIST        *XREF        *ATTR
                                   : *NOOFFSET    *MEMORY      *NOCSECT     *NODFP       *LIBANSI     *FLAG(I)
                                   : *NOTEST(NOSYM,NOBLOCK,NOLINE,NOPATH,HOOK)           *OPTIMIZE(2)
                                   : *INLINE(AUTO,REPORT,1000,8000)         *OPTFILE(DD:OPTIONS)      *NOSERVICE   *NOOE
                                   : *NOLOCALE    *HALT(16)    *NOGOFF      *NOSPLITLIST

                               * * * * *   E N D   O F   P R O L O G   * * * * *
15694-A01 V1.11 z/OS XL C/C++ IPA                                   DD:SYSIN                    04/29/2009 04:55:27   Page     2

                             * * * * *   O B J E C T   F I L E   M A P   * * * * *

 *ORIGIN  IPA  FILE ID  FILE NAME
    P              1   //DD:SYSIN
    PI     Y       2   USERID1.IPA.OBJECT(HELLO1)
    PI     Y       3   USERID1.IPA.OBJECT(HELLO2)
    PI     Y       4   USERID1.IPA.OBJECT(HELLO3)
    L              5   CEE.SCEELKED(PRINTF)
    L              6   CEE.SCEELKED(CEESG003)

 ORIGIN:  P=primary input     PI=primary INCLUDE        SI=secondary INCLUDE   IN=internal
          A=automatic call     U=UPCASE automatic call   R=RENAME card          L=C Library

                      * * * * *   E N D   O F   O B J E C T   F I L E   M A P   * * * * *
15694-A01 V1.11 z/OS XL C/C++ IPA                                   DD:SYSIN                    04/29/2009 04:55:27   Page     3

                        * * * * *   C O M P I L E R   O P T I O N S   M A P   * * * * *

  SOURCE FILE ID      COMPILE OPTIONS
              1        *AGGRCOPY(NOOVERLAP)      *NOALIAS     *ANSIALIAS   *ARCH(5)     *ARGPARSE    *NOASCII
                       *ASSERT(RESTRICT)         *BITFIELD(UNSIGNED)       *CHARS(UNSIGNED)          *NOCOMPACT   *NOCOMPRESS
                       *NOCONVLIT   *NOCSECT     *NODEBUG     *NODFP       *NODLL(NOCALLBACKANY)     *ENUMSIZE(SMALL)
                       *EXECOPS     *NOEXPORTALL *FLOAT(HEX,FOLD,NOMAF,NORRM,AFP(NOVOLATILE))        *NOGOFF      *NOGONUMBER
                       *NOHGPR(NOPRESERVE)       *NOHOT       *NOIGNERRNO  *ILP32       *NOINITAUTO
                       *INLINE(AUTO,NOREPORT,100,1000)        *IPA(NOLINK,NOOBJ,COM,OPT,NOGONUM)     *LANGLVL(EXTENDED)
                       *NOLIBANSI   *NOLOCALE    *LONGNAME    *MAXMEM(2097152)          *OPTIMIZE(2) *PLIST(HOST) *PREFETCH
                       *REDIR       *RENT        *NOROCONST   *ROUND(Z)    *ROSTRING    *NORTCHECK   *NOSERVICE   *SPILL(128)
                       *START       *STRICT      *NOSTRICT_INDUCTION       *TARGET(LE,zOSV1R11)      *TUNE(5)
                       *UNROLL(AUTO)             *NOUPCONV    *NOWSIZEOF   *NOXPLINK

              2        *AGGRCOPY(NOOVERLAP)      *NOALIAS     *ANSIALIAS   *ARCH(5)     *ARGPARSE    *NOASCII
                       *ASSERT(RESTRICT)         *BITFIELD(UNSIGNED)       *CHARS(UNSIGNED)          *NOCOMPACT   *NOCOMPRESS
                       *NOCONVLIT   *NOCSECT     *NODEBUG     *NODFP       *NODLL(NOCALLBACKANY)     *ENUMSIZE(SMALL)
                       *EXECOPS     *NOEXPORTALL *FLOAT(HEX,FOLD,NOMAF,NORRM,AFP(NOVOLATILE))        *NOGOFF      *NOGONUMBER
                       *NOHGPR(NOPRESERVE)       *NOHOT       *NOIGNERRNO  *ILP32       *NOINITAUTO
                       *INLINE(AUTO,NOREPORT,100,1000)        *IPA(NOLINK,NOOBJ,COM,OPT,NOGONUM)     *LANGLVL(EXTENDED)
                       *NOLIBANSI   *NOLOCALE    *LONGNAME    *MAXMEM(2097152)          *OPTIMIZE(2) *PLIST(HOST) *PREFETCH
                       *REDIR       *RENT        *NOROCONST   *ROUND(Z)    *ROSTRING    *NORTCHECK   *NOSERVICE   *SPILL(128)
                       *START       *STRICT      *NOSTRICT_INDUCTION       *TARGET(LE,zOSV1R11)      *TUNE(5)
                       *UNROLL(AUTO)             *NOUPCONV    *NOWSIZEOF   *NOXPLINK

              3        *AGGRCOPY(NOOVERLAP)      *NOALIAS     *ANSIALIAS   *ARCH(5)     *ARGPARSE    *NOASCII
                       *ASSERT(RESTRICT)         *BITFIELD(UNSIGNED)       *CHARS(UNSIGNED)          *NOCOMPACT   *NOCOMPRESS
                       *NOCONVLIT   *NOCSECT     *NODEBUG     *NODFP       *NODLL(NOCALLBACKANY)     *ENUMSIZE(SMALL)
                       *EXECOPS     *NOEXPORTALL *FLOAT(HEX,FOLD,NOMAF,NORRM,AFP(NOVOLATILE))        *NOGOFF      *NOGONUMBER
                       *NOHGPR(NOPRESERVE)       *NOHOT       *NOIGNERRNO  *ILP32       *NOINITAUTO
                       *INLINE(AUTO,NOREPORT,100,1000)        *IPA(NOLINK,NOOBJ,COM,OPT,NOGONUM)     *LANGLVL(EXTENDED)
                       *NOLIBANSI   *NOLOCALE    *LONGNAME    *MAXMEM(2097152)          *OPTIMIZE(2) *PLIST(HOST) *PREFETCH
                       *REDIR       *RENT        *NOROCONST   *ROUND(Z)    *ROSTRING    *NORTCHECK   *NOSERVICE   *SPILL(128)
                       *START       *STRICT      *NOSTRICT_INDUCTION       *TARGET(LE,zOSV1R11)      *TUNE(5)
                       *UNROLL(AUTO)             *NOUPCONV    *NOWSIZEOF   *NOXPLINK


                 * * * * *   E N D   O F   C O M P I L E R   O P T I O N S   M A P   * * * * *
```

Figure 13. Example of an IPA Link step listing (Part 1 of 7)

```
15694-A01 V1.11 z/OS XL C/C++ IPA                                 DD:SYSIN                04/29/2009 04:55:27   Page     4
                               * * * * *   I N L I N E   R E P O R T   * * * * *

                     IPA Inline Report (Summary)

  Reason:    P : #pragma noinline was specified for this routine
             F : #pragma inline was specified for this routine
             A : Automatic inlining
             C : Partition conflict
             N : Not IPA Object
             - : No reason
  Action:    I : Routine is inlined at least once
             L : Routine is initially too large to be inlined
             T : Routine expands too large to be inlined
             C : Candidate for inlining but not inlined
             N : No direct calls to routine are found in file (no action)
             U : Some calls not inlined due to recursion or parameter mismatch
             - : No action
  Status:    D : Internal routine is discarded
             R : A direct call remains to internal routine (cannot discard)
             A : Routine has its address taken (cannot discard)
             E : External routine (cannot discard)
             - : Status unchanged
  Calls/I      : Number of calls to defined routines / Number inline
  Called/I     : Number of times called / Number of times inlined

  Reason  Action  Status   Size (init)    Calls/I   Called/I     Name

     A      I       D      0 (40)          2/1       1/1        func2
     A      I       D      0 (32)          1/0       1/1        func3
     A      N       -      100 (28)        1/1       0          main
     N      -       E      0               0         2/0        PRINTF
  Mode = AUTO     Inlining Threshold =    1000    Expansion Limit =    8000
15694-A01 V1.11 z/OS XL C/C++ IPA                                 DD:SYSIN                04/29/2009 04:55:27   Page     5

                     IPA Inline Report (Call Structure)

  Defined Subprogram    : main
      Calls To(1,1)     : func2(1,1)
   Called From          : 0

  Defined Subprogram    : func2
      Calls To(2,1)     : func3(1,1)
                          PRINTF(1,0)
   Called From(1,1)     : main(1,1)

  Defined Subprogram    : PRINTF
      Calls To          : 0
   Called From(2,0)     : func3(1,0)
                          func2(1,0)

  Defined Subprogram    : func3
      Calls To(1,0)     : PRINTF(1,0)
   Called From(1,1)     : func2(1,1)


                        * * * * *   E N D   O F   I N L I N E   R E P O R T   * * * * *
```

Figure 13. Example of an IPA Link step listing (Part 2 of 7)

```
15694-A01 V1.11 z/OS XL C/C++ IPA                                Partition 1                04/29/2009 04:55:27   Page     6
                               * * * * *   P A R T I T I O N   M A P   * * * * *

 PARTITION 1 OF 1

 PARTITION SIZE:
    Actual: 17300
     Limit: 1572864

 PARTITION CSECT NAMES:
      Code: none
    Static: none
      Test: none

 PARTITION DESCRIPTION:
    Primary partition

 COMPILER OPTIONS FOR PARTITION 1:
   *AGGRCOPY(NOOVERLAP)      *NOALIAS     *ARCH(5)     *ARGPARSE    *ATTR        *NOCOMPACT   *NOCOMPRESS  *NOCSECT     *NODLL
   *EXECOPS     *FLOAT(HEX,FOLD,AFP)      *NOGOFF      *NOGONUMBER  *NOIGNERRNO  *ILP32       *NOINITAUTO
   *INLINE(AUTO,REPORT,1000,8000)         *IPA(LINK)   *LIBANSI     *LIST        *NOLOCALE    *LONGNAME    *MAXMEM(2097152)
   *OPTIMIZE(2) *PLIST(HOST) *PREFETCH    *REDIR       *RENT        *NOROCONST   *SPILL(128)  *START       *STRICT
   *NOSTRICT_INDUCTION       *NOTEST      *TUNE(5)     *NOXPLINK    *XREF

 SYMBOLS IN PARTITION 1:

  *TYPE     FILE ID    SYMBOL
     D           1     seen_main
     D           1     string1
     D           2     seen_func2
     D           2     string2
     D           3     seen_func3
     F           1     main

   TYPE:  F=function     D=data

 SOURCE FILES FOR PARTITION 1:

  *ORIGIN     FILE ID     SOURCE FILE NAME

      P            1      //'USERID1.IPA.SOURCE(HELLO1)'
      P            2      //'USERID1.IPA.SOURCE(HELLO2)'
      P            3      //'USERID1.IPA.SOURCE(HELLO3)'

  ORIGIN:  P=primary input     PI=primary INCLUDE


                        * * * * *   E N D   O F   P A R T I T I O N   M A P   * * * * *
```

Figure 13. Example of an IPA Link step listing (Part 3 of 7)

```
15694-A01 V1.11 z/OS XL C/C++ IPA                              Partition 1                  04/29/2009 04:55:27   Page     7

 OFFSET OBJECT CODE        LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G

                           Timestamp and Version Information
 000000  F2F0  F0F9                                      =C'2009'          Compiled Year
 000004  F0F4  F2F9                                      =C'0429'          Compiled Date MMDD
 000008  F0F4  F5F5  F2F4                                =C'045524'        Compiled Time HHMMSS
 00000E  F0F1  F1F1  F0F0                                =C'011100'        Compiler Version

 000014  0096  ****                                      AL2(150),C'...'   Saved Options String
                           Timestamp and Version End

15694-A01 V1.11 z/OS XL C/C++ IPA                              Partition 1: main            04/29/2009 04:55:27   Page     8

 OFFSET OBJECT CODE        LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G

 0000B8                    000010 |    1   main     DS       0D
 0000B8  47F0  F022        000010 |    1            B        34(,r15)
 0000BC  01C3C5C5                                            CEE eyecatcher
 0000C0  000000A8                                            DSA size
 0000C4  000000E8                                            =A(PPA1-main)
 0000C8  47F0  F001        000010 |    1            B        1(,r15)
 0000CC  58F0  C31C        000010 |    1            L        r15,796(,r12)
 0000D0  184E              000010 |    1            LR       r4,r14
 0000D2  05EF              000010 |    1            BALR     r14,r15
 0000D4  00000000                                            =F'0'
 0000D8  07F3              000010 |    1            BR       r3
 0000DA  90E6  D00C        000010 |    1            STM      r14,r6,12(r13)
 0000DE  58E0  D04C        000010 |    1            L        r14,76(,r13)
 0000E2  4100  E0A8        000010 |    1            LA       r0,168(,r14)
 0000E6  5500  C314        000010 |    1            CL       r0,788(,r12)
 0000EA  4130  F03A        000010 |    1            LA       r3,58(,r15)
 0000EE  4720  F014        000010 |    1            BH       20(,r15)
 0000F2  58F0  C280        000010 |    1            L        r15,640(,r12)
 0000F6  90F0  E048        000010 |    1            STM      r15,r0,72(r14)
 0000FA  9210  E000        000010 |    1            MVI      0(r14),16
 0000FE  50D0  E004        000010 |    1            ST       r13,4(,r14)
 000102  18DE              000010 |    1            LR       r13,r14
 000104                    End of Prolog

 000104  5820  C1F4        000000 |                 L        r2,_CEECAA_(,r12,500)
 000108  5840  308E        000012 |    1            L        r4,=Q(@STATIC)(,r3,142)
 00010C  C050  0000  003E  000000 |                 LARL     r5,F'62'
 000112  58F0  3092        000013 |    2  +         L        r15,=V(printf)(,r3,146)
 000116  4110  D098        000013 |    2  +         LA       r1,#MX_TEMP1(,r13,152)
 00011A  5804  2000        000012 |    1            L        r0,seen_main(r4,r2,0)
 00011E  5864  2008        000011 |    2  +         L        r6,seen_func2(r4,r2,8)
 000122  58E4  2004        000013 |    1            L        r14,string1(r4,r2,4)
 000126  50E0  D09C        000013 |    2  +         ST       r14,#MX_TEMP1(,r13,156)
 00012A  41E0  500F        000013 |    2  +         LA       r14,+CONSTANT_AREA(,r5,15)
 00012E  A70A  0001        000012 |    1            AHI      r0,H'1'
 000132  A76A  0001        000011 |    2  +         AHI      r6,H'1'
 000136  50E0  D098        000013 |    2  +         ST       r14,#MX_TEMP1(,r13,152)
 00013A  5004  2000        000012 |    1            ST       r0,seen_main(r4,r2,0)
 00013E  5064  2008        000011 |    2  +         ST       r6,seen_func2(r4,r2,8)
 000142  0DEF              000013 |    2  +         BASR     r14,r15
 000144  4100  5013        000017 |    3  +         LA       r0,+CONSTANT_AREA(,r5,19)
 000148  5000  D098        000017 |    3  +         ST       r0,#MX_TEMP1(,r13,152)
 00014C  5804  2010        000015 |    3  +         L        r0,seen_func3(r4,r2,16)
 000150  5814  200C        000013 |    2  +         L        r1,string2(r4,r2,12)
 000154  A70A  0001        000015 |    3  +         AHI      r0,H'1'
 000158  58F0  3092        000017 |    3  +         L        r15,=V(printf)(,r3,146)
 00015C  5010  D09C        000017 |    3  +         ST       r1,#MX_TEMP1(,r13,156)
 000160  4110  D098        000017 |    3  +         LA       r1,#MX_TEMP1(,r13,152)
 000164  5004  2010        000015 |    3  +         ST       r0,seen_func3(r4,r2,16)
 000168  0DEF              000017 |    3  +         BASR     r14,r15
 00016A  41F0  0000        000015 |    1            LA       r15,0
 00016E                    000015 |    1   @1L3     DS       0H

 00016E                    Start of Epilog
 00016E  180D              000016 |    1            LR       r0,r13
 000170  58D0  D004        000016 |    1            L        r13,4(,r13)
```

Figure 13. Example of an IPA Link step listing (Part 4 of 7)

```
15694-A01 V1.11 z/OS XL C/C++ IPA                                Partition 1: main              04/29/2009 04:55:27   Page     9

 OFFSET OBJECT CODE        LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G

 000174  58E0  D00C        000016 |     1            L        r14,12(,r13)
 000178  9826  D01C        000016 |     1            LM       r2,r6,28(r13)
 00017C  051E              000016 |     1            BALR     r1,r14
 00017E  0707              000016 |     1            NOPR     7

 000180                    Start of Literals
 000180  00000000                                             =Q(@STATIC)
 000184  00000000                                             =V(printf)
 000188                    End of Literals

                           ***   General purpose registers used: 1111111000001111
                           ***   Floating point  registers used: 1111111100000000
                           ***   Size of register spill area: 128(max) 0(used)
                           ***   Size of dynamic storage: 168
                           ***   Size of executable code: 200

                           Constant Area
 000188  C8859393 96000000 A6969993 845A006C    |Hello...world!.%|
 000198  A240006C A21500                        |s .%s..         |
15694-A01 V1.11 z/OS XL C/C++ IPA                                Partition 1                    04/29/2009 04:55:27   Page    10

 OFFSET OBJECT CODE        LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G

                           PPA1: Entry Point Constants
 0001A0  1CCEA106                                             =F'483303686'      Flags
 0001A4  00000128                                             =A(PPA2-main)
 0001A8  00000000                                             =F'0'              No PPA3
 0001AC  00000000                                             =F'0'              No EPD
 0001B0  FF800000                                             =F'-8388608'       Register save mask
 0001B4  00000000                                             =F'0'              Member flags
 0001B8  90                                                   =AL1(144)          Flags
 0001B9  000000                                               =AL3(0)            Callee's DSA use/8
 0001BC  0040                                                 =H'64'             Flags
 0001BE  0012                                                 =H'18'             Offset/2 to CDL
 0001C0  00000000                                             =F'0'              Reserved
 0001C4  50000064                                             =F'1342177380'     CDL function length/2
 0001C8  FFFFFF18                                             =F'-232'           CDL function EP offset
 0001CC  38260000                                             =F'942014464'      CDL prolog
 0001D0  4009005B                                             =F'1074331739'     CDL epilog
 0001D4  00000000                                             =F'0'              CDL end
 0001D8  0004  ****                                           AL2(4),C'main'
                           PPA1 End

                           PPA2: Compile Unit Block
 0001E0  0300  2203                                           =F'50340355'       Flags
 0001E4  FFFF  FE20                                           =A(CEESTART-PPA2)
 0001E8  0000  0000                                           =F'0'              No PPA4
 0001EC  FFFF  FE20                                           =A(TIMESTMP-PPA2)
 0001F0  0000  0000                                           =F'0'              No primary
 0001F4  0200  0000                                           =F'33554432'       Flags
                           PPA2 End
```

Figure 13. Example of an IPA Link step listing (Part 5 of 7)

```
15694-A01 V1.11 z/OS XL C/C++ IPA                               Partition 1                04/29/2009 04:55:27   Page    11

                                   E X T E R N A L   S Y M B O L   D I C T I O N A R Y

                       TYPE  ID  ADDR    LENGTH           NAME

                        SD    1 000000   0001F8           @STATICP
                        PR    2 000000   000014           @STATIC
                        LD    0 0000B8   000001           main
                        ER    3 000000                    CEESG003
                        ER    4 000000                    PRINTF
                        ER    5 000000                    CEESTART
                        SD    6 000000   000008           @@PPA2
                        SD    7 000000   00000C           CEEMAIN
                        ER    8 000000                    EDCINPL
15694-A01 V1.11 z/OS XL C/C++ IPA                               Partition 1                04/29/2009 04:55:27   Page    12

                             E X T E R N A L   S Y M B O L   C R O S S   R E F E R E N C E

           ORIGINAL NAME                                   EXTERNAL SYMBOL NAME

           @STATICP                                        @STATICP
           @STATIC                                         @STATIC
           main                                            main
           CEESG003                                        CEESG003
           printf                                          PRINTF
           CEESTART                                        CEESTART
           @@PPA2                                          @@PPA2
           CEEMAIN                                         CEEMAIN
           EDCINPL                                         EDCINPL
15694-A01 V1.11 z/OS XL C/C++ IPA                               Partition 1                04/29/2009 04:55:27   Page    13

                                             * * * * *   S T A T I C    M A P   * * * * *

 OFFSET (HEX)   LENGTH (HEX)   NAME

          0              4    seen_main
          4              4    string1
          8              4    seen_func2
          C              4    string2
         10              4    seen_func3

                                    * * * * *   E N D    O F    S T A T I C    M A P   * * * * *
```

Figure 13. Example of an IPA Link step listing (Part 6 of 7)

```
15694-A01 V1.11 z/OS XL C/C++ IPA                                Partition 1               04/29/2009 04:55:27   Page    14
                             * * * * *  S O U R C E   F I L E   M A P  * * * * *

             OBJECT      SOURCE
  *ORIGIN   FILE ID     FILE ID     SOURCE FILE NAME
      P         2           1      //'USERID1.IPA.SOURCE(HELLO1)'
                                     - Compiled by 15694A01   1B00
                                                on 04/29/2009 04:55:24
      P         3           2      //'USERID1.IPA.SOURCE(HELLO2)'
                                     - Compiled by 15694A01   1B00
                                                on 04/29/2009 04:55:25
      P         4           3      //'USERID1.IPA.SOURCE(HELLO3)'
                                     - Compiled by 15694A01   1B00
                                                on 04/29/2009 04:55:26


  ORIGIN:  P=primary input     PI=primary INCLUDE

                      * * * * *  E N D   O F   S O U R C E   F I L E   M A P  * * * * *
15694-A01 V1.11 z/OS XL C/C++ IPA                                Partition 1               04/29/2009 04:55:27   Page    15

                                    * * * * *  M E S S A G E S  * * * * *

  MESSAGE CODE   PAGE   MESSAGE TEXT

                             * * * * *  E N D   O F   M E S S A G E S  * * * * *
15694-A01 V1.11 z/OS XL C/C++ IPA                                Partition 1               04/29/2009 04:55:27   Page    16

                             * * * * *  M E S S A G E   S U M M A R Y  * * * * *

 TOTAL   UNRECOVERABLE  SEVERE       ERROR     WARNING    INFORMATIONAL
                (U)       (S)         (E)        (W)          (I)
     0           0         0           0          0            0

                      * * * * *  E N D   O F   M E S S A G E   S U M M A R Y  * * * * *

                          * * * * *  E N D   O F   C O M P I L A T I O N  * * * * *
```

| *Figure 13. Example of an IPA Link step listing (Part 7 of 7)*

IPA Link step listing components

The following information describes the components of an IPA Link step listing.

Heading information

The first page of the listing is identified by the product number, the compiler version and release numbers, the central title area, the date and time compilation began (formatted according to the current locale), and the page number.

In the following listing sections, the central title area will contain the primary input file identifier:

- Prolog
- Object File Map
- Source File Map
- Compiler Options Map
- Global Symbols Map
- Inline Report
- Messages
- Message Summary

In the following listing sections, the central title area will contain the phrase Partition `nnnn`, where `nnnn` specifies the partition number:

- Partition Map

In the following listing sections, the title contains the phrase Partition `nnnn:name`. `nnnn` specifies the partition number, and `name` specifies the name of the first function in the partition:

- Pseudo Assembly Listing
- External Symbol Cross Reference
- Storage Offset Listing

Prolog section

The Prolog section of the listing provides information about the compile-time library, file identifiers, compiler options, and other items in effect when the IPA Link step was invoked.

The listing displays all compiler options except those with no default (for example, DEFINE). If you specify IPA suboptions that are irrelevant to the IPA Link step, the Prolog does not display them. Any problems with compiler options appear after the body of the Prolog section and before the End of Prolog section.

Object File Map

The Object File Map displays the names of the object files that were used as input to the IPA Link step. Specify any of the following options to generate the Object File Map:

- IPA(MAP)
- LIST

Other listing sections, such as the Source File Map, use the File ID numbers that appear in this listing section.

z/OS UNIX System Services file names that are too long to fit into a single listing record continue on subsequent listing records.

Source File Map

The Source File Map listing section identifies the source files that are included in the object files. The IPA Link step generates this section if you specify any of the following options:

- IPA(MAP)
- LIST

The IPA Link step formats the compilation date and time according to the locale you specify with the LOCALE option in the IPA Link step. If you do not specify the LOCALE option, it uses the default locale.

This section appears near the end of the IPA Link step listing. If the IPA Link step terminates early due to errors, it does not generate this section.

Compiler Options Map

The Compiler Options Map listing section identifies the compiler options that were specified during the IPA Compile step for each compilation unit that is encountered when the object file is processed. For each compilation unit, it displays the final options that are relevant to IPA Link step processing. You may have specified these options through a compiler option or `#pragma` directive, or you may have picked them up as defaults.

The IPA Link step generates this listing section if you specify the IPA(MAP) option.

Global Symbols Map

The Global Symbols Map listing section shows how global symbols are mapped into members of global data structures by the global variable coalescing optimization process.

Each global data structure is limited to 16 MB by the z/OS object architecture. If an application has more than 16 MB of data, IPA Link must generate multiple global data structures for the application. Each global data structure is assigned a unique name.

The Global Symbols Map includes symbol information and file name information (file name information may be approximate). In addition, line number information is available for C compilations if you specified any of the following options during the IPA Compile step:

- XREF
- IPA(XREF)
- XREF(ATTRIBUTE)

The IPA Link step generates this listing section if you specify the IPA(MAP) option and the IPA Link step causes global symbols to be coalesced. The Global Symbols Map is only added to the IPA Link step listing if the IPA Link phase optimization changes the structure and/or layout of the global symbols utilized by the final module. If no changes are made, then the Global Symbols Map is not included in the listing.

Inline Report for IPA inliner

The Inline Report describes the actions that are performed by the IPA Inliner. The IPA Link step generates this listing section if you specify the INLINE(,REPORT,,), NOINLINE(,REPORT,,), or INLRPT option.

This report is similar to the one that is generated by the non-IPA inliner. In the IPA version of this report, the term 'subprogram' is equivalent to a C/C++ function or a C++ method. The summary contains information such as:

- Name of each defined subprogram. IPA sorts subprogram names in alphabetical order.
- Reason for action on a subprogram:
  - A **#pragma noinline** was specified for the subprogram. The P indicates that inlining could not be performed.
  - `inline` was specified for the subprogram. For z/OS XL C++, this is a result of the inline specifier. For C, this a result of the **#pragma inline**. The F indicates that the subprogram was declared inline.
  - The IPA Link step performed auto-inlining on the subprogram.
  - There was no reason to inline the subprogram.
  - There was a partition conflict.
  - The IPA Link step could not inline the object module because it was a non-IPA object module.
- Action on a subprogram:
  - IPA inlined subprogram at least once.
  - IPA did not inline subprogram because of initial size constraints.
  - IPA did not inline subprogram because of expansion beyond size constraint.
  - Subprogram was a candidate for inlining, but IPA did not inline it.
  - Subprogram was a candidate for inlining, but was not referenced.
  - The subprogram is directly recursive, or some calls have mismatched parameters.
- Status of original subprogram after inlining:
  - IPA discarded the subprogram because it is no longer referenced and is defined as `static` internal.
  - IPA did not discard the subprogram, for various reasons :
    - Subprogram is external. (It can be called from outside the compilation unit.)

- Subprogram call to this subprogram remains.
- Subprogram has its address taken.

- Initial relative size of subprogram (in Abstract Code Units (ACUs)).
- Final relative size of subprogram (in ACUs) after inlining.
- Number of calls within the subprogram and the number of these calls that IPA inlined into the subprogram.
- Number of times the subprogram is called by others in the compile unit and the number of times IPA inlined the subprogram.
- Mode that is selected and the value of *threshold* and *limit* you specified for the compilation.

Static functions whose names are not unique within the application as a whole will have names prefixed with `nnnn:`, where `nnnn` is the source file number.

The detailed call structure contains specific information of each subprogram such as:
- Subprograms that it calls
- Subprograms that call it
- Subprograms in which it is inlined.

The information can help you to better analyze your program if you want to use the inliner in selective mode.

Inlining may result in additional messages. For example, if inlining a subprogram with automatic storage increases the automatic storage of the subprogram it is being inlined into by more than 4K, the IPA Link step issues a message.

This report may display information about inlining specific subprograms, at the point at which IPA determines that inlining is impossible.

The counts in this report do not include calls from non-IPA to IPA programs.

Note: Even if the IPA Link step did not perform any inlining, it generates the IPA Inline Report if you request it.

Partition Map

The Partition Map listing section describes each of the object code partitions the IPA Link step creates. It provides the following information:
- The reason for generating each partition
- How the code is packaged (the CSECTs)
- The options used to generate the object code
- The function and global data included in the partition
- The source files that were used to create the partition

The IPA Link step generates this listing section if you specify either of the following options :
- IPA(MAP)
- LIST

The Pseudo Assembly, External Symbol Dictionary, External Symbol Cross Reference, and Storage Offset listing sections follow the Partition Map listing section for the partition, if you have specified the appropriate compiler options.

Pseudo Assembly Listing

The LIST compiler option generates a listing of the machine instructions in the current partition of the object module, in a form similar to assembler language.

This Pseudo Assembly listing displays the source statement line numbers and the line number of inlined code to aid you in debugging inlined code. Refer to "GONUMBER | NOGONUMBER" on page 123, "IPA | NOIPA" on page 139, and "LIST | NOLIST" on page 163 for information about source and line numbers in the listing section.

External Symbol Dictionary

The External Symbol Dictionary lists the names that the IPA Link step generates for the current partition of the object module. It includes address information and size information about each symbol.

External Symbol Cross Reference

The IPA Link step generates this section if you specify the ATTR or XREF compiler option. It shows how the IPA Link step maps internal and ESD names for external symbols that are defined or referenced in the current partition of the object module.

Storage Offset Listing

The Storage Offset listing section displays the offsets for the data in the current partition of the object module.

During the IPA Compile step, the compiler saves symbol storage offset information in the IPA object file as follows:

- For C, if you specify the XREF, IPA(ATTRIBUTE), IPA(XREF) options, or the **#pragma options(XREF)**
- For C++, if you specify the ATTR, XREF, IPA(ATTRIBUTE), or IPA(XREF) options

If this is done and the compilation unit includes variables, the IPA Link step may generate a Storage Offset listing.

If you specify the ATTR or XREF option on the IPA Link step, and any of the compilation units that contributed variables to a particular partition had storage offset information encoded in the IPA object file, the IPA Link step generates a Storage Offset listing section for that partition.

The Storage Offset listing displays the variables that IPA did not coalesce. The symbol definition information appears as `file#:line#`.

Static Map

If you specify the ATTR or XREF option, the listing file includes offset information for file scope read/write static variables.

Messages

If the IPA Link step detects an error, or the possibility of an error, it issues one or more diagnostic messages, and generates the Messages listing section. This listing section contains a summary of the messages that are issued during IPA Link step processing.

The IPA Link step listing sorts the messages by severity. The Messages listing section displays the listing page number where each message was originally shown. It also displays the message text, and optionally, information relating the error to a file name, line (if known), and column (if known).

For more information on compiler messages, see “FLAG | NOFLAG” on page 113, and *z/OS XL C/C++ Messages*.

Message Summary

This listing section displays the total number of messages and the number of messages for each severity level.

The following tables show the components that are included in the listing depending on which option is specified:

Table 34. IPA Link step listing components

Listing Component	`-Wl, I, ATTR`	`-Wl, I, INLINE (, REPORT ,,)`	`-Wl, I, INLRPT (destination)`	`-Wl, I, IPA (MAP)`	`-Wl, I, LIST (destination)`	`-Wl, I, XREF`	`-V`	**Phase**
Compiler Options Map				✓			✓	IPA Link
Cross-Reference Table							✓	Binder
Entry Point and Alias Summary							✓	Binder
External Symbol Cross-Reference	✓					✓	✓	Backend
External Symbol Dictionary					✓		✓	Backend
Global Symbols Map **				✓			✓	IPA Link
Imported and Exported Symbols							✓	Binder
Inline Report		✓	✓					IPA Link
Input List							✓	Binder
Message Summary	✓	✓	✓	✓	✓	✓	✓	IPA Link
Message Summary Report							✓	Binder
Messages *	✓	✓	✓	✓	✓	✓	✓	IPA Link
Module Map							✓	Binder
Object File Map				✓			✓	IPA Link

Table 34. IPA Link step listing components (continued)

Listing Component	`-Wl, I, ATTR`	`-Wl, I, INLINE (, REPORT ,,)`	`-Wl, I, INLRPT (des tina tion)`	`-Wl, I, IPA (MAP)`	`-Wl, I, LIST (des tina tion)`	`-Wl, I, XREF`	`-V`	**Phase**
Partition Map				✓			✓	IPA Link
Processing Options							✓	Binder
Prolog	✓	✓	✓	✓	✓	✓	✓	IPA Link
Pseudo Assembly Listing					✓		✓	Backend
Save Module Attributes							✓	Binder
Save Operation Summary							✓	Binder
Source File Map				✓			✓	IPA Link
Storage Offset Listing							✓	Backend

* This section is only generated if diagnostic messages are issued.

** This section is only generated if the IPA Link phase coalesces global variables.

Chapter 5. Binder options and control statements

This information lists the binder options, suboptions, and control statements that are considered important for a C or C++ programmer. For a detailed description of all the binder options and control statements, see *z/OS MVS Program Management: User's Guide and Reference*.

C or C++ programmers should be familiar with the following binder options and relevant suboptions:

- ALIASES
- AMODE
- CALL
- CASE
- COMPAT
- DYNAM
- INFO
- LET
- LIST
- LISTPRIV
- MAP
- OPTIONS
- REUS
- RMODE
- UPCASE
- XREF

C or C++ programmers should be familiar with the following control statements:

- AUTOCALL
- ENTRY
- IMPORT
- INCLUDE
- LIBRARY
- NAME
- RENAME

© Copyright IBM Corp. 1998, 2009

Chapter 6. Run-Time options

This information describes how to specify run-time options and **#pragma runopts** preprocessor directives available to you with z/OS XL C/C++ and the Language Environment element. For a detailed description of the Language Environment run-time options and information about how to apply them in different environments, refer to *z/OS Language Environment Programming Reference*.

Specifying run-time options

To allow your application to recognize run-time options, either the EXECOPS compiler option, or the **#pragma runopts(execops)** directive must be in effect. The default compiler option is EXECOPS.

You can specify run-time options as follows:

- At execution time in one of the following ways:
  - On the GPARM option of the IBM-supplied cataloged procedures
  - On the option list of the TSO CALL command
  - On the PARM parameter of the EXEC PGM=*your-program-name* JCL statement
  - On the exported _CEE_RUNOPTS environment variable under the z/OS shell
- At compile time, on a **#pragma runopts** directive in your `main` program

If EXECOPS is in effect, use a slash '/' to separate run-time options from arguments that you pass to the application. For example:

```
GPARM='STORAGE(FE,FE,FE)/PARM1,PARM2,PARM3'
```

If EXECOPS is in effect, the Language Environment run-time environment interprets the character string that precedes the slash as run-time options. It passes the character string that follows the slash to your application as arguments. If no slash separates the arguments, the Language Environment run-time environment interprets the entire string as an argument.

If EXECOPS is not in effect, the Language Environment run-time environment passes the entire string to your application.

If you specify two or more contradictory options (for example in a **#pragma runopts** statement), the last option that is encountered is accepted. Run-time options that you specify at execution time have higher precedence than those specified at compile time.

For more information on the precedence and specification of run-time options for applications that are compiled with the Language Environment run-time environment, refer to *z/OS Language Environment Programming Reference*.

Using the #pragma runopts preprocessor directive

You can use the **#pragma runopts** preprocessor directive to specify Language Environment run-time options. You can also use **#pragma runopts** to specify the run-time options ARGPARSE, ENV, PLIST, REDIR, and EXECOPS, which have matching compiler options. If you specify the compiler option, it takes precedence over the **#pragma runopts** directive.

© Copyright IBM Corp. 1998, 2009

When the run-time option EXECOPS is in effect, you can specify run-time options at execution time, as previously described. These options override run-time options that you compiled into the program by using the **#pragma runopts** directive.

You can specify multiple run-time options per directive or multiple directives per compilation unit. If you want to specify the ARGPARSE or REDIR options, the **#pragma runopts** directive must be in the same compilation unit as `main()`. Neither run-time option has an effect on programs invoked under the z/OS shell. This is because the shell program handles the parsing and redirection of command line arguments within that environment. Even though you can specify this directive in multiple compilation units, the specification that will take effect depends on the order of linking. It is advisable to specify it only once, and in the same compilation unit as `main()`.

When you specify multiple instances of **#pragma runopts** in separate compilation units, the compiler generates a CSECT for each compilation unit that contains a **#pragma runopts** directive. When you link multiple compilation units that specify **#pragma runopts**, the linkage editor takes only the first CSECT, thereby ignoring your other option statements. Therefore, you should always specify your **#pragma runopts** directive in the same source file that contains the function `main()`.

For more information on the **#pragma runopts** preprocessor directive, see *z/OS XL C/C++ Language Reference*.

Chapter 7. Compiling

This information describes how to compile your program with the z/OS XL C/C++ compiler and the Language Environment services. For specific information about compiler options, see Chapter 4, “Compiler Options,” on page 41.

The z/OS XL C/C++ compiler analyzes the source program and translates the source code into machine instructions that are known as *object code*.

You can perform compilations under z/OS batch, TSO, or the z/OS UNIX System Services environment.

Note: As of z/OS V1R5 C/C++, the compiler will only work if both the SCEERUN and SCEERUN2 Language Environment libraries are available.

Input to the z/OS XL C/C++ compiler

The following information describes how to specify input to the z/OS XL C/C++ compiler for a regular compilation, or the IPA Compile step. For more information about input for IPA, refer to Chapter 8, “Using the IPA Link step with z/OS XL C/C++ programs,” on page 375.

If you are compiling a C or C++ program, input for the compiler consists of the following:
- Your z/OS XL C/C++ source program
- The z/OS XL C/C++ standard header files including IBM-supplied Class Library header files
- Your header files

When you invoke the z/OS XL C/C++ compiler, the operating system locates and runs the compiler. To run the compiler, you need the following default data sets, which are supplied by IBM:
- CBC.SCCNCMP
- CEE.SCEERUN
- CEE.SCEERUN2

The locations of the compiler and the run-time library were determined by the system programmer who installed the product. The compiler and library should be in the STEPLIB, JOBLIB, LPA, or LNKLST concatenations. LPA can be from either specific modules (`IEALPAxx`) or a list (`LPALSTxx`). See the cataloged procedures shipped with the product in Chapter 12, “Cataloged procedures and REXX EXECs,” on page 465.

Note: For z/OS UNIX System Services file names, unless they appear in JCL, file names, which contain the special characters blank, backslash, and double quotation mark, must escape these characters. The escape character is backslash (\).

Primary input

For a C or C++ program, the primary input to the compiler is the data set that contains your XL C/C++ source program. If you are running the compiler in batch, identify the input source program with the SYSIN DD statement. You can do this by either defining the data set that contains the source code or by placing your source code directly in the JCL stream. In TSO or in z/OS UNIX System Services, identify

the input source program by name as a command line argument. The primary input source file can be any one of the following:
- A sequential data set
- A member of a partitioned data set
- All members of a partitioned data set
- A z/OS UNIX file
- All files in a z/OS UNIX directory

Secondary input

For a C or C++ program, secondary input to the compiler consists of data sets or directories that contain include files. Use the LSEARCH and SEARCH compiler options, or the SYSLIB DD statement when compiling in batch, to specify the location of the include files.

For more information on the use of these compiler options, see “LSEARCH | NOLSEARCH” on page 171 and “SEARCH | NOSEARCH” on page 219. For more information on naming include files, see “Specifying include file names” on page 361. For information on how the compiler searches for include files, see “Search sequences for include files” on page 369. For more information on include files, refer to “Using include files” on page 360.

Note: The LRECL for the SCLBH.H data set has changed from 80 to 120. You should ensure that SCLBH.H is the first data set in your SYSLIB concatenation. Do not use the SYSLIB concatenation to search for C++ header files with the compiler because searching the SYSLIB concatenation cannot distinguish between the old UNIX System Laboratories header files and new ISO Standard Library header files. For example, `#include <iostream.h>` (old USL) and `#include <iostream>` (ISO Standard) are indistinguishable using the SYSLIB concatenation. Use the SEARCH compiler option so that the correct header files are included.

Output from the compiler

You can specify compiler output files as one or more of the following:
- A sequential data set
- A member of a partitioned data set
- A partitioned data set
- A z/OS UNIX file
- A z/OS UNIX directory

For valid combinations of input file types and output file types, refer to Table 37 on page 339.

Specifying output files

You can use compile options to specify compilation output files as follows:

Table 35. Compile options that provide output file names

Output File Type	**Compiler Option**
Object Module	OBJECT(*filename*)
Listing File	SOURCE (*filename*), LIST(*filename*), INLRPT(*filename*) (Note: All listings must go to the same file. The last given location is used.)
Preprocessor Output	PPONLY(*filename*)

Table 35. Compile options that provide output file names (continued)

Output File Type	Compiler Option
Events File	EVENTS(*filename*)
Template Output	TEMPINC(*location*)
Template Registry	TEMPLATEREGISTRY(*filename*)

When compiler options that generate output files are specified without suboptions to identify the output files, and, in the case of a batch job, the designated ddnames are not allocated, the output file names are generated based on the name of the source file.

Note: The exception to this case is Template Registry, which is fixed to `templreg`, and Template Output, which is fixed to `tempinc`.

For data sets, the compiler generates a low-level qualifier by appending a suffix to the data set name of the source, as Table 36 shows.

If you compile source from z/OS UNIX System Services files without specifying output file names in the compiler options, the compiler writes the output files to the current working directory. The compiler does the following to generate the output file names:

- Appends a suffix, if it does not exist
- Replaces the suffix, if it exists

The following default suffixes are used:

Table 36. Defaults for output file types

Output File Type	z/OS File	z/OS UNIX File
Object Module	OBJ	o
Listing File	LIST	lst
Preprocessor Output	EXPAND	i
Template Output	TEMPINC	./tempinc
Template Registry	TEMPLREG	./templreg

Notes:

1. Output files default to the z/OS UNIX directory if the source resides in the z/OS UNIX file system, or to an MVS data set if the source resides in a data set.
2. If you have specified the OE option, see “OE | NOOE” on page 192 for a description of the default naming convention.
3. If you supply inline source in your JCL, the compiler will not generate an output file name automatically. You can specify a file name either as a suboption for a compiler option, or on a ddname in your JCL.
4. If you are using **#pragma options** to specify a compile-time option that generates an output file, you must use a ddname to specify the output file name when compiling under batch. The compiler will not automatically generate file names for output that is created by **#pragma options**.

Example: Under TSO, the compiler generates the object file `userid.TEST.SRC.OBJ` if you compile the following:

```
cc TEST.SRC (OBJ
```

The compiler generates the object file `userid.TEST.SRC.OBJ(HELLO)` if you compile the following:

```
cc 'hlqual.TEST.SRC(HELLO)' (OBJ
```

Listing output

Note: Although the compiler listing is for your use, it is not a programming interface and is subject to change.

To create a listing file that contains source, object, or inline reports use the SOURCE, LIST, or INLRPT compile options, respectively. The listing includes the results of the default or specified options of the CPARM parameter (that is, the diagnostic messages and the object code listing). If you specify *filename* with two or more of these compile options, the compiler combines the listings and writes them to the last file specified in the compile options. If you did not specify *filename*, the listing will go to the SYSCPRT DD name, if you allocated it. Otherwise, the compiler generates a default file name as described in "LIST | NOLIST" on page 163.

Object module output

To create an object module and store it on disk or tape, you can use the OBJECT compiler option.

If you do not specify *filename* with the OBJECT option, the compiler stores the object code in the file that you define in the SYSLIN DD statement. If you do not specify *filename* with the OBJECT option, and did not allocate SYSLIN, the compiler generates a default file name, as described in "OBJECT | NOOBJECT" on page 188.

Under z/OS UNIX System Services, an object name specified with **-o** will take priority over the file name specified with the OBJECT option.

Differences in object modules under IPA: The format of the object module generated by a regular compile may differ from that generated by an IPA Compile, depending on the IPA suboption setting. If it is set to OBJONLY, then they have the same format. If it is set to OBJECT, then the object module will contain an IPA object in addition to the traditional object code and associated symbolic information. If NOOBJECT is specified, then only the IPA object is written to the object module. The IPA Link phase reads the IPA object code from the input object modules. It is the only process which can do so. Therefore, if you attempt to bind an IPA object file that was created by using the IPA(NOLINK,NOOBJECT) option, then the binder issues an error message and the bind will fail.

Refer to "Valid input/output file types" on page 339 for information about valid input/output file types.

Preprocessor output

If you specify *filename* with the PPONLY compile option, the compiler writes the preprocessor output to that file. If you do not specify *filename* with the PPONLY option, the compiler stores the preprocessor output in the file that you define in the SYSUT10 DD statement. If you did not allocate SYSUT10, the compiler generates a default file name, as described in "PPONLY | NOPPONLY" on page 203.

Template instantiation output

If you specify *location*, which is either a z/OS UNIX file or a sequential file (or PDS member), with the TEMPLATEREGISTRY compile option, the compiler writes the template registry to that location. If you do not specify *location* with the

TEMPLATEREGISTRY option, the compiler determines a default destination for the template registry file. See “TEMPLATEREGISTRY | NOTEMPLATEREGISTRY (C++ only)” on page 247 for more information on this default.

If you specify *location*, which is either a z/OS UNIX directory or a PDS, with the TEMPINC compile option, the compiler writes the template instantiation output to that location. If you do not specify *location* with the TEMPINC option, the compiler stores the TEMPINC output in the file that is associated with the TEMPINC DD name. If you did not allocate DD:TEMPINC, the compiler determines a default destination for the template instantiation files. See “TEMPINC | NOTEMPINC (C++ only)” on page 244 for more information on this default.

Valid input/output file types

Depending on the type of file that is used as primary input, certain output file types are allowed. The following table describes these combinations of input and output files:

Table 37. Valid combinations of source and output file types

Input Source File	Output Data Set Specified Without (member) Name, for example `A.B.C`	Output Data Set Specified as filename(member), for example `A.B.C(D)`	Output Specified as a z/OS UNIX file, for example `a/b/c.o`	Output Specified as a z/OS UNIX directory, for example `a/b`
Sequential Data Set, for example `A.B`	1. If the file exists as a sequential data set, overwrites it 2. If the file does not exist, creates sequential data set 3. Otherwise compilation fails	1. If the PDS does not exist, creates PDS and member 2. If the PDS exists and member does not exist, adds member 3. If the PDS and member both exist, then overwrites the member	1. If the directory does not exist, compilation fails 2. If the directory exists but the file does not exist, creates file 3. If the file exists, overwrites the file	Not supported
A member of a PDS using (member), for example `A.B(C)`	1. If the file exists as a sequential data set, overwrites it 2. If the file exists as a PDS, creates or overwrites member 3. If the file does not exist, creates PDS and member	1. If the PDS does not exist, creates PDS and member 2. If the PDS exists and member does not exist, adds member 3. If the PDS and member both exist, then overwrites the member	1. If the directory does not exist, compilation fails 2. If the directory exists and the file with the specified file name does not exist, creates file 3. If the directory exists and the file exists, overwrites file	1. If the directory does not exist, compilation fails 2. If the directory exists and the file with the file name *MEMBER.ext* does not exist, creates file 3. If the directory exists and the file with the file name *MEMBER.ext* also exists, overwrite file

Table 37. Valid combinations of source and output file types (continued)

Input Source File	Output Data Set Specified Without (member) Name, for example `A.B.C`	Output Data Set Specified as filename(member), for example `A.B.C(D)`	Output Specified as a z/OS UNIX file, for example `a/b/c.o`	Output Specified as a z/OS UNIX directory, for example `a/b`
All members of a PDS, for example `A.B`	1. If the file exists as a PDS, creates or overwrites members 2. If the file does not exist, creates PDS and members 3. Otherwise compilation fails	Not Supported	Not Supported	1. If the directory does not exist, compilation fails 2. If the directory exists and the files with the file names *`MEMBER.ext`* do not exist, creates files 3. If the directory exists and the files with the file names *`MEMBER.ext`* exist, overwrites files
z/OS UNIX file, for example `/a/b/d.c`	1. If the file exists as a sequential data set, overwrites file 2. If the file does not exist, creates sequential data set 3. Otherwise compilation fails	1. If the PDS does not exist, creates the PDS and stores a member into the data set 2. If the PDS exists and member does not exist, then adds the member in the PDS 3. If the PDS and member both exist, then overwrites the member	1. If the directory does not exist, compilation fails 2. If the directory exists but the file does not exist, creates file 3. If the file exists, overwrites the file	1. If the directory does not exist, compilation fails 2. If the directory exists and the file does not exist, creates file 3. If the directory exists and the file exists, overwrites file
z/OS UNIX directory, for example `a/b/`	Not supported	Not supported	Not supported	1. If the directory does not exist, compilation fails 2. If the directory exists and the files to be written do not exist, creates files 3. If the directory exists and the files to be written already exist, overwrites files

Compiling under z/OS batch

To compile your C/C++ source program under batch, you can either use cataloged procedures that IBM supplies, or write your own JCL statements.

Using cataloged procedures for z/OS XL C

You can use one of the following IBM-supplied cataloged procedures. Each procedure includes a compilation step to compile your program.

EDCC Compile a 31-bit or 64-bit program

EDCCB
Compile and bind a 31-bit program

EDCXCB
Compile and bind a 31-bit XPLINK C program

EDCQCB
Compile and bind a 64-bit C program

EDCCL
Compile and link-edit a 31-bit naturally re-entrant program

EDCCBG
Compile, bind, and run a 31-bit program

EDCXCBG
Compile, bind, and run a 31-bit XPLINK C program

EDCQCBG
Compile, bind, and run a 64-bit C program

EDCCLG
Compile, link-edit, and run a 31-bit program

EDCCPLG
Compile, prelink, link-edit, and run a 31-bit program

EDCCLIB
Compile and maintain an object library for a 31-bit or 64-bit application

IPA considerations

The EDCC procedure should be used for the IPA Compile step. Only the EDCI and EDCXI procedures apply to the IPA Link step. For information on the EDCI and EDCXI procedures, see Chapter 8, “Using the IPA Link step with z/OS XL C/C++ programs,” on page 375.

To run the IPA Compile step, use the EDCC procedure, and ensure that you specify the IPA(NOLINK) or IPA compiler option. Note that you must also specify the LONGNAME compiler option or the **#pragma longname** directive.

To create an IPA-optimized object module, you must run the IPA Compile step for each source file in your program, and the IPA Link step once for the entire program. Once you have successfully created an IPA-optimized object module, you must bind it to create the final executable.

For further information on IPA, see Chapter 8, “Using the IPA Link step with z/OS XL C/C++ programs,” on page 375.

Using cataloged procedures for z/OS XL C++

You can use one of the following cataloged procedures that IBM supplies. Each procedure includes a compilation step to compile your program.

CBCC	Compile a 31-bit or 64-bit program
CBCCB	Compile and bind a 31-bit non-XPLINK program
CBCXCB	Compile and bind a 31-bit XPLINK C++ program
CBCQCB	Compile and bind a 64-bit C++ program

CBCCL	Compile, prelink, and link for a 31-bit non-XPLINK program
CBCCBG	Compile, bind, and run a 31-bit non-XPLINK program
CBCXCBG	Compile, bind, and run a 31-bit XPLINK C++ program
CBCQCBG	Compile, bind, and run a 64-bit C++ program
CBCCLG	Compile, prelink, link, and run a 31-bit non-XPLINK program

See Chapter 12, “Cataloged procedures and REXX EXECs,” on page 465 for more information on cataloged procedures.

IPA considerations

The CBCC procedure should be used for the IPA Compile step. Only the CBCI and CBCXI procedures apply to the IPA Link step. For information on the CBCI and CBCXI procedures, see Chapter 8, “Using the IPA Link step with z/OS XL C/C++ programs,” on page 375.

To run the IPA Compile step, use the CBCC procedure, and ensure that you specify the IPA(NOLINK) or IPA compiler option. Note that for C you must also specify the LONGNAME compiler option or the **#pragma longname** directive. For C++, you don’t have to do this since C++ always uses LONGNAME. You should not specify the NOLONGNAME option.

To create an IPA-optimized object module, you must run the IPA Compile step for each source file in your program, and the IPA Link step once for the entire program. Once you have successfully created an IPA-optimized object module, you must bind it to create the final executable.

For further information on IPA, see Chapter 8, “Using the IPA Link step with z/OS XL C/C++ programs,” on page 375.

Using special characters

When invoking the compiler directly, if a string contains a single quotation mark (') it should be written as two single quotation marks ('') as in:

```
//COMPILE EXEC PGM=CCNDRVR,PARM='OPTFILE(''USERID.OPTS'')'
```

If you are using the same string to pass a parameter to a cataloged procedure, use four single quotation marks (''''), as follows:

```
//COMPILE EXEC CBCC,CPARM='OPTFILE(''''USERID.OPTS'''')'
```

A backslash need not precede special characters in z/OS UNIX System Services file names that you use in DD cards. For example:

```
//SYSLIN DD PATH='/u/user1/obj 1.o'
```

A backslash must precede special characters in z/OS UNIX file names that you use in the PARM statement. For example:

```
//STEP1 EXEC PGM=CCNDRVR,PARM='/u/user1/obj\ 1.o'
```

Examples of compiling programs using your own JCL

The following example shows sample JCL for compiling a 32-bit C program:

```
//jobname    JOB   acctno,name...
//COMPILE    EXEC  PGM=CCNDRVR,
// PARM='/SEARCH(''CEE.SCEEH.+'') NOOPT SO OBJ'
//STEPLIB    DD    DSNAME=CEE.SCEERUN,DISP=SHR
//           DD    DSNAME=CEE.SCEERUN2,DISP=SHR
//           DD    DSNAME=CBC.SCCNCMP,DISP=SHR
//SYSLIN     DD    DSNAME=MYID.MYPROG.OBJ(MEMBER),DISP=SHR
//SYSPRINT   DD    SYSOUT=*
//SYSIN      DD    DATA,DLM=@@
#include <stdio.h>
   ⋮
int main(void)
{
/*  comment   */
   ⋮
}
@@
//SYSUT1     DD    DSN=...
   ⋮
//*
```

Figure 14. JCL for compiling a 32-bit C program (for NOOPT, SOURCE, and OBJ)

The following example shows sample JCL for compiling a 64-bit C program:

```
//jobname JOB acctno,name...
//COMPILE EXEC PGM=CCNDRVR,
// PARM='/SEARCH(''CEE.SCEEH.+'') NOOPT SO LP64 OPTFILE(DD:CPATH)'
//STEPLIB DD DSNAME=CEE.SCEERUN,DISP=SHR
//        DD DSNAME=CEE.SCEERUN2,DISP=SHR
//        DD DSNAME=CBC.SCCNCMP,DISP=SHR
//SYSLIN DD DSNAME=MYID.MYPROG.OBJ(MEMBER),DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSIN DD DATA,DLM=@@
#include <stdio.h> ...
int main(void)
{
/* comment */
   ⋮
}
@@
//SYSUT1 DD DSN=...
   ⋮
//*
```

Figure 15. JCL for compiling a 64-bit C program (for NOOPT, SOURCE, and LP64)

The following example shows sample JCL for compiling a 32-bit C++ program:

```
//jobname    JOB   acctno,name...
//COMPILE EXEC PGM=CCNDRVR,
// PARM='/CXX SEARCH(''CEE.SCEEH.+'',''CBC.SCLBH.+''),NOOPT,SO,OBJ'
//STEPLIB  DD  DSN=CEE.SCEERUN,DISP=SHR
//         DD  DSN=CEE.SCEERUN2,DISP=SHR
//         DD  DSN=CBC.SCCNCMP,DISP=SHR
//SYSLIN   DD  DSN=MYID.MYPROJ.OBJ,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSIN    DD  DATA,DLM=@@
#include <stdio.h>
#include <iostream.h>
   ⋮
int main(void)
{
//  comment
   ⋮
}
@@
//SYSUT1   DD    DSN=...
   ⋮
//*
```

Figure 16. JCL for compiling a 32-bit C++ program (for NOOPT, SOURCE, and OBJ)

The following example shows sample JCL for compiling a 64-bit C++ program:

```
//jobname JOB acctno,name...
//COMPILE EXEC PGM=CCNDRVR,
// PARM='/CXX SEARCH(''CEE.SCEEH.+'',''CBC.SCLBH.+''),NOOPT,SO,LP64'
//STEPLIB DD DSN=CEE.SCEERUN,DISP=SHR
//        DD DSN=CEE.SCEERUN2,DISP=SHR
//        DD DSN=CBC.SCCNCMP,DISP=SHR
//SYSLIN DD DSN=MYID.MYPROJ.OBJ,DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSIN DD DATA,DLM=@@
#include <stdio.>
#include <iostream.h> ...
int main(void)
{
// comment

   ⋮
}
@@
//SYSUT1 DD DSN=...

   ⋮
//*
```

Figure 17. JCL for compiling a 64-bit C++ program (for NOOPT, SOURCE, and LP64)

Specifying source files

For non-z/OS UNIX System Services files, use this format of the SYSIN DD statement:

```
//SYSIN DD DSNAME=dsname,DISP=SHR
```

If you specify a PDS without a member name, all members of that PDS are compiled.

Note: If you specify a PDS as your primary input, you must specify either a PDS or a z/OS UNIX directory for your output files.

For z/OS UNIX files, use this format of the SYSIN DD statement:

```
//SYSIN DD PATH='pathname'
```

You can specify compilation for a single file or all source files in a z/OS UNIX directory, for example:

```
//SYSIN DD PATH='/u/david'
//* All files in the directory /u/david are compiled
```

Note: If you specify a z/OS UNIX directory as your primary input, you must specify a z/OS UNIX directory for your output files.

When you place your source code directly in the input stream, use the following form of the SYSIN DD statement:

```
//SYSIN DD DATA,DLM=
```

rather than:

```
//SYSIN DD *
```

When you use the `DD *` convention, the first XL C/C++ comment statement that starts in column 1 will terminate the input to the compiler. This is because /*, the beginning of a C or C++ comment, is also the default delimiter.

Note: To treat columns 73 through 80 as sequence numbers, use the SEQUENCE compiler option.

For more information about the DD * convention, refer to the publications that are listed in *z/OS Information Roadmap*.

Specifying include files

Example: Use the SEARCH option to specify system include files, and the LSEARCH option to specify your include files:

```
//C EXEC PGM=CCNDRVR,PARM='/CXX SEARCH(''CEE.SCEEH.+'',''CBC.SCLBH.+'')'
```

You can also use the SYSLIB and USERLIB DD statements (note that the SYSLIB DD statement has a different use if you are running the IPA Link step). To specify more than one library, concatenate multiple DD statements as follows:

```
//SYSLIB     DD     DSNAME=USERLIB,DISP=SHR
//           DD     DSNAME=DUPX,DISP=SHR
```

Note: If the concatenated data sets have different block sizes, either specify the data set with the largest block size first, or use the `DCB=`*dsname* subparameter on the first DD statement. For example:

```
//USERLIB DD DSNAME=TINYLIB,DISP=SHR,DCB=BIGLIB
//        DD DSNAME=BIGLIB,DISP=SHR
```

where `BIGLIB` has the largest block size. For rules regarding concatenation of data sets in JCL, refer to *z/OS XL C/C++ Programming Guide*.

Specifying output files

You can specify output file names as suboptions to the compiler. You can direct the output to a PDS member as follows:

```
//    CPARM='LIST(MY.LISTINGS(MEMBER1))'
```

You can direct the output to a z/OS UNIX System Services file as follows:

```
//    CPARM='LIST(./listings/member1.lst)'
```

You can also use DD statements to specify output file names.

To specify non-z/OS UNIX files, use DD statements with the DSNAME parameter. For example:

```
//SYSLIN DD DSN=USERID.TEST.OBJ(HELLO),DISP=SHR
```

To specify z/OS UNIX directories or z/OS UNIX files, use DD statements with the PATH parameter.

```
//SYSLIN DD PATH='/u/david/test.o',PATHOPTS=(OWRONLY,OCREAT,OTRUNC)
```

Note: Use the PATH and PATHOPTs parameters when specifying z/OS UNIX files in the DD statements. For additional information on these parameters, refer to the list of publications in *z/OS Information Roadmap*.

If you do not specify the output *filename* as a suboption, and do not allocate the associated ddname, the compiler generates a default output file name. There are two situations when the compiler will not generate a default file name:
- You supply instream source in your JCL.
- You are using **#pragma options** to specify a compile-time option that generates an output file.

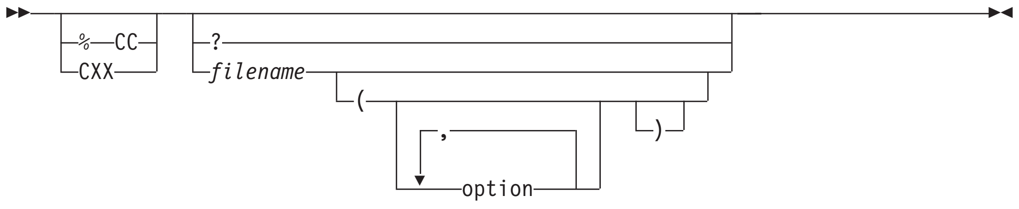
Compiling under TSO

You can invoke the z/OS XL C/C++ compiler under TSO by foreground execution from TSO READY. This method of foreground execution calls the CC or CXX REXX EXECs supplied by IBM.

Note: To run the compiler under TSO, you must have access to the run-time libraries. To ensure that you have access to the run-time library and compiler, do one of the following:
- Have your system programmer add the libraries to the LPALST or LPA
- Have your system programmer add the libraries to the LNKLST
- Have your system programmer change the LOGON PROC so the libraries are added to the STEPLIB for the TSO session
- Have your system programmer customize the REXX EXEC CCNCCUST, which is called by the CC, CXX, and other EXECs to set up the environment

Using the CC and CXX REXX EXECs

You can use the CC REXX EXEC to invoke the z/OS XL C compiler, and the CXX REXX EXEC to invoke the z/OS XL C++ compiler. These REXX EXECs share the same syntax:



where

% invokes the REXX EXEC CC

option is any valid compiler option

filename can be one of the following:
- A sequential data set
- A member of a partitioned data set
- All members of a partitioned data set
- A z/OS UNIX System Services file
- All files in a z/OS UNIX directory

If *filename* is not immediately recognizable as a z/OS UNIX file or data set, it is assumed to be a data set. Prefix the file name with `//` to identify it as a data set, and with `./` or `/` to identify it as a z/OS UNIX file. For more information on file naming considerations refer to *z/OS XL C/C++ Programming Guide*.

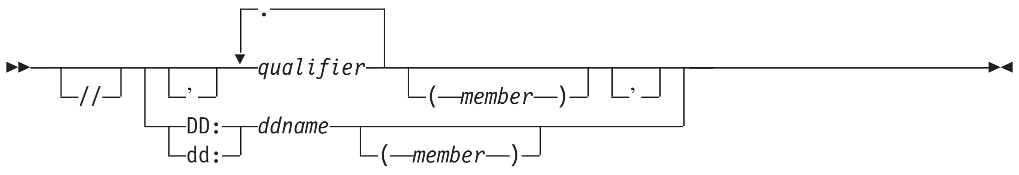
If you invoke either CC or CXX with no arguments or with only a single question mark, the appropriate preceding syntax diagram is displayed.

If you are using **#pragma options** to specify a compile-time option that generates an output file, you must use a ddname to specify the output file name. The compiler will not automatically generate file names for output that is created by **#pragma options**.

Unless CCNCCUST has been customized, the default SYSLIB for CC is CEE.SCEEH.H, and CEE.SCEEH.SYS.H concatenated. If you want to override the default SYSLIB that is allocated by the CC exec, you must allocate the ddname SYSLIB before you invoke CC. If you did not allocate the ddname SYSLIB before you invoked CC EXEC, the CC EXEC allocates the default SYSLIB.

Specifying sequential and partitioned data sets

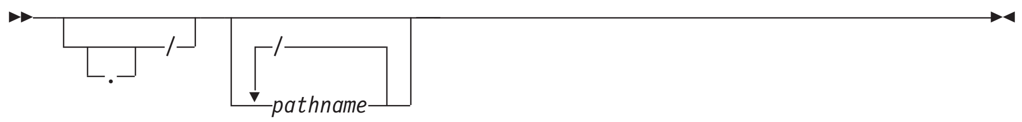
To specify a sequential or partitioned data set for your source file use the following syntax:



Note: If you use the leading single quotation mark to indicating a fully qualified data set name, you must also use the trailing single quotation mark.

Specifying z/OS UNIX files or directories

You can use the CC or CXX REXX EXECs to compile source code that is stored in z/OS UNIX files and directories. Use the following syntax when specifying a z/OS UNIX file or directory as your input or output file:



If you specify a z/OS UNIX directory, all the source files in that directory are compiled. In the following example all the files in `/u/david/src` are compiled:

```
CC /u/david/src
```

When the file name contains the special characters double quotation mark, blank, or backslash, you must precede these characters with a backslash, as follows:

```
CC  /u/david/db\ 1.c
CC  file\"one
```

When you use the CC or CXX REXX EXEC, you must use unambiguous z/OS UNIX source file names. For example, the following input files are z/OS UNIX files:

```
CXX ./test/hello.c
CC  /u/david/test/hello.c
CXX test/hello.c
CC  ///hello.c
CC  ../test/hello.c
```

If you specify a file name that does not include pathnames with single slashes, the compiler treats the file as a non-z/OS UNIX file. The compiler treats the following input files as non-z/OS UNIX files:

```
CXX hello.c
CC  //hello.c
```

Using special characters

When z/OS UNIX file names contain the special characters blank, backslash, and double quotation mark, you must precede the special character with a backslash(\).

When suboptions contain the special characters left bracket `(`, right bracket `)`, comma, backslash, blank and double quotation mark, you must precede these characters with a double backslash(\\) so that they are interpreted correctly, as in:

```
    def(errno=\\(*__errno\\(\\)\\))
```

Note: Under TSO, you must precede special characters by a backslash \ in both file names and options.

Specifying compiler options under TSO

When you use REXX EXECs supplied by IBM, you can override the default compiler options by specifying the options directly on the invocation line after an open left parenthesis `(`.

Example: The following example specifies, multiple compiler options with the sequential file `STUDENT.GRADES.CXX`:

```
CXX 'STUDENT.GRADES.CXX'
      ( LIST,TEST,
       LSEARCH(MASTER.STUDENT,COURSE.TEACHER),
       SEARCH(VGM9.FINANCE,SYSABC.REPORTS),
       OBJ('GRADUATE.GRADES.OBJ(REPORT)')
```

See "Summary of compiler options" on page 49 for more information on compiler options.

Compiling and binding in the z/OS UNIX System Services environment

z/OS UNIX System Services C/C++ programs with source code in z/OS UNIX files or data sets must be compiled to create output object files residing either in z/OS UNIX files or data sets.

Both the SCEERUN and the SCEERUN2 libraries must be available when compiling in the z/OS UNIX System Services environment.

You can compile and bind application source code at one time, or compile the source and then bind at another time with other application source files or compiled objects.

As of z/OS V1R6, there are two utilities that enable you to invoke the compiler. The c89 utility enables compiler invocation using host environment variables and the xlc utility uses an external configuration file to control the invocation of the compiler. The following list highlights the differences between the xlc and c89 utilities:

- xlc utility uses the c89 utility to invoke the binder and the assembler and it has no direct interface to them
- xlc does not require that **lp64** and **xplink** be explicitly specified as options on the command line for both the compile and the bind step; it uses the _64 and _x command name suffixes to ensure 64-bit and XPLINK compiles and binds
- xlc utility supports **-q** options syntax as the primary method of specifying options on the command line
- xlc utility is unaffected by the value assigned to the STEPLIB environment variable in the z/OS UNIX Systems Services session; it obtains the STEPLIB from the configuration file
- xlc utility supports the same command names as the c89 utility (**cc**, **c89**, **c++**, and **cxx**), so the PATH environment variable must contain the path to the **xlc** `bin` directory ahead of the `/bin` directory if the **xlc** version of **cc**, **c89**, **c++**, and **cxx** is desired
- xlc utility does not support -WI for invoking IPA; it uses -O4 and -O5 or **-qipa** as the mechanism for invoking IPA

Note: For more information on the xlc utility, see Chapter 24, "xlc — Compiler invocation using a customizable configuration file," on page 577.

The c89 utility and xlc utility invoke the binder by default, unless the output file of the link-editing phase (**-o** option) is a PDS, in which case the prelinker is used.

For information on customizing your environment to compile and bind in the z/OS UNIX System Services environment, see "Environment variables" on page 550 or "Setting up a configuration file" on page 581.

Use the c89 utility or the xlc utility to compile and bind a C application program from the z/OS shell. The syntax is:

```
c89 [-options ...] [file.c ...] [file.a ...] [file.o ...] [-l libname]
```

where:

options	are **c89** or **xlc** options.
file.c	is a source file. Note that C source files have a file extension of lowercase c.
file.o	is an object file.
file.a	is an archive file.
libname	is an archive library.

The c89 and xlc utilities support IPA. For information on how to invoke the IPA Compile step using c89 or xlc, refer to "Invoking IPA using the c89 or xlc utilities" on page 354.

You can also use the **cc** command to compile a C application program from the z/OS shell. For more information, see Chapter 21, "c89 — Compiler invocation using host environment variables," on page 535 or the **xlc** command names described in Chapter 24, "xlc — Compiler invocation using a customizable configuration file," on page 577.

Use the **c++** command to compile and bind a C++ application program from the z/OS shell. The syntax for **c++** is:

```
c++ [-options ...] [file.C ...] [file.a ...] [file.o ...] [-l libname]
```

where:

options	are C++ options.
file.C	is a source file. Note that C++ files have a file extension of uppercase C. The _CXX_CXXSUFFIX environment variable or `cxxsuffix` configuration file attribute can also be used to control which extensions are recognized as C++ file source extensions.
file.o	is an object file.
file.a	is an archive file.
libname	is an archive library.

Another name for the **c++** command is **cxx**. The **cxx** command and the **c++** command are identical. You can use **cxx** instead of **c++** in all the examples that are shown in this topic. If you are using the xlc utility, you can also use the **xlc** and the **xlc++** commands, which are identical to **c++** and **cxx**.

For a complete list of **c++** options, and for more information on **cxx**, see Chapter 21, "c89 — Compiler invocation using host environment variables," on page 535 and Chapter 24, "xlc — Compiler invocation using a customizable configuration file," on page 577.

Note: You can compile and bind application program source and objects from within the shell using the c89 or xlc utilities. If you use one of these utilities, you must keep track of and maintain all the source and object files for the application program. You can use the make utility to maintain your z/OS UNIX System Services application source files and object files automatically when you update individual modules. The make utility will only compile files that have changed since the last make run.

For more information on using the make utility, see Chapter 18, “Archive and make utilities,” on page 525 and *z/OS UNIX System Services Programming Tools*.

Compiling without binding using compiler invocation command names supported by c89 and xlc

To compile source files without binding them, enter one of the supported command names (for example, **c89** or **c++**) with the **-c** option to create object file output. Use the **-o** option to specify placement of the application program executable file to be generated. The placement of the intermediate object file output depends on the location of the source file:

- If the z/OS XL C/C++ source module is a z/OS UNIX System Services file, the object file is created in the working directory.
- If the z/OS XL C/C++ source module is a data set, the object file is created as a data set. The object file is placed in a data set with the qualified name of the source and identified as an object.

  For example, if the z/OS XL C/C++ source is in the sequential data set `LANE.APPROG.USERSRC.C`, the object is placed in the data set `LANE.APPROG.USERSRC.OBJ`. If the source is in the partitioned data set (PDS) member `OLSEN.IPROGS.C(FILSER)`, the object is placed in the PDS member `OLSEN.IPROGS.OBJ(FILSER)`.

  Note: When the z/OS XL C/C++ source is located in a PDS member, you should specify double quotation marks around the qualified data set name. For example:

  ```
  c89 -c "//'OLSEN.IPROGS.C(FILSER)'"
  ```

  If the file name is not bracketed by quotation marks, the parentheses around the member name in the fully qualified PDS name would be subject to special shell parsing rules.

  Since the data set name is always converted to uppercase, you can specify it in lowercase or mixed case.

Compiling z/OS XL C application source to produce only object files

c89 and xlc recognize that a file is a C source file by the `.c` suffix for z/OS UNIX System Services files, and the `.C` low-level qualifier for data sets. They recognize that a file is an object file by the `.o` suffix for z/OS UNIX files, and the `.OBJ` low-level qualifier for data sets.

To compile z/OS XL C source to create the default 32-bit object file `usersource.o` in your working z/OS UNIX directory, specify:

```
c89 -c usersource.c
```

To compile z/OS XL C source to create the default 64-bit object file `usersource.o` in your working z/OS UNIX directory, specify the following using the c89 utility:

```
c89 -c -Wc,lp64 usersource.c
```

The following shows the same example using the xlc utility:

```
c89_64 -c usersource.c
```

To compile z/OS XL C source to create an object file as a member in the PDS `KENT.APPROG.OBJ`, specify:

```
c89 -c "//'kent.approg.c(usersrc)'"
```

Compiling z/OS XL C++ application source to produce only object files

c89 and xlc recognize that a file is a C++ source file by the `.C` suffix for z/OS UNIX System Services files, and the .CXX low-level qualifier for data sets. They recognize that a file is an object file by the `.o` suffix for z/OS UNIX files, and the .OBJ low-level qualifier for data sets.

To compile z/OS XL C++ source to create the default 32-bit object file `usersource.o` in your working z/OS UNIX directory, specify the following:

```
c++ -c usersource.C
```

To compile z/OS XL C++ source to create the default 64-bit object file `usersource.o` in your working z/OS UNIX directory, using the c89 utility specify:

```
c++ -c -Wc,lp64 usersource.C
```

The following shows the same example using the xlc utility:

```
c++_64 usersource.C
```

To compile z/OS XL C++ source to create an object file as a member in the PDS `JONATHAN.APPROG.OBJ`, specify:

```
 c++ -c "//'jonathan.approg.CXX(usersrc)'"
```

Note:

To use the TSO utility OGET to copy a C++ z/OS UNIX listing file to a VBA data set, you must add a blank to any null records in the listing file. Use the **awk** command as follows if you are using the c89 utility:

```
c++ -cV mypgm.C | awk '/^[^$]/ {print} /^$/
   {printf "%s \n", $0}' > mypgm.lst
```

The following shows the same example using the xlc utility:

```
xlC -c -qsource mypgm.C | awk '/^[^$]/ {print} /^$/
   {printf "%s \n", $0}' > mypgm.lst
```

Compiling and binding application source to produce an application executable file

To compile an application source file to create the 32-bit object file `usersource.o` in the z/OS UNIX System Services working directory and the executable file `mymod.out` in the `/app/bin` directory, specify:

```
c89 -o /app/bin/mymod.out usersource.c
```

To compile an application source file, to create the 64-bit object file `usersource.o` in the z/OS UNIX working directory and the executable file `mymod.out` in the `/app/bin` directory, specify the following using the c89 utility

```
 c89 -Wc,lp64 -Wl,lp64 -o /app/bin/mymod.out usersource.c
```

The following shows the same example using the xlc utility:

```
c89_64 -o /app/bin/mymod.out usersource.c
```

To compile the z/OS XL C source member `MAINBAL` in the PDS `CLAUDIO.PGMS.C`, and bind it to produce the application executable file `/u/claudio/myappls/bin/mainbal.out`, specify:

```
c89 -o /u/claudio/myappls/bin/mainbal.out "//'claudio.pgms.C(MAINBAL)'"
```

Compiling and binding in one step using compiler invocation command names supported by c89 and xlc

To compile and bind a XL C/C++ application program in one step to produce an executable file, specify **c89** or **c++** *without* specifying the **-c** option. You can use the **-o** option with the command to specify the name and location of the application program executable file that will be created. The **c++** and **cxx** commands are identical. You can use **cxx** instead of **c++** in all the examples that are shown in this topic. If you are using the xlc utility, you can also use the **xlc** and **xlc++** commands, which are identical to **c++** and **cxx**.

The c89 utility and xlc utility invoke the binder by default, unless the output file of the link-editing phase (**-o** option) is a PDS, in which case the prelinker is used.

- To compile and bind an application source file to create the 32-bit default executable file `a.out` in the z/OS UNIX System Services working directory, specify:

  ```
  c89 usersource.c
  c++ usersource.C
  ```

- To compile and bind an application source file to create the 64-bit default executable file `a.out` in the z/OS UNIX working directory, specify:

  ```
  c89 -Wc,lp64 -Wl,lp64 usersource.c
  c++ -Wc,lp64 -Wl,lp64 usersource.C
  xlC_64 usersource.C
  ```

- To compile and bind an application source file to create the `mymod.out` executable file in your `/app/bin` directory, specify:

  ```
  c89 -o /app/bin/mymod.out usersource.c
  c++ -o /app/bin/mymod.out usersource.C
  ```

- To compile and bind several application source files to create the `mymod.out` executable file in your `/app/bin` directory, specify:

  ```
  c89 -o /app/bin/mymod.out usrsrc.c otsrc.c "//'MUSR.C(PWAPP)'"
  c++ -o /app/bin/mymod.out usrsrc.C otsrc.C "//'MUSR.C(PWAPP)'"
  ```

- To compile and bind an application source file to create the `MYLOADMD` member of your `APPROG.LIB` PDS, specify:

  ```
  c89 -o "//'APPROG.LIB(MYLOADMD)'" usersource.c
  c++ -o "//'APPROG.LIB(MYLOADMD)'" usersource.C
  ```

- To compile and bind an application source file with several previously compiled object files to create the executable file `zinfo` in your `/prg/lib` z/OS UNIX directory, specify:

  ```
  c89 -o /prg/lib/zinfo usrsrc.c xstobj.o "//'MUSR.OBJ(PWAPP)'"
  c++ -o /prg/lib/zinfo usrsrc.C xstobj.o "//'MUSR.OBJ(PWAPP)'"
  ```

- To compile and bind an application source file and capture the listings from the compile and bind steps into another file, specify:

  ```
  c89 -V barryl.c > barryl.lst
  c++ -V barryl.C > barryl.lst
  ```

Note: -V does not cause all listings to be emitted when you invoke the compiler using xlc. Use, for example, **-qsource** or **-qlist** instead.

Building an application with XPLINK using the c89 or xlc utilities

To build an application with XPLINK using the c89 utility you must specify the XPLINK compiler option (i.e., **–Wc,xplink**) and the XPLINK binder option (i.e., **–Wl,xplink**). The binder option is not actually passed to the binder. It is used by c89 to set up the appropriate link data sets.

To build an application with XPLINK using the xlc utility, you do not have to explicitly specify the **xplink** option on the command line for either the compile or the bind step. xlc uses the `_x` command name suffix to ensure XPLINK compiles and binds.

Building a 64-bit application using the c89 or xlc utilities

To build a 64-bit application using the c89 utility, you must use the LP64 compiler option (i.e., **-Wc,lp64**) and the LP64 binder option (i.e., **-Wl,lp64**). The binder option is not actually passed to the binder. It is used by c89 to set up the appropriate link data sets.

To build a 64-bit application using the **xlc** utility, you do not have to explicitly specify the **lp64** option on the command line for either the compile or the bind step. xlc uses the `_64` command name suffix to ensure 64-bit compiles and binds.

Invoking IPA using the c89 or xlc utilities

You can invoke the IPA Compile step, the IPA Link step, or both using the c89 or xlc utilities. The step that you invoke depends upon the invocation parameters and type of files specified. To invoke IPA using **c89**, you must specify the `I` phase indicator along with the W option of the **c89** utility. You can specify IPA suboptions as comma-separated keywords. To invoke IPA using **xlc**, you must use the **-qipa**, -O4, or -O5 options. You can specify IPA suboptions as colon-separated keywords.

If you invoke the c89 utility or xlc utility by specifying the **-c** compiler option and at least one source file, **c89** or **xlc** automatically specifies IPA(NOLINK) and automatically invokes the IPA Compile step. For example, the following **c89** command invokes the IPA Compile step for the source file `hello.c`:

```
c89 -c -WI,noobject hello.c
```

The following **xlc** command invokes the IPA Compile step for the source file `hello.c`:

```
xlc -c -qipa=noobject hello.c
```

If you invoke c89 or xlc with at least one source file for compilation and any number of object files, and do not specify the **-c** option, c89 or xlc invokes the IPA Compile step once for each compilation unit. It then invokes the IPA Link step once for the entire program, and then invokes the binder.

Example: The following **c89** command invokes the IPA Compile step, the IPA Link step, and the bind step while creating program `foo`:

```
c89 -o foo -WI,object foo.c
```

The following shows the same example using the xlc utility:

```
xlc -o foo -qipa=object foo.c
```

Refer to Chapter 21, “c89 — Compiler invocation using host environment variables,” on page 535 for more information about the c89 utility or Chapter 24, “xlc — Compiler invocation using a customizable configuration file,” on page 577 for more information about the xlc utility.

Specifying options for the IPA Compile step

You can pass options to the IPA Compile step, as follows:

- You can pass IPA compiler option suboptions by specifying **-WI,** for c89 or **-qipa=** for xlc, followed by the suboptions.
- You can pass compiler options by specifying **-Wc**, for c89 or **-q** for xlc, followed by the options.

Using IPA(OBJONLY) with the c89 or xlc utilities

A compilation using IPA(OBJONLY) is simply a standard non-IPA compilation with this option added. Do not use the **-WI** flag with c89 or **-qipa** with xlc, as this would convert the compilation into an IPA Compile step.

Example: The following c89 command results in an OPT(2) IPA(OBJONLY) compilation for the source file `hello.c`:

```
c89 -c -Wc,ipa\(objonly\) -2 hello.c
```

The following shows the same example using the xlc utility:

```
xlc -c -Wc,ipa\(objonly\) -O2 hello.c
```

Using the make utility

You can use the make utility to control the build of your z/OS UNIX System Services XL C/C++ applications. The make utility calls the c89 utility by default to compile and bind the programs that the previously created makefile specifies.

Example: To create `myappl` you compile and bind two source parts `mymain.c` and `mysub.c`. This dependency is captured in makefile `/u/jake/myappl/Makefile`. No recipe is specified, so the default makefile rules are used. If `myappl` was built and a subsequent change was made only to `mysub.c`, you would specify:

```
cd /u/jake/myappl
make
```

The make utility sees that `mysub.c` has changed, and invokes the following commands for you:

```
c89  -O -c mysub.c
c89  -o myappl mymain.o mysub.o
```

Note: The make utility requires that application program source files that are to be “maintained” through use of a makefile reside in z/OS UNIX files. To compile and bind z/OS XL C/C++ source files that are in data sets, you must use the c89 utility directly.

See *z/OS UNIX System Services Command Reference* for a description of the make utility. For a detailed discussion on how to create and use makefiles to manage application parts, see *z/OS UNIX System Services Programming Tools*.

Compiling with IPA

If you request Interprocedural Analysis (IPA) through the IPA compiler option, the compilation process changes significantly. IPA instructs the compiler to optimize your z/OS XL C/C++ program across compilation units, and to perform optimizations that are not otherwise available with the z/OS XL C/C++ compiler. You should refer to *z/OS XL C/C++ Programming Guide* for an overview of IPA processing before you invoke the compiler with the IPA compiler option.

Differences between the IPA compilation process and the regular compilation process are noted throughout this topic.

Figure 18 shows the flow of processing for a regular compilation:

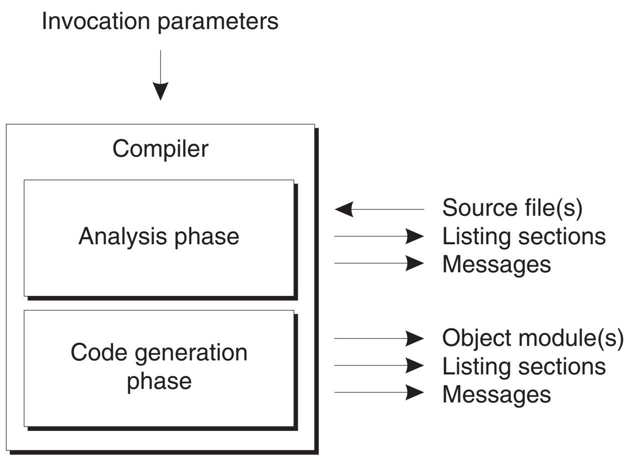


Figure 18. Flow of regular compiler processing

IPA processing consists of two separate steps, called the *IPA Compile step* and the *IPA Link step.*

The IPA Compile step

The IPA Compile step is similar to a regular compilation.

You invoke the IPA Compile step for each source file in your application by specifying the IPA(NOLINK) compiler option or by specifying **-Wc,IPA** or **-Wl -c** in z/OS UNIX System Services. The output of the IPA Compile step is an object file which contains IPA information, or both IPA information and conventional object code and data. The IPA information is an encoded form of the compilation unit with additional IPA-specific compile-time optimizations.

Figure 19 on page 357 shows the flow of IPA Compile step processing.

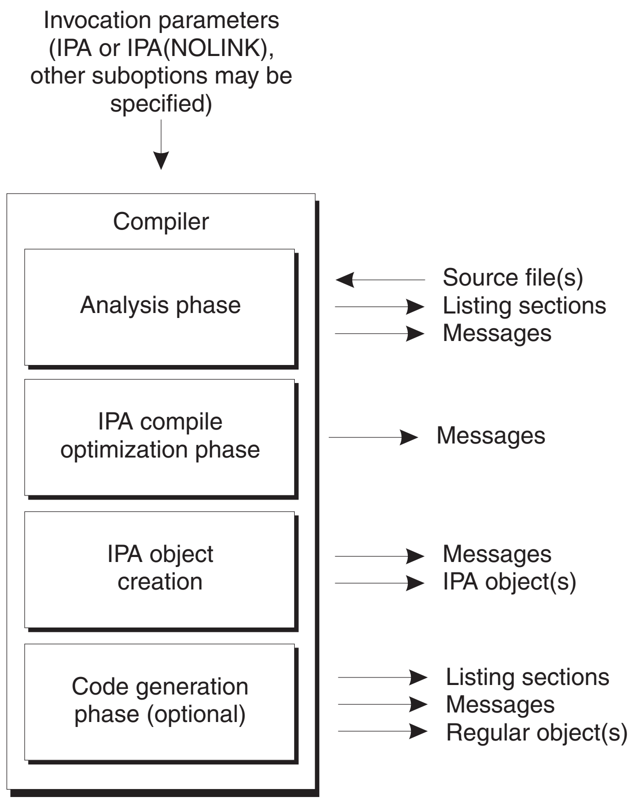


Figure 19. IPA Compile step processing

The same environments that support a regular compilation also support the IPA Compile step.

The IPA Link step

The IPA Link step is similar to the binding process.

You invoke the IPA Link step by specifying the IPA(LINK) compiler option or by specifying **-Wl** without specifying **-c** in z/OS UNIX System Services. This step links the user application program together by combining object files with IPA information, object files with conventional object code and data, and load module members. It merges IPA information, performs IPA Link-time optimizations, and generates the final object code and data.

Each application program module must be built with a single invocation of the IPA Link step. All parts must be available during the IPA Link step; missing parts may result in termination of IPA Link processing.

Figure 20 on page 358 shows the flow of IPA Link step processing:

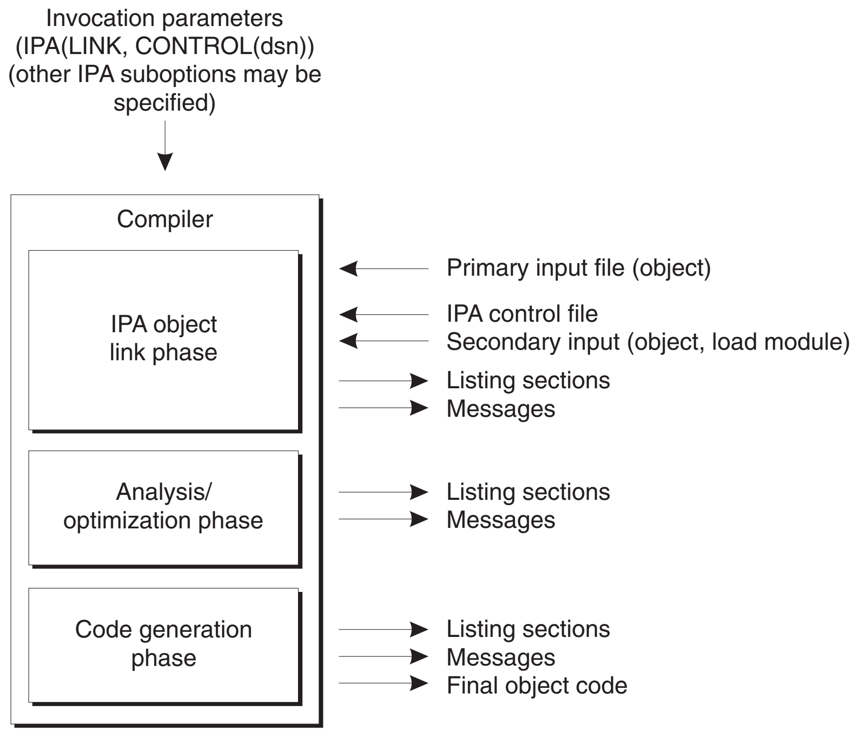


Figure 20. IPA Link step processing

Only c89, xlc, `c++` and z/OS batch support the IPA Link step. Refer to Chapter 8, "Using the IPA Link step with z/OS XL C/C++ programs," on page 375 for information about the IPA Link step.

Compiling with IPA(OBJONLY)

The full Interprocedural Analysis using the IPA Compile and IPA Link steps performs significant optimizations beyond those which are available using regular compilation. If problems occur, diagnosis may take significant time and effort.

The IPA(OBJONLY) compilation is an intermediate level of optimization. This results in a modified regular compile, not an IPA Compile step. Unlike the IPA Compile step, no IPA information is written to the object file.

During compilation, this step performs the same IPA-specific compile-time optimizations as the IPA Compile step, performs the requested non-IPA optimizations, and then generates optimized object code and data.

You invoke the compiler for each source file in your application by specifying the IPA(OBJONLY) compiler option.

The object file may be used by an IPA Link step, a prelink/link, or a bind. If it is used as input to an IPA Link step, no IPA Link-time optimizations can be performed for this compilation unit because no IPA information is available.

Figure 21 on page 359 shows the flow of processing for an IPA(OBJONLY) compilation.

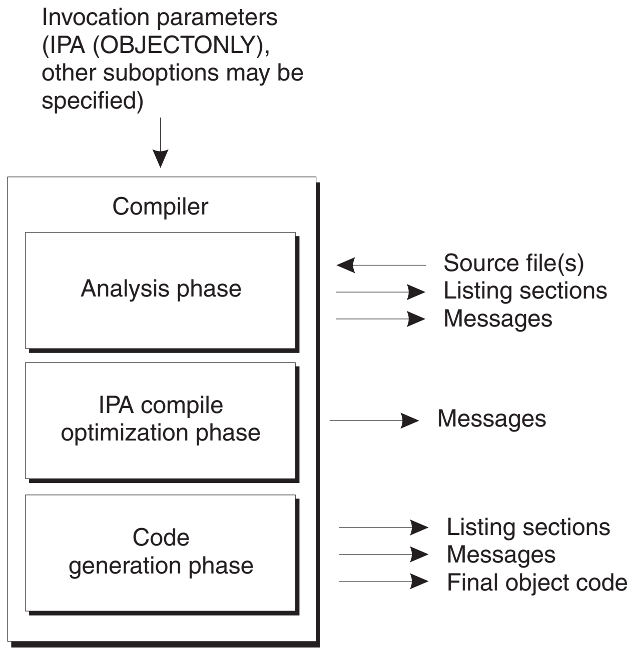


Figure 21. Compiling with IPA(OBJECTONLY)

Working with object files

z/OS object files are composed of a stream of 80 byte records. These may be binary object records, or link control statements. It is useful to be able to browse the contents of an object file, so that some basic information can be determined.

Browsing object files

Object files, which are sequential data sets or are members of a PDS or PDSE object library, can be browsed directly using the Program Development Facility (PDF) edit and browse options.

Object files, which are z/OS UNIX System Services files, can be browsed using the PDF **obrowse** command. z/OS UNIX files can be browsed using the TSO ISHELL command, and then using the V (View) action (V on the Command line, or equivalently **Browse records** from the File pull-down menu). This will result in a pop-up window for entering a record length. To force display in F 80 record mode, one would issue the following sequence of operations:

1. Enter the command: **obrowse file.o**

   Note that the file name is deliberately typed with an extra character. This will result in the display of an obrowse dialog panel with an error message that the file is not found. After pressing Enter, a second obrowse dialog is displayed to allow the file name to be corrected. This panel has an entry field for the record length.
2. Correct the file name and enter 80 in the record length entry field.
3. Browse the object records as you would a F 80 data set.

The hex display mode (enabled by the HEX ON primary command) allows the value of each byte to be displayed.

Identifying object file variations

Browse the object file and scroll to the end of the file. The last few records contain a character string, which lists the options used during compilation.

In addition, it is possible to identify the compiler mode used to generate the object file, as follows:

1. NOIPA

   Option text has ″`NOIPA`″.
2. IPA(NOOBJECT)

   Option text has ″`IPA (NOLINK, NOOBJ)`″. Towards the beginning of the file, an ESD record will contain the symbol ″`@@IPAOBJ`″. A second ESD record will contain the symbol ″`@@DOIPA`″.
3. IPA(OBJECT)

   Option text has ″`IPA (NOLINK, OBJ)`″. Towards the beginning of the file, an ESD record will contain the symbol ″`@@IPAOBJ`″. The IPA information will be separated from the ″real″ code and data by a delimiter END record with the comment ″of IPA object″. After the real code and data, there will be a second delimiter END record with the comment ″of object″.
4. IPA(OBJONLY)

   Option text has ″`IPA (OBJONLY)`″.

Using feature test macros

The compiler predefines feature test macros when certain features are available. For example, the _LONG_LONG macro is predefined if the compiler supports the `long long` data type. (Please refer to *z/OS XL C/C++ Language Reference* for further information on macros.

Using include files

The `#include` preprocessor directive allows you to retrieve source statements from secondary input files and incorporate them into your C/C++ program.

z/OS XL C/C++ Language Reference describes the `#include` directive. Its syntax is:

```
►►──#include──┬─<──┬──────┬──filename──>─┬──────────────────────────────────►◄
              │    └─//───┘               │
              └─″──┬──────┬──filename──″──┘
                   └─//───┘
```

The angle brackets specify system include files, and double quotation marks specify user include files.

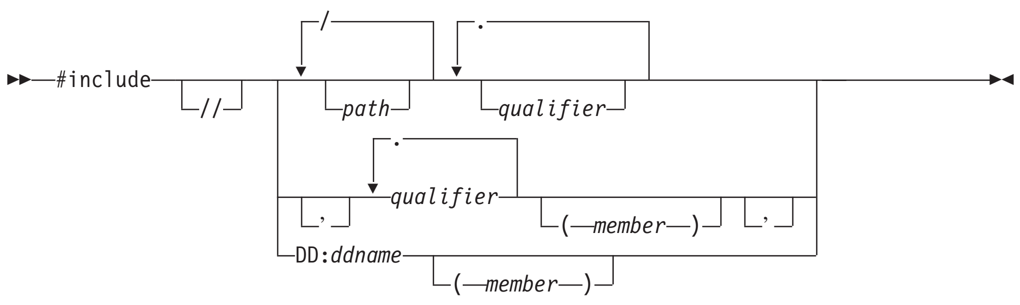
When you use the `#include` directive, you must be aware of the following:

- The *library search sequence*, the search order that XL C/C++ uses to locate the file. See “Search sequences for include files” on page 369 for more information on the library search sequence.
- The file-naming conversions that the XL C/C++ compiler performs.
- The area of the input record that contains sequence numbers when you are including files with different record formats. See *z/OS XL C/C++ Language Reference* for more information on **#pragma sequence**.

Specifying include file names

You can use the SEARCH and LSEARCH compiler options to specify search paths for system include files and user include files. For more information on these options, see “LSEARCH | NOLSEARCH” on page 171 and “SEARCH | NOSEARCH” on page 219.

You can specify *filename* of the `#include` directive in the following format:



The leading double slashes (//) not followed by a slash (in the first character of *filename*) indicate that the file is to be treated as a non-z/OS UNIX System Services file, hereafter called a data set.

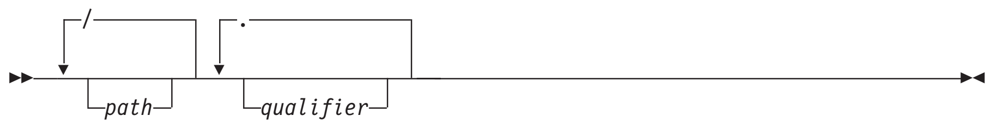
Note:

1. *filename* immediately follows the double slashes (//) without spaces.
2. Absolute data set names are specified by putting single quotation marks (’) around the name. Refer to the syntax diagram in this topic for this specification.
3. Absolute z/OS UNIX file names are specified by putting a leading slash (/) as the first character in the file name.
4. ddnames are always considered absolute.

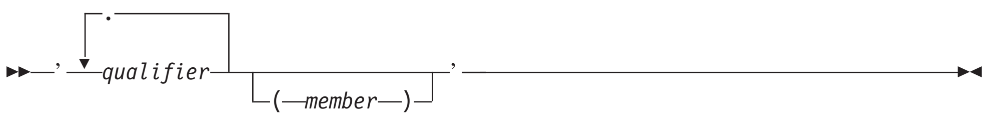
Forming file names

Refer to “Determining whether the file name is in absolute form” on page 365 for information on absolute file names. When the compiler performs a library search, it treats *filename* as either a z/OS UNIX System Services file name or a data set name. This depends on whether the library being searched is a z/OS UNIX library or MVS library. If the compiler treats *filename* as a z/OS UNIX file name, it does not perform any conversions on it. If it treats *filename* as a data set name (DSN), it performs the following conversion:

- For the first DSN format:

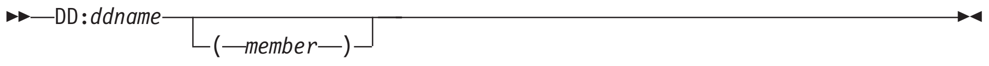


  The compiler:
  1. Uppercases *qualifier* and *path*
  2. Truncates each *qualifier* and *path* to 8 characters
  3. Converts the underscore character (which is invalid for a DSN) to the ’@’ character (hex 7c)
- For the second DSN format:



The compiler:
1. Uppercases the *qualifier* and *member*
2. Converts the underscore character (which is invalid for a DSN) to the ’@’ character (hex 7c)

- For the third DSN format:



The compiler:
1. Uppercases the `DD:`, *ddname*, and *member*
2. Converts the underscore character (which is invalid for a DSN) to the ’@’ character (hex 7c)

Forming data set names with LSEARCH | SEARCH options

When the *filename* specified in the `#include` directive is not in absolute form, the compiler combines it with different types of libraries to form complete data set specifications. These libraries may be specified by the LSEARCH or SEARCH compiler options. When the LSEARCH or SEARCH option indicates a data set then, depending on whether it is a ddname, sequential data set, or PDS, different parts of *filename* are used to form the ddname or data set name.

Forming DDname

Example: The leftmost qualifier of the *filename* in the `#include` directive is used when the *filename* is to be a ddname:

Invocation:
```
SEARCH(DD:SYSLIB)
```

Include directive:
```
#include "sys/afile.g.h"
```

Resulting `ddname`:
```
DD:SYSLIB(AFILE)
```

In this example, if your header file includes an underscore (_), for example, `#include "sys/afile_1.g.h"`, the resulting ddname is `DD:SYSLIB(AFILE@1)`.

Forming sequential data set names

Example: You specify libraries in the SEARCH | LSEARCH options as sequential data sets by using a trailing period followed by an asterisk (.*), or by a single asterisk (*). See “LSEARCH | NOLSEARCH” on page 171 to understand how to specify sequential data sets. All *qualifier*s and periods (.) in *filename* are used for sequential data set specification.

Invocation:
```
SEARCH(AA.*)
```

Include directive:
```
#include "sys/afile.g.h"
```

Resulting fully qualified data set name:
```
userid.AA.AFILE.G.H
```

Forming PDS name with LSEARCH | SEARCH + specification

Example: To specify libraries in the SEARCH and LSEARCH options as PDSs, use a period that is followed by a plus sign (.+), or a single plus sign (+). See "LSEARCH | NOLSEARCH" on page 171 to understand how PDSs are specified. When this is the case then all the *path*s, slashes (replaced by periods), and any *qualifier*s following the leftmost *qualifier* of the *filename* are appended to form the data set name. The leftmost *qualifier* is then used as the member name.

Invocation:
```
SEARCH('AA.+')
```

Include directive:
```
#include "sys/afile.g.h"
```

Resulting fully qualified data set name:
```
AA.SYS.G.H(AFILE)
```

and

Invocation:
```
SEARCH('AA.+')
```

Include directive:
```
#include "sys/bfile"
```

Resulting fully qualified data set name:
```
AA.SYS(BFILE)
```

Forming PDS with LSEARCH | SEARCH Options with No +

Example: When the LSEARCH or SEARCH option specifies a library but it neither ends with an asterisk (*) nor a plus sign (+), it is treated as a PDS. The leftmost qualifier of the *filename* in the `#include` directive is used as the member name.

Invocation:
```
SEARCH('AA')
```

Include directive:
```
#include "sys/afile.g.h"
```

Resulting fully qualified data set name:
```
AA(AFILE)
```

Examples of forming data set names

The following table gives the original format of the *filename* and the resulting converted name when you specify the NOOE option:

Table 38. Include filename conversions when NOOE is specified

`#include` Directive	Converted Name
Example 1. This *filename* is absolute because single quotation marks (') are used. It is a sequential data set. A library search is not performed. LSEARCH is ignored.	
`#include "'USER1.SRC.MYINCS'"`	`USER1.SRC.MYINCS`
Example 2. This *filename* is absolute because single quotation marks (') are used. The compiler attempts to open data set `COMIC/BOOK.OLDIES.K` and fails because it is not a valid data set name. A library search is not performed when *filename* is in absolute form. SEARCH is ignored.	
`#include <'COMIC/BOOK.OLDIES.K'>`	`COMIC/BOOK.OLDIES.K`
Example 3.	

Table 38. Include filename conversions when NOOE is specified (continued)

`#include` **Directive**	**Converted Name**
`SEARCH(LIB1.*,LIB2.+,LIB3) #include "sys/abc/xx"`	• first *opt* in `SEARCH SEQUENTIAL FILE =` *`userid`*`.LIB1.XX` • second *opt* in `SEARCH PDS =` *`userid`*`.LIB2.SYS.ABC(XX)` • third *opt* in `SEARCH PDS =` *`userid`*`.LIB3(XX)`
Example 4.	
`SEARCH(LIB1.*,LIB2.+,LIB3) #include "Sys/ABC/xx.x"`	• first *opt* in `SEARCH SEQUENTIAL FILE =` *`userid`*`.LIB1.XX.X` • second *opt* in `SEARCH PDS =` *`userid`*`.LIB2.SYS.ABC.X(XX)` • third *opt* in `SEARCH PDS =` *`userid`*`.LIB3(XX)`
Example 5.	
`SEARCH(LIB1.*,LIB2.+,LIB3) #include <sys/name_1>`	• first *opt* in `SEARCH SEQUENTIAL FILE =` *`userid`*`.LIB1.NAME@1` • second *opt* in `SEARCH PDS =` *`userid`*`.LIB2.SYS(NAME@1)` • third *opt* in `SEARCH PDS =` *`userid`*`.LIB3(NAME@1)`
Example 6.	
`SEARCH(LIB1.*,LIB2.+,LIB3) #include <Name2/App1.App2.H>`	• first *opt* in `SEARCH SEQUENTIAL FILE =` *`userid`*`.LIB1.APP1.APP2.H` • second *opt* in `SEARCH PDS =` *`userid`*`.LIB2.NAME2.APP2.H(APP1)` • third *opt* in `SEARCH PDS =` *`userid`*`.LIB3(APP1)`
Example 7. The PDS member named YEAREND of the library associated with the ddname PLANLIB is used. A library search is not performed when *filename* in the `#include` directive is in absolute form (ddname is used). SEARCH is ignored.	
`#include <dd:planlib(YEAREND)>`	`DD:PLANLIB(YEAREND)`

Search sequence

The following diagram describes the compiler file searching sequence:

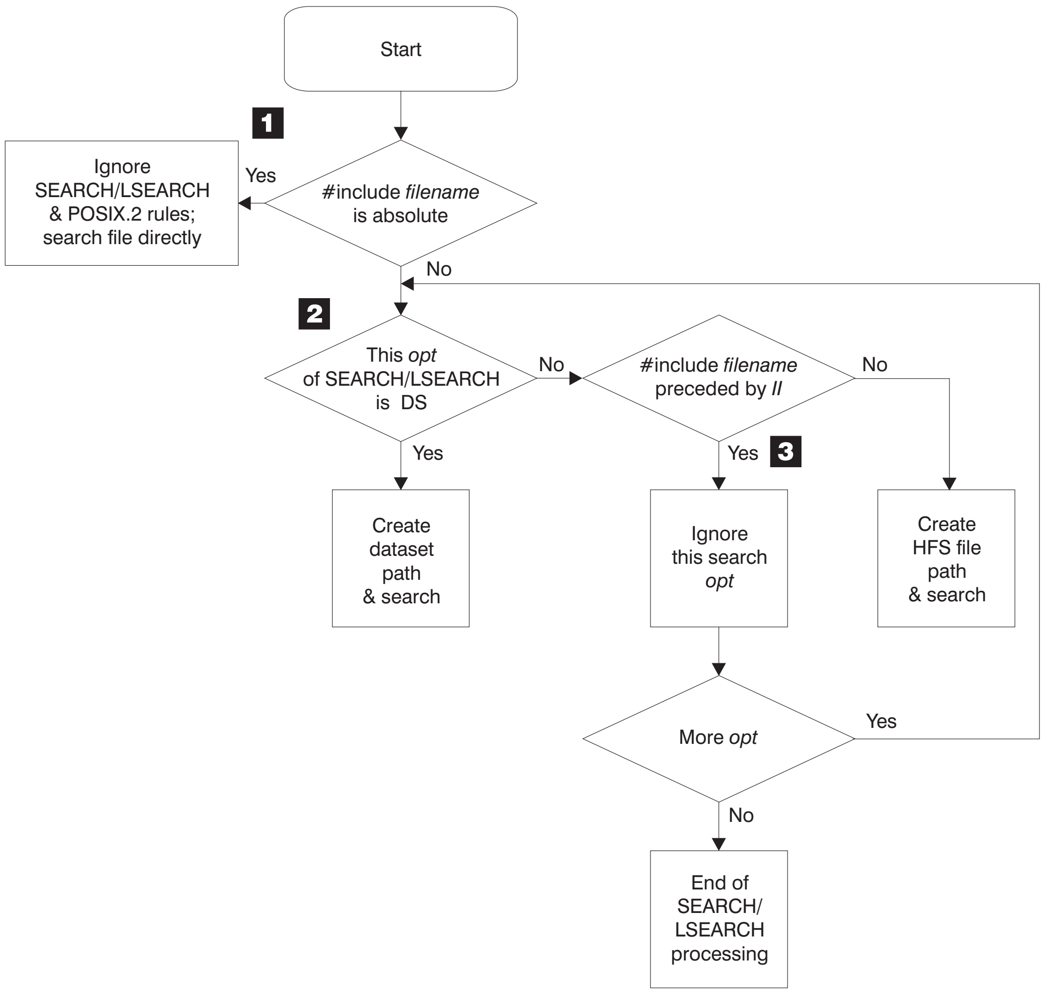

Start
1
Ignore SEARCH/LSEARCH & POSIX.2 rules; search file directly
Yes
#include *filename* is absolute
No
2
This *opt* of SEARCH/LSEARCH is DS
No
#include *filename* preceded by *//*
No
Yes
Yes
3
Create dataset path & search
Ignore this search *opt*
Create HFS file path & search
More *opt*
Yes
No
End of SEARCH/ LSEARCH processing

Figure 22. Overview of include file searching

1 The compiler opens the file without library search when the file name that is specified in `#include` is in absolute form. This also means that it bypasses the rules for the SEARCH and LSEARCH compiler options, and for POSIX.2. See Figure 23 on page 366 for more information on absolute file testing.

2 When the file name is not in absolute form, the compiler evaluates each option in SEARCH and LSEARCH to determine whether to treat the file as a data set or a z/OS UNIX System Services file search. The LSEARCH/SEARCH *opt* testing here is described in Figure 24 on page 368.

3 When the `#include` file name is not absolute, and is preceded by exactly two slashes (//), the compiler treats the file as a data set. It then bypasses all z/OS UNIX file options of the SEARCH and LSEARCH options in the search.

Determining whether the file name is in absolute form

The compiler determines if the file name that is specified in `#include` is in absolute form as follows:

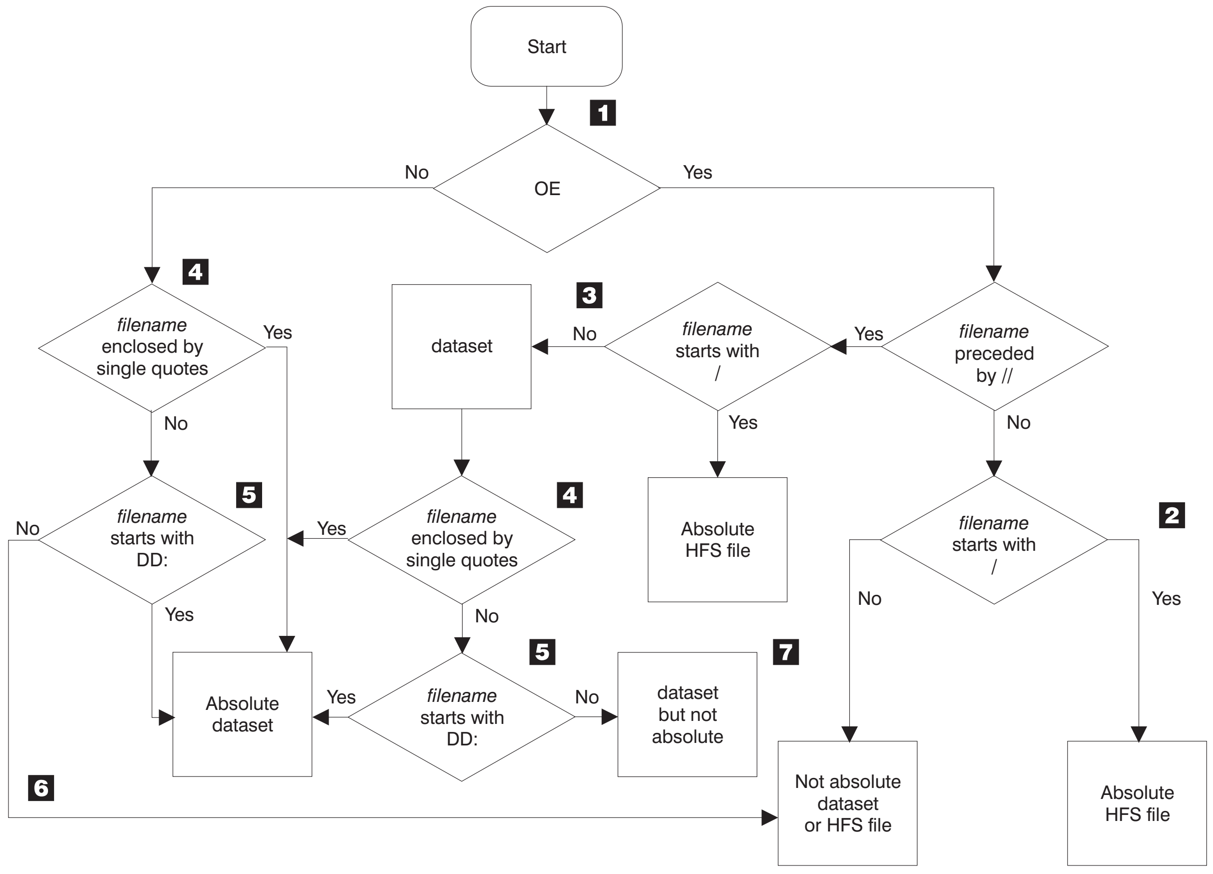


Figure 23. Testing if filename is in absolute form

1 The compiler first checks whether you specified OE.

2 When you specify OE, if double slashes (//) do not precede *filename*, and the file name starts with a slash (/), then *filename* is in absolute form and the compiler opens the file directly as a z/OS UNIX file. Otherwise, the file is not an absolute file and each *opt* in the SEARCH or LSEARCH compiler option determines if the file is treated as a z/OS UNIX file or data set in the search for the include file.

3 When OE is specified, if double slashes (//) precede *filename*, and the file name starts with a slash (/), then *filename* is in absolute form and the compiler opens the file directly as a z/OS UNIX file. Otherwise, the file is a data set, and more testing is done to see if the file is absolute.

4 If *filename* is enclosed in single quotation marks ('), then it is an absolute data set. The compiler directly opens the file and ignores the libraries that are specified in the LSEARCH or SEARCH options. If there are any invalid characters in *filename*, the compiler converts the invalid characters to at signs (@, hex 7c).

5 If you used the ddname format of the `#include` directive, the compiler uses the file associated with the ddname and directly opens the file as a data set. The libraries that are specified in the LSEARCH or SEARCH options are ignored.

6 If none of the conditions are true then *filename* is not in absolute format and each *opt* in the SEARCH or LSEARCH compiler option determines if the file is a z/OS UNIX file or a data set and then searches for the include file.

7 If none of the conditions are true, then *filename* is a data set, but it is not in absolute form. Only *opt*s in the SEARCH or LSEARCH compiler option that are in data set format are used in the search for include file.

For example:

```
 Options specified:

  OE

Include Directive:

  #include "apath/afile.h"       NOT absolute, z/OS UNIX file/
                                 MVS (no starting slash)
  #include "/apath/afile.h"      absolute z/OS UNIX file,
                                 (starts with 1 slash)
  #include "//apath/afile.h.c"   NOT absolute, MVS (starts with 2 slashes)
  #include "a.b.c"               NOT absolute, z/OS UNIX file/
                                 MVS (no starting slash)
  #include "///apath/afile.h"    absolute z/OS UNIX file,
                                 (starts with 3 slashes)
  #include "DD:SYSLIB"           NOT absolute, z/OS UNIX file/
                                 MVS (no starting slash)
  #include "//DD:SYSLIB"         absolute, MVS (DD name)
  #include "a.b(c)"              NOT absolute, z/OS UNIX file/
                                 MVS (no starting slash)
  #include "//a.b(c)"            NOT absolute, OS/MVS (PDS member name)
```

Using SEARCH and LSEARCH

When the file name in the `#include` directive is not in absolute form, the *opt*s in SEARCH are used to find system include files and the *opt*s in LSEARCH are used to find user include files. Each *opt* is a library path and its format determines if it is a z/OS UNIX System Services path or a data set path:

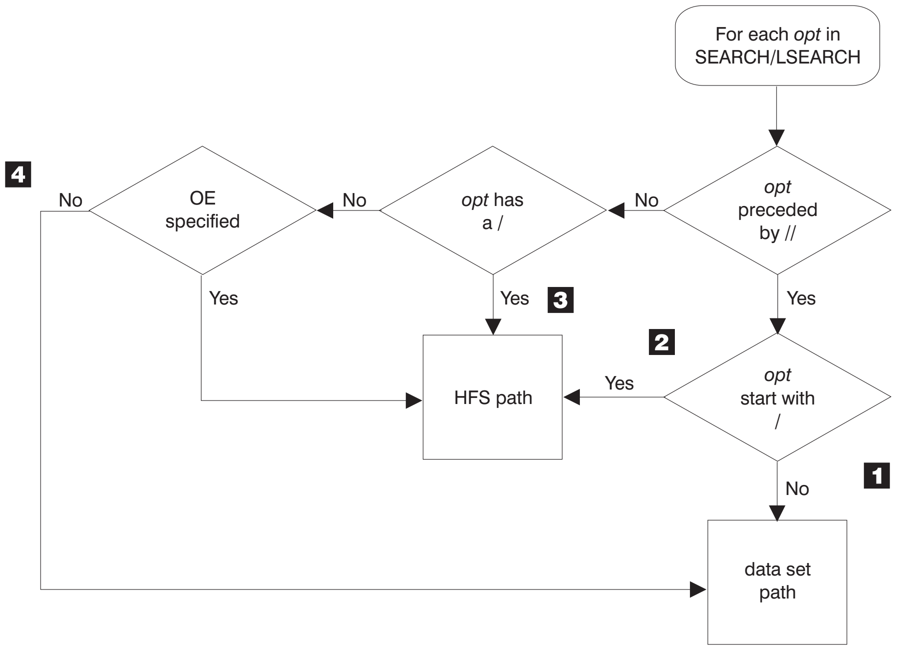


Figure 24. Determining if the SEARCH/LSEARCH opt is a z/OS UNIX path

Note:

1. If *opt* is preceded by double slashes (//) and *opt* does not start with a slash (/), then this path is a data set path.
2. If *opt* is preceded by double slashes (//) and *opt* starts with a slash (/), then this path is a z/OS UNIX path.
3. If *opt* is **not** preceded by double slashes (//) and *opt* starts with a slash (/), then this path is a z/OS UNIX path.
4. If *opt* is **not** preceded by double slashes (//), *opt* does not start with a slash (/) and NOOE is specified then this path is a data set path.

For example:

`SEARCH(./PATH)`	is an explicit z/OS UNIX path
`OE SEARCH(PATH)`	is treated as az/OS UNIX path
`NOOE SEARCH(PATH)`	is treated as a non-z/OS UNIX path
`NOOE SEARCH(//PATH)`	is an explicit non-z/OS UNIX path

Example: When combining the library with the file name specified on the `#include` directive, it is the form of the library that determines how the include file name is to be transformed:

```
Options specified:

  NOOE LSEARCH(Z, /u/myincs, (*.h)=(LIB(mac1)))

Include Directive:

  #include "apath/afile.h"

Resulting fully qualified include names:

1. userid.Z(AFILE)  (Z is non-z/OS UNIX file so file name is treated
as non-z/OS UNIX file)
```

```
2. /u/myincs/apath/afile.h   (/u/myincs is z/OS UNIX file so
file name is treated as z/OS UNIX file)
3. userid.MAC1.H(AFILE)            (afile.h matches *.h)
```

Example: A z/OS UNIX path specified on a SEARCH or LSEARCH option only combines with the file name specified on an `#include` directive if the file name is not explicitly stated as being MVS only. A file name is explicitly stated as being MVS only if two slashes (//) precede it, and *filename* does not start with a slash (/).

```
Options specified:

  OE LSEARCH(/u/myincs, q, //w)

Include Directive:

  #include "//file.h"

Resulting fully qualified include names

 userid.W(FILE)
```

`/u/myincs` and `q` would not be combined with `//file.h` because both paths are z/OS UNIX paths and `//file.h` is explicitly MVS.

The order in which options on the LSEARCH or SEARCH option are specified is the order that is searched.

See “LSEARCH | NOLSEARCH” on page 171 and “SEARCH | NOSEARCH” on page 219 for more information on these compiler options.

Search sequences for include files

The search path is a list of include paths, each of which may form the start of a fully qualified file name. The include path can be specified through the -I option. For the z/OS XL C/C++ compiler, it can also be specified through the SEARCH and LSEARCH options.

If the same z/OS UNIX System Services directory is specified in the search path multiple times, then only the first one is used. For example, `/usr/include` and `/usr/include/sys/..` resolve to the same z/OS UNIX System Services directory, therefore only the first path will be used in the final search path.

For the z/OS XL C/C++ compiler, if the same data set name is specified multiple times, then only the first one is used. For example, the data set names //’MYHLQ.SCEEH’ and //’MYHLQ.SCEEH’ are the same, therefore only the first data set will be used in the final search path. The data set names //’MYHLQ.SCEEH’ and //SCEEH are different, therefore both data sets will be used in the final search path. This reduction is an optimization which applies only to the include paths within the same or equivalent search option. A default search path is not merged.

The search order is effected by USERLIB concatenation normally found in JCL. It can contain multiple data sets, which are searched to find any user header files included in the source. All data sets specified on USERLIB concatenation are treated as one entry in the search sequence for the **#include_next** directive.

In the following USERLIB concatenation example, if an including file is located in data set DSN=JONES.LIB1.H and it contains a **#include_next** `test.h` directive, then, DSN=JONES.LIB2.H will not be searched to find `test.h` but rather the next entry in the search sequence for user include files. The search will continue using the search order for system include files.

```
//USERLIB  DD DSN=JONES.LIB1.H,DISP=SHR
//         DD DSN=JONES.LIB2.H,DISP=SHR
```

This restriction can be easily avoided by using the LSEARCH or SEARCH compiler option instead of USERLIB concatenation. In this example, specifying the LSEARCH(LIB1.+) LSEARCH(LIB2.+) compiler options will cause the DSN=JONES.LIB2.H data set to be searched to find the include file `test.h`.

The same restriction applies to SYSLIB concatenation as well and it can be avoided by using SEARCH option.

The status of the OE option affects the search sequence.

With the NOOE option

Search sequences for include files are used when the include file is not in absolute form. "Determining whether the file name is in absolute form" on page 365 describes the absolute form of include files.

If the include filename is not absolute, the compiler performs the library search as follows:

- For system include files:
  1. The search order as specified on the SEARCH option, if any
  2. The libraries specified on the SYSLIB DD statement
- For user include files:
  1. When the containing file is a z/OS UNIX System Services file, the search order as specified by the LSEARCH option, if any
  2. The libraries specified on the USERLIB DD statement
  3. The search order for system include files

Example: This example shows an excerpt from a JCL stream, that compiles a C program for a user whose user prefix is `JONES`:

```
//COMPILE  EXEC PROC=EDCC,
//             CPARM='SEARCH(''''BB.D'''',BB.F),LSEARCH(CC.X)'
//SYSLIB   DD DSN=JONES.ABC.A,DISP=SHR
//         DD DSN=ABC.B,DISP=SHR
//USERLIB  DD DSN=JONES.XYZ.A,DISP=SHR
//         DD DSN=XYZ.B,DISP=SHR
//SYSIN    DD DSN=JONES.ABC.C(D),DISP=SHR
  .
  .
  .
```

The search sequence that results from the preceding JCL statements is:

Table 39. Order of search for include files

Order of Search	For System Include Files	For User Include Files
First	BB.D	JONES.CC.X
Second	JONES.BB.F	JONES.XYZ.A
Third	JONES.ABC.A	XYZ.B
Fourth	ABC.B	BB.D
Fifth		JONES.BB.F

Table 39. Order of search for include files (continued)

Order of Search	For System Include Files	For User Include Files
Sixth		JONES.ABC.A
Seventh		ABC.B

With the OE option

Search sequences for include files are used when the include file is not in absolute form. "Determining whether the file name is in absolute form" on page 365 describes the absolute form of an include file.

If the include filename is not absolute, the compiler performs the library search as follows:

- For system include files:
  1. The search order as specified on the SEARCH option, if any
  2. The libraries specified on the SYSLIB DD statement
- For user include files:
  1. If you specified OE with a file name and the including file is a z/OS UNIX System Services file and a main source file, the directory of the file name specified with the OE option; otherwise, the directory of the including file
  2. The search order as specified by the LSEARCH option, if any
  3. The libraries specified on the USERLIB DD statement
  4. The search order for system include files

Example: The following shows an example where you are given a file `/r/you/cproc.c` that contains the following `#include` directives:

```
#include "/u/usr/header1.h"
#include "//aa/bb/header2.x"
#include "common/header3.h"
#include <header4.h>
```

And the following options:

```
OE(/u/crossi/myincs/cproc)
SEARCH(//V.+, /new/inc1, /new/inc2)
LSEARCH(//(*.x)=(lib(AAA)), /c/c1, /c/c2)
```

The include files would be searched as follows:

Table 40. Examples of search order for z/OS UNIX System Services

`#include` Directive Filename	Files in Search Order
Example 1. This is an absolute pathname, so no search is performed.	
`#include "/u/usr/header1.h"`	1. /u/usr/header.h
Example 2. This is a data set (starts with //) and is treated as such.	
"//aa/bb/header2.x"	1. *userid*.AAA(HEADER2) 2. DD:USERLIB(HEADER2) 3. *userid*.V.AA.BB.X(HEADER2) 4. DD:SYSLIB(HEADER2)
Example 3. This is a user include file with a relative path name. The search starts with the directory of the parent file or the name specified on the OE option if the parent is the main source file (in this case the parent file is the main source file so the OE suboption is chosen i.e. /u/crossi/myincs).	

Table 40. Examples of search order for z/OS UNIX System Services (continued)

`#include` Directive Filename	Files in Search Order
″common/header3.h″	1. /u/crossi/myincs/common/header3.h 2. /c/c1/common/header3.h 3. /c/c2/common/header3.h 4. DD:USERLIB(HEADER3) 5. *userid*.V.COMMON.H(HEADER3) 6. /new/inc1/common/header3.h 7. /new/inc2/common/header3.h 8. DD:SYSLIB(HEADER3)
Example 4. This is a system include file with a relative path name. The search follows the order of suboptions of the SEARCH option.	
<header4.h>	1. *userid*.V.H(HEADER4) 2. /new/inc1/common/header4.h 3. /new/inc2/common/header4.h 4. DD:SYSLIB(HEADER4)

Compiling z/OS XL C source code using the SEARCH option

The following data sets contain the commonly-used system header files for C: [3]

- CEE.SCEEH.H (standard header files)
- CEE.SCEEH.SYS.H (standard system header files)
- CEE.SCEEH.ARPA.H (standard internet operations headers)
- CEE.SCEEH.NET.H (standard network interface headers)
- CEE.SCEEH.NETINET.H (standard internet protocol headers)

To specify that the compiler search these data sets, code the option:

```
SEARCH('CEE.SCEEH.+')
```

These header files are also in the z/OS UNIX System Services directory `/usr/include`. To specify that the compiler search this directory, code the option:

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

This option is the default for the c89 utility.

IBM supplies this option as input to the Installation and Customization of the compiler. Your system programmer can modify it as required for your installation.

The cataloged procedures, REXX EXECs, and panels that are supplied by IBM for C specify the following data sets for the SYSLIB ddname by default:

- CEE.SCEEH.H (standard header files)
- CEE.SCEEH.SYS.H (standard system header files)

They are supplied for compatibility with previous releases, and will be overridden if SEARCH() is used.

Compiling z/OS XL C++ source code using the SEARCH option

The following data sets contain the commonly-used system header files for z/OS XL C++: [3]

- CEE.SCEEH (standard C++ header files)
- CEE.SCEEH.H (standard header files)
- CEE.SCEEH.SYS.H (standard system header files)

3. The high-level qualifier may be different for your installation.

- CEE.SCEEH.ARPA.H (standard internet operations headers)
- CEE.SCEEH.NET.H (standard network interface headers)
- CEE.SCEEH.NETINET.H (standard internet protocol headers)
- CEE.SCEEH.T (standard template definitions)
- CBC.SCLBH.H (class library header files)

To specify that the compiler search these data sets, code the option:

```
SEARCH('CEE.SCEEH.+','CBC.SCLBH.+')
```

These header files are also in the z/OS UNIX System Services directories `/usr/include` and `/usr/lpp/cbclib/include`. To specify that the compiler search these directories, code the option:

```
SEARCH(/usr/include/,/usr/lpp/cbclib/include/)
```

This option is the default for the **cxx** z/OS UNIX System Services command.

IBM supplies this option as input to the installation and customization of the compiler. Your system programmer can modify it as required for your installation.

Chapter 8. Using the IPA Link step with z/OS XL C/C++ programs

Traditional optimizers only have the ability to optimize within a function (*intra-procedural optimization*) or at most within a compile unit (a single source file and its included header files). This is because traditional optimizers are only given one compile unit at a time.

Interprocedural optimizations are a class of optimizations that operate across function boundaries. IBM's Interprocedural Analysis (IPA) optimizer is designed to optimize complete modules at a time. This allows for increased optimization. By seeing more of the application at once, IPA is able to find more opportunities for optimization and this can result in much faster code.

In order to get a global module view of the application, IPA uses the following two pass process:

- The first pass is called an *IPA Compile*. During this pass, IPA collects all of the relevant information about the compile unit and stores it in the object file. This collected information is referred to as an IPA Object. You can optionally request that both an IPA object and a traditional object are created from an IPA Compile.
- The second pass is called the *IPA Link*. During this step, IPA acts like a traditional linker, and all object files, object libraries and side decks are fed to IPA so that it can optimize the entire module. The IPA Link step involves two separate optimizers. The IPA optimizer is run first and focuses optimizations across the module. IPA then breaks down the module into logical chunks called partitions and invokes the traditional optimizer with these partitions.

Whenever a compiler attempts to perform more optimizations, or looks at a larger portion of an application, more time, and more memory are required. Since IPA does more optimizations than either OPT(2) or OPT(3) and has a global view of the module, the compile time and memory used by the IPA Compile or Link process is more than that used by a traditional OPT(2) or OPT(3) compilation.

The first two topics of this information provide several examples on how to create modules (with a `main`) or DLLs using IPA. The third topic discusses the Profile-Directed Feedback option that can be used with IPA to get even more performance benefits. The fourth topic gives some reference information on IPA-specific subjects, like the IPA control file. The final topic provides some hints and tips for troubleshooting situations that come up when compiling and debugging IPA applications. All example source can be found in the sample data set SCCNSAM. The names of the sample data set members are given in each example below.

Invoking IPA using the c89 and xlc utilities

You can invoke the IPA Compile step, the IPA Link step, or both. The step that c89 invokes depends upon the invocation parameters and type of files you specify. You must specify the `I` phase indicator along with the W option of the c89 utility.

If you invoke the c89 utility with at least one source file and the **-c** option and the -WI option, c89 automatically specifies the IPA(NOLINK) option and invokes the IPA Compile step. For example, the following command invokes the IPA Compile step for the source file `hello.c`:

```
c89 -c -WI hello.c
```

The syntax when using the xlc utility is:

```
c89 -c -qipa hello.c
```

If you invoke the c89 utility with the -WI option and with at least one object file, do not specify the **-c** option and do not specify any source files. c89 automatically specifies IPA(LINK) and automatically invokes the IPA Link step and the binder. For example, the following command invokes the IPA Link step and the binder, to create a program called `hello`:

```
c89 -o hello -WI hello.o
```

The syntax when using the xlc utility is:

```
c89 -o hello -qipa hello.o
```

If you invoke c89 with the -WI option and with at least one source file for compilation and any number of object files, and do not specify the **-c** c89 compiler option, **c89** automatically invokes the IPA Compile step once for each compilation unit and the IPA Link step once for the entire program. It then invokes the binder. For example, the following command invokes the IPA Compile step, the IPA Link step, and the binder to create a program called `foo`:

```
c89 -o foo -WI,object foo.c
```

The syntax when using the xlc utility is:

```
c89 -o foo -qipa=object foo.c
```

When linking an application built with IPA(PDF1), you must specify **-WI,PDF1** so that the application links correctly.

Specifying options

When using c89, you can pass options to IPA, as follows:

- If you specify -WI, followed by IPA suboptions, c89 passes those suboptions to both the IPA Compile step and the IPA Link step (provided the IPA Link step is invoked)
- If you specify **-Wc**, followed by compiler options, c89 passes those options only to the IPA Compile step
- If you specify **-WI,I**, followed by compiler options, c89 passes those options only to the IPA Link step

The following is an example of passing options using the c89 utility:

```
c89 -O2 -WI,noobject -Wc,source -Wl,I,"maxmem(2048)" file.c
```

If you specify the previous command, you pass the IPA(NOOBJECT) and the SOURCE option to the IPA Compile step, and the MAXMEM(2048) option to both the IPA Compile and the IPA Link step.

The syntax when using the xlc utility is:

```
c89 -O2 -qipa=noobject -qsource -qmaxmem=2048 hello.c
```

Other considerations

The c89 and xlc utilities automatically generate all INCLUDE and LIBRARY IPA Link control statements.

IPA under c89 and xlc supports the following types of files:
- MVS PDS members
- Sequential data sets
- z/OS UNIX System Services files
- z/OS UNIX archive (.a) files

Compiling under z/OS batch

To compile your C/C++ source program under batch, you can either use the cataloged procedures that IBM supplies, or write your own JCL statements.

Using cataloged procedures for IPA Link

You can use one of the following IBM-supplied cataloged procedures.

EDCI Run the IPA Link step for a non-XPLINK 31-bit C program

EDCQI
Run the IPA Link step for a 64-bit C program

EDCXI
Run the IPA Link step for a 31-bit or 64-bit XPLINK C program

CBCI Run the IPA Link step for a 31-bit non-XPLINK C++ program

CBCQI
Run the IPA Link step for a 64-bit C++ program

CBCXI
Run the IPA Link step for a 31-bit or 64-bit XPLINK C++ program

Creating a module with IPA

This topic describes creating a module that contains the function `main`.

Example 1. all C parts

The simplest case for IPA is an application that does not import any information from a DLL, and that is all in a single language that supports IPA. The following example covers this case. The sample programs mentioned here can be found in the sample data set with the member names given here.

The first example shows a simple application that is made up of three source files. The target is to compile it with IPA(Level(2)) and OPT(2). We also want a full inline report and pseudo-assembly listing. This is the only example where the full source will be shown.

CCNGHI1.C

```
hello1.c:
    int seen_main;
    int seen_unused3;

    char *string1 = "Hello";
    char *stringU1 = "I'm not going to use this one!";


    int func2( char *);

    int main (void) {

      seen_main++;
      func2(string1);

      return 0;
   }

   float unused3( int a ) {
     seen_unused3++;
     return (float) a+seen_unused3;
   }
```

Figure 25. hello1.c example source code

CCNGHI2.C

```
hello2.c:
   #include <stdio.h>

   int seen_func2;
   int seen_unused2;
   char *string2 = "world!";

   int func3 (char *);

   int func2( char * s1) {

     seen_func2++;

     printf("%s ",s1);

     return func3(string2);
  }

  double unused2(float x) {

    seen_unused2++;

    return x+ seen_unused2;
  }
```

Figure 26. hello2.c example source code

CCNGHI3.C

```
hello3.c:
 #include <stdio.h>

 int seen_func3;
 int seen_unused1;

 int unused1(int x) {

   seen_unused1++;

   return x+ seen_unused1;
 }

 int func3( char * string2) {

    seen_func3++;

    printf("%s\n",string2);

   return seen_func3;
 }
```

Figure 27. hello3.c example source code

Building example 1. under z/OS UNIX System Services

For this example, the following table shows the mapping of SCCNSAM data set members to given file names:

SCCNSAM member name	Name used in this example
CCNGHI1	`hello1.c`
CCNGHI2	`hello2.c`
CCNGHI3	`hello3.c`

The following commands can be used to create this module under z/OS UNIX System Services:

```
c89 -c -2 -WI,NOOBJECT,LIST hello1.c hello2.c hello3.c
c89 -2 -WI,MAP,LEVEL\(2\) -Wl,I,INLRPT,LIST\(hello.lst\) -o hello hello1.o
  hello2.o hello3.o
```

The first **c89** command performs an IPA Compile on `hello1.c`, `hello2.c`, and `hello3.c`. The options after -WI are IPA suboptions, which are described below (for further information on these suboptions, see “IPA | NOIPA” on page 139):

NOOBJECT
: This compile performs an IPA Compile (since **-c** was specified). This option specifies that only IPA objects should be generated by the IPA Compile step. The NOOBJECT suboption will reduce the size of the output object files. It causes only the IPA object to be written to the output file. The NOOBJECT option should be used unless the traditional object is needed for debugging purposes or the object file may be passed in a non-IPA Link. NOOBJECT significantly shortens the overall compile time.

LIST
: This option tells IPA to save enough information that a listing with source file and line number information can be generated during the IPA(LINK) phase.

Note: -2 was specified on the IPA Compile step. While it is not strictly necessary, it does allow for faster code to be generated in some cases.

The second **c89** command does the IPA Link processing. Since -WI and **WI,I** were specified with .o files, c89 automatically turns on the LINK suboption of IPA. The -WI suboptions within this command are those that are valid for IPA(LINK):

MAP Generates additional information in the listing that shows where variables and data came from. For more information on specifying IPA(MAP), see "Using the IPA Link step listing" on page 317.

LEVEL(2)
Specifies that the maximum level of IPA optimization is to be used

The **-WI,I** option keyword specifies that these are compiler options that are to be passed to the IPA(LINK) step. Chapter 4, "Compiler Options," on page 41 documents the compiler options and whether they are valid during the IPA Link step. INLRPT triggers an inline report that shows the inlining that was done by IPA. LIST triggers a pseudo assembly listing for each partition.

Notes:
1. In this case, the name of the output file for the listing was provided as a suboption.
2. Even with IPA, the -2 or -3 option should be used to specify the opt level that the traditional optimizer should be called with.

This example shows the advantage of using discrete listing options (MAP, LIST, INLRPT) over using -V. -V may give you so much information that it creates a huge file. By using the individual options, you get more control and (with LIST) the ability to route the listing to the location of your choice without redirecting the output of your **c89** command.

Building example 1. in batch

For this example the following table shows the mapping of SCCNSAM data set members to given file names:

SCCNSAM member name	Name used in this example
CCNGHI1	IPA.SOURCE(HELLO1)
CCNGHI2	IPA.SOURCE(HELLO2)
CCNGHI3	IPA.SOURCE(HELLO3)

The following JCL can be used to create an object deck that can be linked to create the module (the link JCL is omitted for brevity):

```
/USERID1A JOB (127A,0329),'$MEM$',
//  MSGLEVEL=(2,0),MSGCLASS=S,CLASS=A,
//  NOTIFY=USERID1,REGION=1024M
//PROC JCLLIB ORDER=(CBC.SCCNPRC)
//*-------------------------------------------------------------
//* IPA Compile Step for hello1.c
//*-------------------------------------------------------------
//C001F336  EXEC EDCC,
//          INFILE='USERID1.IPA.SOURCE(HELLO1)',
//          OUTFILE='USERID1.IPA.OBJECT(HELLO1),DISP=SHR',
//          CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS  DD *
    IPA(NOOBJECT,LIST) RENT LONG OPT(2)
```

```
/*
//*-----------------------------------------------------------
//* IPA Compile Step for hello2.c
//*-----------------------------------------------------------
//C001F336  EXEC EDCC,
//          INFILE='USERID1.IPA.SOURCE(HELLO2)',
//          OUTFILE='USERID1.IPA.OBJECT(HELLO2),DISP=SHR',
//          CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS  DD *
    IPA(NOOBJECT,LIST) RENT LONG OPT(2)
/*
//*-----------------------------------------------------------
//* IPA Compile Step for hello3.c
//*-----------------------------------------------------------
//C001F336  EXEC EDCC,
//          INFILE='USERID1.IPA.SOURCE(HELLO3)',
//          OUTFILE='USERID1.IPA.OBJECT(HELLO3),DISP=SHR',
//          CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS  DD *
    IPA(NOOBJECT,LIST) RENT LONG OPT(2)
/*
//*-------------------------------------------------------------
//* IPA Link Step for the hello module
//*-------------------------------------------------------------
//C001F336  EXEC EDCI,
//          OUTFILE='USERID1.IPALINK.OBJECT(HELLO),DISP=SHR',
//          IPARM='OPTFILE(DD:OPTIONS)'
//* The following line sets up an input file that just includes all
//*   the IPA Compile Step object files.
//SYSIN  DD DATA,DLM='/>'
  INCLUDE OBJECT(HELLO1,HELLO2,HELLO3)
/>
//* The following line redirects the listing
//SYSCPRT DD DSN=USERID1.IPA.LISTING(HELLO),DISP=SHR
//* These are the options used
//OPTIONS  DD DATA,DLM='/>'
    IPA(LINK,MAP,LEVEL(2)) OPT(2) INLRPT LIST RENT LONGNAME
/>
//* The following line gives the object library
//OBJECT DD DSN=USERID1.IPA.OBJECT,DISP=SHR
```

The options used are the same as those given in “Building example 1. under z/OS UNIX System Services” on page 379 with the exception that IPA(LINK) should be explicitly specified, and RENT, and LONGNAME are not the default for C in batch so they also need to be specified. This sample JCL was created using the standard cataloged procedures shipped with the z/OS XL C/C++ compiler.

The generated file `hello.lst` is as follows:

```
15694-A01 V1.11 z/OS XL C/C++ IPA                                    DD:SYSIN                    04/29/2009 04:55:30   Page     1

                                      * * * * *   P R O L O G   * * * * *

  Compile Time Library . . . . . . : 410B0000
  Command options:
     Primary input name. . . . . . : DD:SYSIN
     Compiler options. . . . . . . : *IPA(LINK,MAP,LEVEL(2),DUP,ER,NONCAL,NOUPCASE,NOPDF1,NOPDF2,NOPDFNAME,NOCONTROL)
                                   : *NOGONUMBER  *NOHOT       *NOALIAS     *TERMINAL    *LIST        *NOXREF      *NOATTR
                                   : *NOOFFSET    *MEMORY      *NOCSECT     *NODFP       *LIBANSI     *FLAG(I)
                                   : *NOTEST(NOSYM,NOBLOCK,NOLINE,NOPATH,HOOK)           *OPTIMIZE(2)
                                   : *INLINE(AUTO,REPORT,1000,8000)         *OPTFILE(DD:OPTIONS)      *NOSERVICE   *NOOE
                                   : *NOLOCALE    *HALT(16)    *NOGOFF      *NOSPLITLIST

                               * * * * *   E N D   O F   P R O L O G   * * * * *
15694-A01 V1.11 z/OS XL C/C++ IPA                                    DD:SYSIN                    04/29/2009 04:55:30   Page     2

                             * * * * *   O B J E C T   F I L E   M A P   * * * * *

 *ORIGIN  IPA  FILE ID  FILE NAME
    P              1   //DD:SYSIN
    PI     Y       2   USERID1.IPA.OBJECT(HELLO1)
    PI     Y       3   USERID1.IPA.OBJECT(HELLO2)
    PI     Y       4   USERID1.IPA.OBJECT(HELLO3)
    L              5   CEE.SCEELKED(PRINTF)
    L              6   CEE.SCEELKED(CEESG003)

 ORIGIN:  P=primary input     PI=primary INCLUDE       SI=secondary INCLUDE   IN=internal
          A=automatic call     U=UPCASE automatic call   R=RENAME card          L=C Library

                      * * * * *   E N D   O F   O B J E C T   F I L E   M A P   * * * * *
```

Figure 28. Example of an IPA listing (Part 1 of 6)

```
15694-A01 V1.11 z/OS XL C/C++ IPA                            DD:SYSIN                 04/29/2009 04:55:30   Page     3
                       * * * * *   C O M P I L E R   O P T I O N S   M A P   * * * * *

 SOURCE FILE ID      COMPILE OPTIONS
             1        *AGGRCOPY(NOOVERLAP)      *NOALIAS     *ANSIALIAS   *ARCH(5)     *ARGPARSE    *NOASCII
                      *ASSERT(RESTRICT)         *BITFIELD(UNSIGNED)       *CHARS(UNSIGNED)          *NOCOMPACT   *NOCOMPRESS
                      *NOCONVLIT   *NOCSECT     *NODEBUG     *NODFP       *NODLL(NOCALLBACKANY)     *ENUMSIZE(SMALL)
                      *EXECOPS     *NOEXPORTALL *FLOAT(HEX,FOLD,NOMAF,NORRM,AFP(NOVOLATILE))        *NOGOFF      *NOGONUMBER
                      *NOHGPR(NOPRESERVE)       *NOHOT       *NOIGNERRNO  *ILP32       *NOINITAUTO
                      *INLINE(AUTO,NOREPORT,100,1000)        *IPA(NOLINK,NOOBJ,COM,OPT,NOGONUM)     *LANGLVL(EXTENDED)
                      *NOLIBANSI   *NOLOCALE    *LONGNAME    *MAXMEM(2097152)          *OPTIMIZE(2) *PLIST(HOST) *PREFETCH
                      *REDIR       *RENT        *NOROCONST   *ROUND(Z)    *ROSTRING    *NORTCHECK   *NOSERVICE   *SPILL(128)
                      *START       *STRICT      *NOSTRICT_INDUCTION       *TARGET(LE,zOSV1R11)      *TUNE(5)
                      *UNROLL(AUTO)             *NOUPCONV    *NOWSIZEOF   *NOXPLINK

             2        *AGGRCOPY(NOOVERLAP)      *NOALIAS     *ANSIALIAS   *ARCH(5)     *ARGPARSE    *NOASCII
                      *ASSERT(RESTRICT)         *BITFIELD(UNSIGNED)       *CHARS(UNSIGNED)          *NOCOMPACT   *NOCOMPRESS
                      *NOCONVLIT   *NOCSECT     *NODEBUG     *NODFP       *NODLL(NOCALLBACKANY)     *ENUMSIZE(SMALL)
                      *EXECOPS     *NOEXPORTALL *FLOAT(HEX,FOLD,NOMAF,NORRM,AFP(NOVOLATILE))        *NOGOFF      *NOGONUMBER
                      *NOHGPR(NOPRESERVE)       *NOHOT       *NOIGNERRNO  *ILP32       *NOINITAUTO
                      *INLINE(AUTO,NOREPORT,100,1000)        *IPA(NOLINK,NOOBJ,COM,OPT,NOGONUM)     *LANGLVL(EXTENDED)
                      *NOLIBANSI   *NOLOCALE    *LONGNAME    *MAXMEM(2097152)          *OPTIMIZE(2) *PLIST(HOST) *PREFETCH
                      *REDIR       *RENT        *NOROCONST   *ROUND(Z)    *ROSTRING    *NORTCHECK   *NOSERVICE   *SPILL(128)
                      *START       *STRICT      *NOSTRICT_INDUCTION       *TARGET(LE,zOSV1R11)      *TUNE(5)
                      *UNROLL(AUTO)             *NOUPCONV    *NOWSIZEOF   *NOXPLINK

             3        *AGGRCOPY(NOOVERLAP)      *NOALIAS     *ANSIALIAS   *ARCH(5)     *ARGPARSE    *NOASCII
                      *ASSERT(RESTRICT)         *BITFIELD(UNSIGNED)       *CHARS(UNSIGNED)          *NOCOMPACT   *NOCOMPRESS
                      *NOCONVLIT   *NOCSECT     *NODEBUG     *NODFP       *NODLL(NOCALLBACKANY)     *ENUMSIZE(SMALL)
                      *EXECOPS     *NOEXPORTALL *FLOAT(HEX,FOLD,NOMAF,NORRM,AFP(NOVOLATILE))        *NOGOFF      *NOGONUMBER
                      *NOHGPR(NOPRESERVE)       *NOHOT       *NOIGNERRNO  *ILP32       *NOINITAUTO
                      *INLINE(AUTO,NOREPORT,100,1000)        *IPA(NOLINK,NOOBJ,COM,OPT,NOGONUM)     *LANGLVL(EXTENDED)
                      *NOLIBANSI   *NOLOCALE    *LONGNAME    *MAXMEM(2097152)          *OPTIMIZE(2) *PLIST(HOST) *PREFETCH
                      *REDIR       *RENT        *NOROCONST   *ROUND(Z)    *ROSTRING    *NORTCHECK   *NOSERVICE   *SPILL(128)
                      *START       *STRICT      *NOSTRICT_INDUCTION       *TARGET(LE,zOSV1R11)      *TUNE(5)
                      *UNROLL(AUTO)             *NOUPCONV    *NOWSIZEOF   *NOXPLINK


                * * * * *   E N D   O F   C O M P I L E R   O P T I O N S   M A P   * * * * *
15694-A01 V1.11 z/OS XL C/C++ IPA                            DD:SYSIN                 04/29/2009 04:55:30   Page     4

                              * * * * *   I N L I N E   R E P O R T   * * * * *

                     IPA Inline Report (Summary)

 Reason:    P : #pragma noinline was specified for this routine
            F : #pragma inline was specified for this routine
            A : Automatic inlining
            C : Partition conflict
            N : Not IPA Object
            - : No reason
 Action:    I : Routine is inlined at least once
            L : Routine is initially too large to be inlined
            T : Routine expands too large to be inlined
            C : Candidate for inlining but not inlined
            N : No direct calls to routine are found in file (no action)
            U : Some calls not inlined due to recursion or parameter mismatch
            - : No action
 Status:    D : Internal routine is discarded
            R : A direct call remains to internal routine (cannot discard)
            A : Routine has its address taken (cannot discard)
            E : External routine (cannot discard)
            - : Status unchanged
 Calls/I      : Number of calls to defined routines / Number inline
 Called/I     : Number of times called / Number of times inlined

 Reason  Action  Status   Size (init)    Calls/I   Called/I     Name

    A      I       D      0 (40)           2/1       1/1        func2
    A      I       D      0 (32)           1/0       1/1        func3
    A      N       -      38 (28)          1/1       0          main
    N      -       E      0                0         2/0        PRINTF
 Mode = AUTO     Inlining Threshold =    1000    Expansion Limit =    8000
```

Figure 28. Example of an IPA listing (Part 2 of 6)

```
15694-A01 V1.11 z/OS XL C/C++ IPA                                   DD:SYSIN                 04/29/2009 04:55:30   Page     5

                     IPA Inline Report (Call Structure)

  Defined Subprogram    : main
      Calls To(1,1)     : func2(1,1)
   Called From          : 0

  Defined Subprogram    : func2
      Calls To(2,1)     : func3(1,1)
                          PRINTF(1,0)
   Called From(1,1)     : main(1,1)

  Defined Subprogram    : PRINTF
      Calls To          : 0
   Called From(2,0)     : func3(1,0)
                          func2(1,0)

  Defined Subprogram    : func3
      Calls To(1,0)     : PRINTF(1,0)
   Called From(1,1)     : func2(1,1)


                        * * * * *   E N D   O F   I N L I N E   R E P O R T   * * * * *
15694-A01 V1.11 z/OS XL C/C++ IPA                                Partition 1                 04/29/2009 04:55:30   Page     6

                               * * * * *   P A R T I T I O N   M A P   * * * * *

 PARTITION 1 OF 1

 PARTITION SIZE:
    Actual: 3800
     Limit: 1572864

 PARTITION CSECT NAMES:
      Code: none
    Static: none
      Test: none

 PARTITION DESCRIPTION:
    Primary partition

 COMPILER OPTIONS FOR PARTITION 1:
   *AGGRCOPY(NOOVERLAP)      *NOALIAS     *ARCH(5)     *ARGPARSE    *ATTR        *NOCOMPACT   *NOCOMPRESS  *NOCSECT     *NODLL
   *EXECOPS     *FLOAT(HEX,FOLD,AFP)      *NOGOFF      *NOGONUMBER  *NOIGNERRNO  *ILP32       *NOINITAUTO
   *INLINE(AUTO,REPORT,1000,8000)         *IPA(LINK)   *LIBANSI     *LIST        *NOLOCALE    *LONGNAME    *MAXMEM(2097152)
   *OPTIMIZE(2) *PLIST(HOST) *PREFETCH    *REDIR       *RENT        *NOROCONST   *SPILL(128)  *START       *STRICT
   *NOSTRICT_INDUCTION       *NOTEST      *TUNE(5)     *NOXPLINK    *NOXREF

 SYMBOLS IN PARTITION 1:

  *TYPE     FILE ID    SYMBOL
     D          1     string1
     D          2     string2
     F          1     main

   TYPE:  F=function     D=data

 SOURCE FILES FOR PARTITION 1:

  *ORIGIN     FILE ID     SOURCE FILE NAME
      P           1      //'USERID1.IPA.SOURCE(HELLO1)'
      P           2      //'USERID1.IPA.SOURCE(HELLO2)'
      P           3      //'USERID1.IPA.SOURCE(HELLO3)'


  ORIGIN:  P=primary input     PI=primary INCLUDE
```

Figure 28. Example of an IPA listing (Part 3 of 6)

```
15694-A01 V1.11 z/OS XL C/C++ IPA                               Partition 1                04/29/2009 04:55:30   Page     7

 OFFSET OBJECT CODE        LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G

                           Timestamp and Version Information
 000000  F2F0  F0F9                                            =C'2009'          Compiled Year
 000004  F0F4  F2F9                                            =C'0429'          Compiled Date MMDD
 000008  F0F4  F5F5  F2F4                                      =C'045524'        Compiled Time HHMMSS
 00000E  F0F1  F1F1  F0F0                                      =C'011100'        Compiler Version

 000014  0096  ****                                            AL2(150),C'...'   Saved Options String
                           Timestamp and Version End

15694-A01 V1.11 z/OS XL C/C++ IPA                               Partition 1: main          04/29/2009 04:55:30   Page     8

 OFFSET OBJECT CODE        LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G

 0000B8                    000010 |     1   main     DS       0D
 0000B8  47F0  F022        000010 |     1            B        34(,r15)
 0000BC  01C3C5C5                                             CEE eyecatcher
 0000C0  000000A8                                             DSA size
 0000C4  000000C8                                             =A(PPA1-main)
 0000C8  47F0  F001        000010 |     1            B        1(,r15)
 0000CC  58F0  C31C        000010 |     1            L        r15,796(,r12)
 0000D0  184E              000010 |     1            LR       r4,r14
 0000D2  05EF              000010 |     1            BALR     r14,r15
 0000D4  00000000                                             =F'0'
 0000D8  07F3              000010 |     1            BR       r3
 0000DA  90E5  D00C        000010 |     1            STM      r14,r5,12(r13)
 0000DE  58E0  D04C        000010 |     1            L        r14,76(,r13)
 0000E2  4100  E0A8        000010 |     1            LA       r0,168(,r14)
 0000E6  5500  C314        000010 |     1            CL       r0,788(,r12)
 0000EA  4130  F03A        000010 |     1            LA       r3,58(,r15)
 0000EE  4720  F014        000010 |     1            BH       20(,r15)
 0000F2  58F0  C280        000010 |     1            L        r15,640(,r12)
 0000F6  90F0  E048        000010 |     1            STM      r15,r0,72(r14)
 0000FA  9210  E000        000010 |     1            MVI      0(r14),16
 0000FE  50D0  E004        000010 |     1            ST       r13,4(,r14)
 000102  18DE              000010 |     1            LR       r13,r14
 000104                    End of Prolog

 000104  5820  C1F4        000000 |                  L        r2,_CEECAA_(,r12,500)
 000108  5840  306A        000010 |     1            L        r4,=Q(@STATIC)(,r3,106)
 00010C  C050  0000  002E  000000                    LARL     r5,F'46'
 000112  58F0  306E        000013 |     2  +         L        r15,=V(printf)(,r3,110)
 000116  4110  D098        000013 |     2  +         LA       r1,#MX_TEMP1(,r13,152)
 00011A  5804  2000        000010 |     1            L        r0,string1(r4,r2,0)
 00011E  5000  D09C        000013 |     2  +         ST       r0,#MX_TEMP1(,r13,156)
 000122  4100  500F        000013 |     2  +         LA       r0,+CONSTANT_AREA(,r5,15)
 000126  5000  D098        000013 |     2  +         ST       r0,#MX_TEMP1(,r13,152)
 00012A  0DEF              000013 |     2  +         BASR     r14,r15
 00012C  5804  2004        000013 |     2  +         L        r0,string2(r4,r2,4)
 000130  5000  D09C        000017 |     3  +         ST       r0,#MX_TEMP1(,r13,156)
 000134  4100  5013        000017 |     3  +         LA       r0,+CONSTANT_AREA(,r5,19)
 000138  58F0  306E        000017 |     3  +         L        r15,=V(printf)(,r3,110)
 00013C  5000  D098        000017 |     3  +         ST       r0,#MX_TEMP1(,r13,152)
 000140  4110  D098        000017 |     3  +         LA       r1,#MX_TEMP1(,r13,152)
 000144  0DEF              000017 |     3  +         BASR     r14,r15
 000146  41F0  0000        000015 |     1            LA       r15,0
 00014A                    000015 |     1   @1L3     DS       0H

 00014A                    Start of Epilog
 00014A  180D              000016 |     1            LR       r0,r13
 00014C  58D0  D004        000016 |     1            L        r13,4(,r13)
 000150  58E0  D00C        000016 |     1            L        r14,12(,r13)
 000154  9825  D01C        000016 |     1            LM       r2,r5,28(r13)
 000158  051E              000016 |     1            BALR     r1,r14
 00015A  0707              000016 |     1            NOPR     7

 00015C                    Start of Literals
 00015C  00000000                                             =Q(@STATIC)
 000160  00000000                                             =V(printf)
 000164                    End of Literals
```

Figure 28. Example of an IPA listing (Part 4 of 6)

```
15694-A01 V1.11 z/OS XL C/C++ IPA                                Partition 1: main            04/29/2009 04:55:30   Page     9

 OFFSET OBJECT CODE        LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G


                           ***   General purpose registers used: 1111110000001111
                           ***   Floating point  registers used: 1111111100000000
                           ***   Size of register spill area: 128(max) 0(used)
                           ***   Size of dynamic storage: 168
                           ***   Size of executable code: 164

 000164  0000  0000

                           Constant Area
 000168  C8859393 96000000 A6969993 845A006C    |Hello...world!.%|
 000178  A240006C A21500                        |s .%s..         |
15694-A01 V1.11 z/OS XL C/C++ IPA                                Partition 1                  04/29/2009 04:55:30   Page    10

 OFFSET OBJECT CODE        LINE#  FILE#    P S E U D O   A S S E M B L Y   L I S T I N G

                           PPA1: Entry Point Constants
 000180  1CCEA106                                        =F'483303686'      Flags
 000184  00000108                                        =A(PPA2-main)
 000188  00000000                                        =F'0'              No PPA3
 00018C  00000000                                        =F'0'              No EPD
 000190  FF000000                                        =F'-16777216'      Register save mask
 000194  00000000                                        =F'0'              Member flags
 000198  90                                              =AL1(144)          Flags
 000199  000000                                          =AL3(0)            Callee's DSA use/8
 00019C  0040                                            =H'64'             Flags
 00019E  0012                                            =H'18'             Offset/2 to CDL
 0001A0  00000000                                        =F'0'              Reserved
 0001A4  50000052                                        =F'1342177362'     CDL function length/2
 0001A8  FFFFFF38                                        =F'-200'           CDL function EP offset
 0001AC  38260000                                        =F'942014464'      CDL prolog
 0001B0  40090049                                        =F'1074331721'     CDL epilog
 0001B4  00000000                                        =F'0'              CDL end
 0001B8  0004  ****                                      AL2(4),C'main'
                           PPA1 End

                           PPA2: Compile Unit Block
 0001C0  0300  2203                                      =F'50340355'       Flags
 0001C4  FFFF  FE40                                      =A(CEESTART-PPA2)
 0001C8  0000  0000                                      =F'0'              No PPA4
 0001CC  FFFF  FE40                                      =A(TIMESTMP-PPA2)
 0001D0  0000  0000                                      =F'0'              No primary
 0001D4  0200  0000                                      =F'33554432'       Flags
                           PPA2 End
```

Figure 28. Example of an IPA listing (Part 5 of 6)

```
15694-A01 V1.11 z/OS XL C/C++ IPA                               Partition 1                04/29/2009 04:55:30   Page    11

                                   E X T E R N A L   S Y M B O L   D I C T I O N A R Y

                        TYPE  ID  ADDR    LENGTH           NAME

                         SD    1 000000   0001D8           @STATICP
                         PR    2 000000   000008           @STATIC
                         LD    0 0000B8   000001           main
                         ER    3 000000                    CEESG003
                         ER    4 000000                    PRINTF
                         ER    5 000000                    CEESTART
                         SD    6 000000   000008           @@PPA2
                         SD    7 000000   00000C           CEEMAIN
                         ER    8 000000                    EDCINPL
15694-A01 V1.11 z/OS XL C/C++ IPA                               Partition 1                04/29/2009 04:55:30   Page    12

                             * * * * *   S O U R C E   F I L E   M A P   * * * * *

             OBJECT      SOURCE
  *ORIGIN   FILE ID     FILE ID     SOURCE FILE NAME
      P         2           1      //'USERID1.IPA.SOURCE(HELLO1)'
                                     - Compiled by 15694A01   1B00
                                                on 04/29/2009 04:55:24
      P         3           2      //'USERID1.IPA.SOURCE(HELLO2)'
                                     - Compiled by 15694A01   1B00
                                                on 04/29/2009 04:55:25
      P         4           3      //'USERID1.IPA.SOURCE(HELLO3)'
                                     - Compiled by 15694A01   1B00
                                                on 04/29/2009 04:55:26


  ORIGIN:  P=primary input     PI=primary INCLUDE

                      * * * * *   E N D   O F   S O U R C E   F I L E   M A P   * * * * *
15694-A01 V1.11 z/OS XL C/C++ IPA                               Partition 1                04/29/2009 04:55:30   Page    13

                                    * * * * *   M E S S A G E S   * * * * *

  MESSAGE CODE   PAGE   MESSAGE TEXT

                             * * * * *   E N D   O F   M E S S A G E S   * * * * *
15694-A01 V1.11 z/OS XL C/C++ IPA                               Partition 1                04/29/2009 04:55:30   Page    14

                             * * * * *   M E S S A G E   S U M M A R Y   * * * * *

 TOTAL   UNRECOVERABLE  SEVERE       ERROR     WARNING    INFORMATIONAL
                (U)       (S)         (E)        (W)          (I)
     0          0          0           0          0            0

                      * * * * *   E N D   O F   M E S S A G E   S U M M A R Y   * * * * *

                          * * * * *   E N D   O F   C O M P I L A T I O N   * * * * *
```

| *Figure 28. Example of an IPA listing (Part 6 of 6)*

After a traditional compile, there are three object files, six external functions, and eight external variables. Without a global view of the application, the compiler looks at `hello1.c` and cannot tell that `unused3` is really unused and that `stringU1` is never referenced. So the compiler has to keep all of the code and variables. IPA has the global view so it can remove the unused functions. As you can see from Figure 28 on page 382, only the `main` function remains. The other functions were inlined, and because they were not exported, and their address was not taken, they were removed.

Example 2. all C parts built with XPLINK

The second example is a variation of the first example. The purpose of this example is to show how easy it is to build an application with both XPLINK and IPA. To simplify the options even more, this example will not generate any listings. Please refer to the appropriate sections of “Example 1. all C parts” on page 377 to map the given names to the members of the SCCNSAM data set.

Building example 2. under z/OS UNIX System Services

The only addition to the IPA Compile step is the required addition of the XPLINK option. The GOFF option has also been added (this option defaults on when XPLINK is specified) for convenience purposes.

```
c89 -c -2 -WI,NOOBJECT -Wc,XPLINK,GOFF hello1.c hello2.c hello3.c
```

For the IPA Link step, the changes are similar to the compile step, and the basic changes that must be done to use XPLINK under z/OS UNIX System Services. The option **-WI,XPLINK** is added to guide **c89** to include the XPLINK libraries in the IPA Link step.

```
c89 -2 -WI,LEVEL\(2\) -Wl,XPLINK -o hello hello1.o
  hello2.o hello3.o
```

Building example 2. in batch

In batch, the same basic changes are made. XPLINK and GOFF are added to the IPA Compile steps and the XPLINK proc EDCXI is used instead of EDCI. A few extra includes (CELHS003,CELHS001) are placed in the IPA input to allow IPA to resolve XPLINK library references. This job will result in an object deck that can then be linked to create the module.

```
//USERID1A JOB (127A,0329),'$MEM$',
//  MSGLEVEL=(2,0),MSGCLASS=S,CLASS=A,
//  NOTIFY=USERID1,REGION=1024M
//PROC JCLLIB ORDER=(CBC.SCCNPRC)
//*-------------------------------------------------------------------
//* IPA Compile Step for hello1.c
//*-------------------------------------------------------------------
//C001F336  EXEC EDCC,
//          INFILE='USERID1.IPA.SOURCE(HELLO1)',
//          OUTFILE='USERID1.IPA.OBJECT(HELLOX1),DISP=SHR',
//          CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS  DD *
    IPA(NOOBJECT) RENT LONG OPT(2) XPLINK GOFF
/*
//*-------------------------------------------------------------------
//* IPA Compile Step for hello2.c
//*-------------------------------------------------------------------
//C001F336  EXEC EDCC,
//          INFILE='USERID1.IPA.SOURCE(HELLO2)',
//          OUTFILE='USERID1.IPA.OBJECT(HELLOX2),DISP=SHR',
//          CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS  DD *
    IPA(NOOBJECT) RENT LONG OPT(2) XPLINK GOFF
/*
//*-------------------------------------------------------------------
//* IPA Compile Step for hello3.c
//*-------------------------------------------------------------------
//C001F336  EXEC EDCC,
//          INFILE='USERID1.IPA.SOURCE(HELLO3)',
//          OUTFILE='USERID1.IPA.OBJECT(HELLOX3),DISP=SHR',
//          CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS  DD *
    IPA(NOOBJECT) RENT LONG OPT(2) XPLINK GOFF
/*
//*-------------------------------------------------------------------
//* IPA Link Step for the hello module
//*-------------------------------------------------------------------
//C001F336  EXEC EDCXI,
//          OUTFILE='USERID1.IPALINK.OBJECT(HELLOXP),DISP=SHR',
//          IPARM='OPTFILE(DD:OPTIONS)'
//* The following line sets up an input file that just includes all
//*   the IPA Compile Step object files.
//SYSIN  DD DATA,DLM='/>'
  INCLUDE OBJECT(HELLOX1,HELLOX2,HELLOX3)
  INCLUDE SYSLIB(CELHS003,CELHS001)
/>
//* These are the options used
//OPTIONS  DD DATA,DLM='/>'
    IPA(LINK,LEVEL(2)) OPT(2) RENT LONGNAME
    XPLINK GOFF
/>
//* The following line gives the object library
//OBJECT DD DSN=USERID1.IPA.OBJECT,DISP=SHR
```

Creating a DLL with IPA

This section gives several examples, which describe the aspects of building a simple DLL, as well as how to use some of the advanced IPA features to build a faster DLL. By default, IPA will try to remove unused code and variables (even global variables). In DLL situations, (or with exported variables) this ability becomes limited. For modules with a `main` function, IPA can build a function call tree and determine which functions are or may be called. This list of functions is used to remove unused functions and variables. For DLLs, IPA must treat the list of exported functions as potential entry points, and all exported variables as used. For this reason, the use of the EXPORTALL compiler option is not recommended. IPA provides a control file option that allows you to specify exactly which functions and variables you wish to be exported. This gives the programmer who cannot change the source another way to avoid EXPORTALL. For an example of this, please see "Example 2. using the IPA control file" on page 391.

Example 1. a mixture of C and C++

For this example, the following table shows the mapping of SCCNSAM data set members to given file names. The `main` program is provided to allow you to run the created DLL, it is not used in the following example.

SCCNSAM member name	Name used in this example
CCNGID1	`GlobInfo.h`
CCNGID2	`UserInt.h`
CCNGID3	`UserInterface.C`
CCNGID4	`c_DLL.c`
CCNGID5	`c_DLL.h`
CCNGID6	`cpp_DLL.C`
CCNGID7	`cpp_DLL.h`
CCNGIDM	`main.C`

This example involves the creation of a C/C++ DLL. The DLL is built from one C source file and two C++ source files. For your convenience, a `main` SCCNSAM(CCNGIDM) is provided so that the program can be executed. Instructions to build the `main` will not be given in this example. In general, IPA DLLs are created in the same manner as IPA modules with the extra commands for DLLs added in for the IPA Link step.

Building example 1. under z/OS UNIX System Services

First, IPA must compile each source file. Since NOOBJECT is the default, it is not specifically mentioned in this example. -WI is specified to trigger an IPA Compile.

```
c89 -c -2 -WI -Wc,"FLAG(I),DLL"  c_DLL.c
c++ -c -2 -WI -Wc,"FLAG(I)" -+ cpp_DLL.C
c++ -c -2 -WI -Wc,"FLAG(I),EXPORTALL" -+  UserInterface.C
```

If you are using the xlc utility, the same IPA Compile is invoked by the following command lines:

```
c89 -c -O2 -qipa -qflag=i -qdll          c_DLL.c
c++ -c -O2 -qipa -qflag=i             -+ cpp_DLL.C
c++ -c -O2 -qipa -qflag=i -qexportall -+ UserInterface.C
```

Next, the IPA Link step is performed. In this case, IPA level(1) optimizations are used:

```
c++ -2  -WI,"LEVEL(1)" -Wl,I,DLL -Wl,DLL -o mydll
    UserInterface.o c_DLL.o cpp_DLL.o
```

The LEVEL(1) suboption is fed to IPA. The DLL option is given to the traditional optimizer using `-Wl,I,DLL` and the usual linker command for DLLs is given.

If you are using the xlc utility, the same IPA Link is invoked by the following command line:

```
c++ -O2  -qipa=level=1 -qdll -Wl,DLL -o mydll
  UserInterface.o c_DLL.o cpp_DLL.o
```

Building example 1. under batch

For this example, the following table shows the mapping of SCCNSAM data set members to given PDS member names. The `main` program is provided to allow you to run the created DLL, it is not used in the following example.

SCCNSAM member name	Name used in this example
CCNGID1	IPA.H(GLOBINFO)
CCNGID2	IPA.H(USERINT)
CCNGID3	IPA.SOURCE(USERINT)
CCNGID4	IPA.SOURCE(CDLL)
CCNGID5	IPA.H(C@DLL)
CCNGID6	IPA.SOURCE(CPPDLL)
CCNGID7	IPA.H(CPP@DLL)
CCNGIDM	IPA.SOURCE(MAIN)

```
//USERID1A JOB (127A,0329),'$MEM$',
//  MSGLEVEL=(2,0),MSGCLASS=S,CLASS=A,
//  NOTIFY=USERID1,REGION=1024M
//PROC JCLLIB ORDER=(CBC.SCCNPRC)
//*-------------------------------------------------------------------
//* IPA Compile Step for CDLL
//*-------------------------------------------------------------------
//C001F336  EXEC EDCC,
//          INFILE='USERID1.IPA.SOURCE(CDLL)',
//          OUTFILE='USERID1.IPA.OBJECT(CDLL),DISP=SHR',
//          CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS  DD *
    IPA(NOOBJECT) RENT LONG OPT(2) DLL
    LSEARCH('USERID1.IPA.+')
    SEARCH('CEE.SCEEH.+')
/*
//*-------------------------------------------------------------------
//* IPA Compile Step for CPPDLL
//*-------------------------------------------------------------------
//C001F336  EXEC CBCC,
//          INFILE='USERID1.IPA.SOURCE(CPPDLL)',
//          OUTFILE='USERID1.IPA.OBJECT(CPPDLL),DISP=SHR',
//          CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS  DD *
```

```
    IPA(NOOBJECT) OPT(2)
    LSEARCH('USERID1.IPA.+')
    SEARCH('CEE.SCEEH.+')
/*
//*-------------------------------------------------------------------
//* IPA Compile Step for USERINT
//*-------------------------------------------------------------------
//C001F336  EXEC CBCC,
//          INFILE='USERID1.IPA.SOURCE(USERINT)',
//          OUTFILE='USERID1.IPA.OBJECT(USERINT),DISP=SHR',
//          CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS  DD *
    IPA(NOOBJECT) OPT(2) EXPORTALL
    LSEARCH('USERID1.IPA.+')
    SEARCH('CEE.SCEEH.+')
/*
//*-------------------------------------------------------------------
//* IPA Link Step for the hello module
//*-------------------------------------------------------------------
//C001F336  EXEC CBCI,
//          OUTFILE='USERID1.IPALINK.OBJECT(MYDLL),DISP=SHR',
//          IPARM='OPTFILE(DD:OPTIONS)'
//* The following line sets up an input file that just includes all
//*   the IPA Compile Step object files.
//SYSIN  DD DATA,DLM='/>'
  INCLUDE OBJECT(USERINT,CDLL,CPPDLL)
  INCLUDE SYSLIB(C128,IOSTREAM,COMPLEX)
/>
//* These are the options used
//OPTIONS  DD DATA,DLM='/>'
    IPA(LINK,MAP,LEVEL(1)) OPT(2) RENT LONGNAME
/>
//* The following line gives the object library
//OBJECT DD DSN=USERID1.IPA.OBJECT,DISP=SHR
```

Example 2. using the IPA control file

The following example uses the IPA control file to choose which functions should be exported from UserInterface.C. This allows the IPA Compile step to be done without the EXPORTALL option. The first step is to construct an IPA control file. The function names appearing in the IPA control file must be mangled names if the names in the source file are going to be mangled by the compiler. The file content is as follows:

```
exports=get_user_input__7UIclassFv,
       get_user_sort_method__7UIclassFRi,
       call_user_sort_method__7UIclassFi,
       print_sort_result__7UIclassFv
```

Please refer to the appropriate sections of “Example 1. a mixture of C and C++” on page 389 to map the given names to the members of the SCCNSAM data set.

Building example 2. under z/OS UNIX System Services

First, IPA must compile each source file using the following commands:

```
c89 -c -2 -WI -Wc,"FLAG(I),DLL"  c_DLL.c
c++ -c -2 -WI -Wc,"FLAG(I)" -+ cpp_DLL.C
c++ -c -2 -WI -Wc,"FLAG(I)" -+  UserInterface.C
```

If you are using the xlc utility, the same IPA Compile is invoked by the following command lines:

```
c89 -c -O2 -qipa -qflag=i -qdll          c_DLL.c
c++ -c -O2 -qipa -qflag=i             -+ cpp_DLL.C
c++ -c -O2 -qipa -qflag=i             -+ UserInterface.C
```

Next, the IPA Link step is run to specify a control file:

```
c++ -2  -WI,"LEVEL(1),CONTROL(mydll.cntl)" -Wl,I,DLL -Wl,DLL -o mydll
  UserInterface.o c_DLL.o cpp_DLL.o
```

If you are using the xlc utility, the same IPA Link is invoked by the following command line:

```
c++ -O2 -qipa=level=1 -qipa=control=mydll.cntl -qdll -Wl,DLL -o mydll
    UserInterface.o c_DLL.o cpp_DLL.o
```

This creates a DLL where only the specified functions are exported from `UserInterface.C`.

Building example 2. in batch

```
//USERID1A JOB (127A,0329),'$MEM$',
//  MSGLEVEL=(2,0),MSGCLASS=S,CLASS=A,
//  NOTIFY=USERID1,REGION=1024M
//PROC JCLLIB ORDER=(CBC.SCCNPRC)
//*-------------------------------------------------------------------
//* IPA Compile Step for CDLL
//*-------------------------------------------------------------------
//C001F336  EXEC EDCC,
//          INFILE='USERID1.IPA.SOURCE(CDLL)',
//          OUTFILE='USERID1.IPA.OBJECT(CDLL),DISP=SHR',
//          CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS  DD *
    IPA(NOOBJECT) RENT LONG OPT(2) DLL
    LSEARCH('USERID1.IPA.+')
    SEARCH('CEE.SCEEH.+')
/*
//*-------------------------------------------------------------------
//* IPA Compile Step for CPPDLL
//*-------------------------------------------------------------------
//C001F336  EXEC CBCC,
//          INFILE='USERID1.IPA.SOURCE(CPPDLL)',
//          OUTFILE='USERID1.IPA.OBJECT(CPPDLL),DISP=SHR',
//          CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS  DD *
    IPA(NOOBJECT) OPT(2)
    LSEARCH('USERID1.IPA.+')
    SEARCH('CEE.SCEEH.+')
/*
//*-------------------------------------------------------------------
//* IPA Compile Step for USERINT
//*-------------------------------------------------------------------
 //C001F336  EXEC CBCC,
 //          INFILE='USERID1.IPA.SOURCE(USERINT)',
 //          OUTFILE='USERID1.IPA.OBJECT(USERINT),DISP=SHR',
 //          CPARM='OPTFILE(DD:OPTIONS)'
 //OPTIONS  DD *
     IPA(NOOBJECT) OPT(2)
     LSEARCH('USERID1.IPA.+')
     SEARCH('CEE.SCEEH.+')
 /*
 //*-------------------------------------------------------------------
 //* IPA Link Step for the hello module
 //*-------------------------------------------------------------------
 //C001F336  EXEC CBCI,
 //          OUTFILE='USERID1.IPALINK.OBJECT(MYDLL),DISP=SHR',
 //          IPARM='OPTFILE(DD:OPTIONS)'
 //* The following line sets up an input file that just includes all
 //*   the IPA Compile Step object files.
 //SYSIN  DD DATA,DLM='/>'
   INCLUDE OBJECT(USERINT,CDLL,CPPDLL)
   INCLUDE SYSLIB(C128,IOSTREAM,COMPLEX)
 />
```

```
//* These are the options used
//OPTIONS  DD DATA,DLM='/>'
  IPA(LINK,LEVEL(1),CONTROL('USERID1.ipa.cntl(dllex)'))
  OPT(2) RENT LONGNAME
/>
//* The following line gives the object library
//OBJECT DD DSN=USERID1.IPA.OBJECT,DISP=SHR
```

In the resultant object deck (MYDLL), only functions that are explicitly exported using **#pragma export** and the four functions given in the control file are exported.

Using Profile-Directed Feedback (PDF)

In any large application, there are sections of code that are not often executed, such as code for error-handling. A traditional compiler cannot tell what these low frequency sections of code or functions are, and may spend a lot of time optimizing code that will never be executed. *Profile-Directed Feedback* (PDF) can be used to collect information about the way the program is really used and the compiler can use this information when optimizing the code. PDF also enables you to receive estimates on how many times loops are iterated.

Steps for utilizing PDF optimization

Perform the following four steps to utilize the PDF optimization:

1. Compile some or all of the source files in a program with the IPA PDF1 suboption. The OPTIMIZE(2) option, or preferably the OPTIMIZE(3) option, as well as the IPA(LEVEL(1|2)) option should be specified for optimization. Special attention should be paid to the compiler options that are used to compile the files because the same options (other than IPA(PDF1)) must be used later.

   In a large application, the use of the PDF1 suboption should be concentrated on those areas of the code that can benefit most from optimization. You do not need to compile all of the code for your application with the PDF1 suboption, but you do need to compile the `main` function with the PDF1 suboption.

2. If you are using an MVS data set for your PDF file, preallocate the PDF data set using `RECFM = U` and `LRECL = 0`.
3. Run the program build from step 1 with typical input data. The program records profiling information when it finishes. The program can be run multiple times with different input data sets, and the profiling information is accumulated to provide a count of how often branches are taken and blocks of code are executed, based on the input data sets used. It is critically important that the data used is representative of the data that will be used during a normal run of the finished program.

4. Re-build your program using the identical set of source files with the identical compiler options that you used in step 1, but change the PDF1 suboption to PDF2. In this second stage, the accumulated profiling information is used to fine-tune the optimizations. The resulting program contains no profiling overhead.

Specifically, the following JCL can be used to perform a PDF1 compile of the hello world program (see “Example 1. all C parts” on page 377).

```
//USERID1A JOB (127A,0329),'$MEM$',
//  MSGLEVEL=(2,0),MSGCLASS=S,CLASS=A,
//  NOTIFY=USERID1,REGION=1024M
//PROC JCLLIB ORDER=(CBC.SCCNPRC)
//*-------------------------------------------------------------------
//* IPA Compile Step for hello1.c
//*-------------------------------------------------------------------
//C001F336  EXEC EDCC,
//          INFILE='USERID1.IPA.SOURCE(HELLO1)',
//          OUTFILE='USERID1.IPA.OBJECT(HELLOX1),DISP=SHR',
//          CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS  DD *
    IPA(NOOBJECT,PDF1) RENT LONG OPT(2) XPLINK GOFF
/*
//*-------------------------------------------------------------------
//* IPA Compile Step for hello2.c
//*-------------------------------------------------------------------
//C001F336  EXEC EDCC,
//          INFILE='USERID1.IPA.SOURCE(HELLO2)',
//          OUTFILE='USERID1.IPA.OBJECT(HELLOX2),DISP=SHR',
//          CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS  DD *
    IPA(NOOBJECT,PDF1) RENT LONG OPT(2) XPLINK GOFF
/*
//*-------------------------------------------------------------------
//* IPA Compile Step for hello3.c
//*-------------------------------------------------------------------
//C001F336  EXEC EDCC,
//          INFILE='USERID1.IPA.SOURCE(HELLO3)',
//          OUTFILE='USERID1.IPA.OBJECT(HELLOX3),DISP=SHR',
//          CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS  DD *
    IPA(NOOBJECT,LIST,PDF1) RENT LONG OPT(2) XPLINK GOFF
/*
//*-------------------------------------------------------------------
//* IPA Link Step for the hello module
//*-------------------------------------------------------------------
//C001F336  EXEC EDCXI,
//          OUTFILE='USERID1.IPALINK.OBJECT(HELLOXP),DISP=SHR',
//          IPARM='OPTFILE(DD:OPTIONS)'
//* The following line sets up an input file that just includes all
//*   the IPA Compile Step object files.
//SYSIN  DD DATA,DLM='/>'
  INCLUDE OBJECT(HELLOX1,HELLOX2,HELLOX3)
  INCLUDE SYSLIB(CELHS003,CELHS001)
/>
//* These are the options used
//OPTIONS  DD DATA,DLM='/>'
    IPA(LINK,LEVEL(2),MAP,PDF1,PDFNAME(//'USERID1.MY.PDF'))
    OPT(2) RENT LONGNAME LIST
    XPLINK GOFF
/>
//* The following line gives the object library
//OBJECT DD DSN=USERID1.IPA.OBJECT,DISP=SHR
//*-------------------------------------------------------------------
//* LINK the hello module
//*-------------------------------------------------------------------
//C001F336  EXEC CCNXPD1B,
//          INFILE='USERID1.IPALINK.OBJECT(HELLOXP)',
//          OUTFILE='USERID1.DEV.LOAD1(HELLOXP),DISP=SHR'
```

Note: The PDF1 option is specified on each of the IPA compiles, and the PDFNAME suboption is specified on the IPA Link step. This PDFNAME suboption gives the name of the file where the statistics about the program will be stored, this file is referred to as the *PDF file*. While it is not strictly required to preallocate the PDF file, when using a PS or PDS file, the data set may be required to preallocate to ensure the file is large enough. If the PDF file is preallocated, it should be allocated with an LRECL of 0 and a RECFM of U.

Finally, instead of using a traditional link proc, the link of the PDF1 code should be done with the CCNPD1B proc (for non-XPLINK code), the CCNXPD1B proc (for XPLINK code), or the CCNQPD1B proc for 64-bit code and linking. These procs provide all the libraries necessary to allow the object file created by the IPA Link step to be linked with the PDF run-time function that stores the statistical information.

A PDF2 IPA compile job looks very similar to a PDF1 IPA compile job except that the:

- PDF2 suboption replaces PDF1
- Traditional EDCB proc can be used to bind the object created during the IPA Link step into a module

Steps for building a module in z/OS UNIX System Services using PDF

Perform the following steps in z/OS UNIX System Services to build a module using the PDF process:

1. Build the PDF1 module using the following commands:

   ```
   c89 -c -2 -WI,PDF1 hello1.c hello2.c hello3.c
   c89 -2 -Wl,PDF1 -WI,PDF1,PDFNAME\(./hello.pdf\),LEVEL\(2\) -o
     hello hello1.o hello2.o hello3.o
   ```

   If you are using the xlc utility, the command line syntax is:

   ```
   c89 -c -O2 -qipa=pdf1 hello1.c hello2.c hello3.c
   c89 -O2 -qipa=pdf1 -qipa=level=2 -qipa=pdfname=./hello.pdf -o
       hello hello1.o hello2.o hello3.o
   ```

2. Run the module, to create `hello.pdf`:

   ```
   hello
   ```

3. Rebuild the module using the information in `hello.pdf` using the following commands:

   ```
   c89 -c -2 -WI,PDF2 hello1.c hello2.c hello3.c
   c89 -2 -WI,PDF2,PDFNAME\(./hello.pdf\),LEVEL\(2\) -o hello hello1.o
     hello2.o hello3.o
   ```

   If you are using the xlc utility, the command line syntax is:

   ```
   c89 -c -O2 -qipa=pdf2 hello1.c hello2.c hello3.c
   c89 -O2 -qipa=pdf2 -qipa=level=2 -qipa=pdfname=./hello.pdf -o
       hello hello1.o hello2.o hello3.o
   ```

Reference Information

The following topic provides reference information concerning the IPA Link step control file, and object file directives understood by IPA.

IPA Link step control file

The IPA Link step control file is a fixed-length or variable-length format file that contains additional IPA processing directives. The CONTROL suboption of the IPA compiler option identifies this file.

The IPA Link step issues an error message if any of the following conditions exist in the control file:

- The control file directives have invalid syntax.
- There are no entries in the control file.
- Duplicate names exist in the control file.

You can specify the following directives in the control file.

csect=*csect_names_prefix*

Supplies information that the IPA Link step uses to name the CSECTs for each partition that it creates. The *csect_names_prefix* parameter is a comma-separated list of tokens that is used to construct CSECT names.

The behavior of the IPA Link steps varies depending upon whether you specify the CSECT option with a qualifier.

- If you do not specify the CSECT option with a qualifier, the IPA Link step does the following:
  - Truncates each name prefix or pads it at the end with @ symbols, if necessary, to create a 7 character token
  - Uppercases the token
  - Adds a suffix to specify the type of CSECT, as follows:

C	code
S	static data
T	test

- If you specify the CSECT option with a non-null qualifier, the IPA Link step does the following:
  - Uppercases the token
  - Adds a suffix to specify the type of CSECT, as follows where *qualifier* is the qualifier you specified for CSECT and *nameprefix* is the name you specified in the IPA Link Step Control File:

*qualifier***#***nameprefix***#C**	code
*qualifier***#***nameprefix***#S**	static data
*qualifier***#***nameprefix***#T**	test

- If you specify the CSECT option with a null qualifier, the IPA Link step does the following:
  - Uppercases the token
  - Adds a suffix to specify the type of CSECT, as follows where *nameprefix* is the name you specified in the IPA Link Step Control File:

*nameprefix***#C**	code
*nameprefix***#S**	static data
*nameprefix***#T**	test

The IPA Link step issues an error message if you specify the CSECT option but no control file, or did not specify any `csect` directives in the control file. In this situation, IPA generates a CSECT name and an error message for each partition.

The IPA Link step issues a warning or error message (depending upon the presence of the CSECT option) if you specify CSECT name prefixes, but the number of entries in the `csect_names` list is fewer than the number of partitions that IPA generated. In this situation, for each unnamed partition, the IPA Link step generates a

CSECT name prefix with format `@CSnnnn`, where `nnnn` is the partition number. If you specify the CSECT option, the IPA Link step also generates an error message for each unnamed partition. Otherwise, the IPA Link step generates a warning message for each unnamed partition.

noexports
Removes the ″export″ flag from all symbols (functions and variables) in IPA and non-IPA input files.

| **exports=***name[,name]*
Specifies a list of symbols (functions and variables) to export by setting the symbol ″export″ flag. Note: Only symbols defined within IPA objects can be exported using this directive.

inline=*name[,name]*
Specifies a list of functions that are desirable for the compiler to inline. The functions may or may not be inlined.

inline=*name[,name]* **from** *name[,name]*
Specifies a list of functions that are desirable for the compiler to inline, if the functions are called from a particular function or list of functions. The functions may or may not be inlined.

noinline=*name[,name]*
Specifies a list of functions that the compiler will not inline.

noinline=*name[,name]* **from** *name[,name]*
Specifies a list of functions that the compiler will not inline, if the functions are called from a particular function or list of functions.

exits=*name[,name]*
Specifies names of functions that represent program exits. Program exits are calls that can never return, and can never call any procedure that was compiled with the IPA Compile step.

lowfreq=*name[,name]*
Specifies names of functions that are expected to be called infrequently. These functions are typically error handling or trace functions.

partition=small|*medium***|large|unsigned-integer**
Specifies the size of each program partition that the IPA Link step creates. When partition sizes are large, it usually takes longer to complete the code generation, but the quality of the generated code is usually better.

For a finer degree of control, you can use an *unsigned-integer* value to specify the partition size. The integer is in ACUs (Abstract Code Units), and its meaning may change between releases. You should only use this integer for very short term tuning efforts, or when the number of partitions (and therefore the number of CSECTs in the output object module) must remain constant.

The size of a CSECT cannot exceed 16 MB with the XOBJ format. Large CSECTs require the GOFF option.

The default for this directive is `medium`.

partitionlist=*partition_number***[,***partition_number***]**
Used to reduce the size of an IPA Link listing. If the IPA Link control file contains this directive and the LIST option is active, a pseudo-assembly listing is generated for only these partitions.

partition_number is a decimal number representing an `unsigned int`.

safe=*name[,name]*
Specifies a list of *safe functions* that are not compiled as IPA objects. These are functions that do not call a visible (not missing) function either through a direct call or a function pointer. Safe functions can modify global variables, but may not call functions that are not compiled as IPA objects.

isolated=*name[,name]*
Specifies a list of *isolated functions* that are not compiled as IPA objects. Neither isolated functions nor functions within their call chain can refer to global variables. IPA assumes that functions that are bound from shared libraries are isolated.

pure=*name[,name]*
Specifies a list of *pure functions* that are not compiled as IPA objects. These are functions that are safe and isolated and do not indirectly alter storage accessible to visible functions. A pure function has no observable internal state nor has side-effects, defined as potentially altering any data visible to the caller. This means that the returned value for a given invocation of a function is independent of any previous or future invocation of the function.

unknown=*name[,name]*
Specifies a list of *unknown functions* that are not compiled as IPA objects. These are functions that are not safe, isolated, or pure. This is the default for all functions defined within non-IPA objects. Any function specified as unknown can make calls to other parts of the program compiled as IPA objects and modify global variables and dummy arguments. This option greatly restricts the amount of interprocedural optimization for calls to unknown functions.

missing=*attribute*

Specifies the characteristics of *missing functions*. There are two types of missing functions:

- Functions dynamically linked from another DLL (defined using an IPA Link IMPORT control statement)
- Functions that are statically available but not compiled with the IPA option

IPA has no visibility to the code within these functions. You must ensure that all user references are resolved at IPA Link time with user libraries or run-time libraries.

The default setting for this directive is `unknown`. This instructs IPA to make pessimistic assumptions about the data that may be used and modified through a call to such a missing function, and about the functions that may be called indirectly through it.

You can specify the following attributes for this directive:

safe	Specifies that the missing functions are safe. See the description for the `safe` directive in this topic.
isolated	Specifies that the missing functions are isolated. See the description for the `isolated` directive in this topic.

pure
: Specifies that the missing functions are pure. See the description for the `pure` directive in this topic.

unknown
: Specifies that the missing functions are unknown. See the description for the `unknown` directive in this topic. This is the default attribute.

retain=*symbol-list*
: Specifies a list of exported functions or variables that the IPA Link step retains in the final object module. The IPA Link step does not prune these functions or variables during optimization.

Note: In the listed directives, *name* can be a regular expression. Thus, *name* can match multiple symbols in your application through pattern matching. The regular expression syntax supported by the IPA control file processor is as follows:

Table 41. Syntax rules for specifying regular expressions

Expression	Description
string	Matches any of the characters specified in *string*. For example, `test` will match `testimony`, `latest`, and `intestine`.
^string	Matches the pattern specified by *string* only if it occurs at the beginning of a line.
string$	Matches the pattern specified by *string* only if it occurs at the end of a line.
str.ing	The period (.) matches any single character. For example, `t.st` will match `test`, `tast`, `tZst`, and `t1st`.
string\special_char	The backslash (\ ) can be used to escape special characters. For example, assume that you want to find lines ending with a period. Simply specifying the expression `.$` would show all lines that had at least one character of any kind in it. Specifying `\.$` escapes the period (.), and treats it as an ordinary character for matching purposes.
[*string*]	Matches any of the characters specified in *string*. For example, `t[a-g123]st` matches tast and test, but not `t-st` or `tAst`.
[^*string*]	Does not match any of the characters specified in *string*. For example, `t[^a-zA-Z]` st matches `t1st`, `t-st`, and `t,st` but not `test` or `tYst`.
*string**	Matches zero or more occurrences of the pattern specified by *string*. For example, `te*st` will match `tst`, `test`, and `teeeeeest`.
string+	Matches one or more occurrences of the pattern specified by *string*. For example, `t(es)+t` matches `test`, `tesest`, but not `tt`.
string?	Matches zero or one occurrences of the pattern specified by *string*. For example, `te?st` matches either `tst` or `test`.
string{*m,n*}	Matches between *m* and *n* occurrence(s) of the pattern specified by *string*. For example, `a{2}` matches `aa`, and `b{1,4}` matches `b`, `bb`, `bbb`, and `bbbb`.
string1 | *string2*	Matches the pattern specified by either *string1* or *string2*. For example, `s | o` matches both characters `s` and `o`.

Object file directives understood by IPA

IPA recognizes and acts on the following binder object control directives:

- INCLUDE
- LIBRARY

- IMPORT

Some other linkage control statements (such as NAME, RENAME and ALIAS) are accepted and passed through to the linker.

Troubleshooting

It is strongly recommended that you resolve all warnings that occur during the IPA Link step. Resolution of these warnings often removes seemingly unrelated problems.

The following list provides frequently asked questions (Q) and their respective answers (A):

- Q - I am running out of memory while using IPA. Are there any options for reducing its use of memory and increasing the system-defined limits?

  A - IPA reacts to the NOMEMORY option, and the code generator will react to the MAXMEM option. If this does not give you sufficient memory, consider running IPA from batch where more memory can be accessed. Before switching to batch, verify with your system programmer that you have access to the maximum possible memory (both in batch and in z/OS UNIX System Services). See “Steps for diagnosing errors that occur at IPA Link time” on page 651 for more information on setting the MEMLIMIT and the REGION system parameters. You could also reduce the level of IPA processing via the IPA LEVEL suboption.
- Q - I am receiving a ″partition too large″ warning. How do I fix it?

  A - Use the IPA Control file to specify a different partition size.
- Q - My IPA Compile time is too long. Are there any options?

  A - Using a lower IPA compilation level (0 or 1 instead of 2) will reduce the compile time. To minimize the compile time, ensure you are using the IPA(NOOBJECT) option for your IPA compiles. A smaller partition size, specified in the control file, may minimize the amount of time spent in the code generator. Limiting inlining, may improve your compile time, but it will decrease your performance gain significantly and should only be done selectively using the IPA control file. Use the IPA control file to specify little used functions as low frequency so that IPA does not spend too much time trying to optimize them.
- Q - Can I tune the IPA automatic inlining like I can for the regular inliner?

  A - Yes. Use the INLINE option for the IPA Link step.
- Q - I am using IPA(PDF1) and my program will not bind. What do I do?

  A - Under z/OS UNIX System Services, specify **-Wl,PDF1** when linking with c89 or C++. Under MVS batch, use the CCNPD1B, CCNXPD1B, or CCNQPD1B PROCs. For further information on these PROCs, see Chapter 12, “Cataloged procedures and REXX EXECs,” on page 465.

Chapter 9. Binding z/OS XL C/C++ programs

This information describes how to bind your programs using the program management binder in the z/OS batch, z/OS UNIX System Services, and TSO environments.

When you can use the binder

The output of the binder is a program object. You can store program objects in a PDSE member or in a z/OS UNIX System Services file. Depending on the environment you use, you can produce binder program objects as follows:

- For c89:

  If the targets of your executables are z/OS UNIX files, you can use the binder. If the targets of your executables are PDSs, you must use the prelinker, followed by the binder. If the targets of your executables are PDSEs, you can use the binder alone.
- For z/OS batch or TSO:

  If you can use PDSEs, you can use the binder. If you want to to use PDSs, you must use the prelinker for the following:
  - C++ code
  - C code compiled with the LONGNAME, RENT, or DLL options
- For GOFF and XPLINK:

  If you have compiled your program with the GOFF, XPLINK, or LP64 compiler options, you must use the binder.

For more information on the prelinker, see Appendix A, “Prelinking and linking z/OS XL C/C++ programs,” on page 601.

When you cannot use the binder

The following restrictions apply when you are using the binder to produce a program object.

Your output is a PDS, not a PDSE

If you are using z/OS batch or TSO, and your output must target a PDS instead of a PDSE, you cannot use the binder.

CICS

Prior to CICS Transaction Server for z/OS V3R1, PDSEs are not supported. From CICS Transaction Server for z/OS V3R1 onwards, there is support in CICS for PDSEs. Please refer to *CICS Transaction Server for z/OS Release Guide*, where there are several references to PDSEs, and a list of prerequisite APAR fixes.

MTF

MTF does not support PDSEs. If you have to target MTF, you cannot use the binder.

IPA

Object files that are generated by the IPA Compile step using the IPA(NOLINK,OBJECT) compiler option may be given as input to the binder. Such an object file is a combination of an IPA object module, and a regular compiler object module. The binder processes the regular compiler object module, ignores the IPA object module, and no IPA optimization is done.

Object files that are generated by the IPA Compile step using the IPA(NOLINK,NOOBJECT) compiler option should not be given as input to the binder. These are IPA-only object files, and do not contain a regular compiler object module.

The IPA Link step will not accept a program object as input. IPA Link can process load module (PDS) input files, but not program object (PDSE) input files.

Using different methods to bind

This topic shows you how to use the following different methods to bind your application:

Single final bind
Compile all your code and then perform a single final bind of all the object modules.

Bind each compile unit
Compile and bind each compilation unit, then perform a final bind of all the partially bound program objects.

Build and use a DLL
Build DLLs and programs that use those DLLs.

Rebind a changed compile unit
Recompile only changed compile units, and rebind them into a program object without needing other unchanged compile units.

Single final bind

You can use the method that is shown in Figure 29 on page 403 to build your application executable for the first time. With this method, you compile each source code unit separately, then bind all of the resultant object modules together to produce an executable program object.

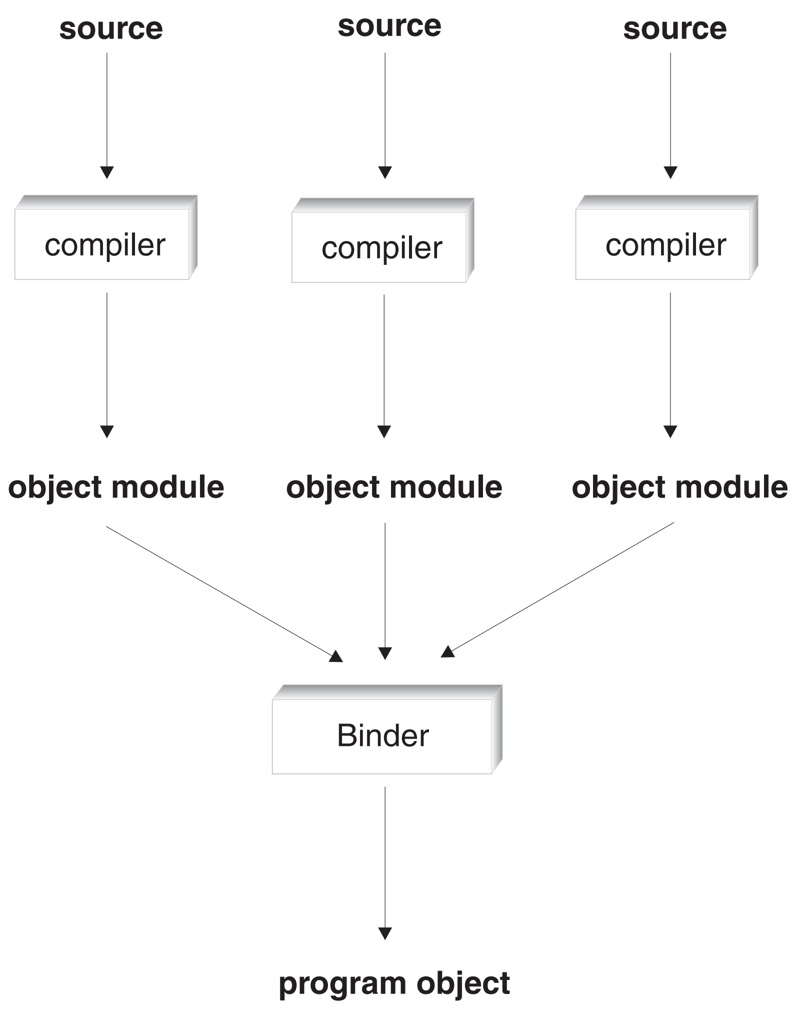


Figure 29. Single final bind

Bind each compile unit

If you have changed the source in a compile unit, you can use the method that is shown in Figure 30 on page 404. With this method, you compile and bind your changed compile unit into an intermediate program object, which may have unresolved references. Then you bind all your program objects together to produce a single executable program object.

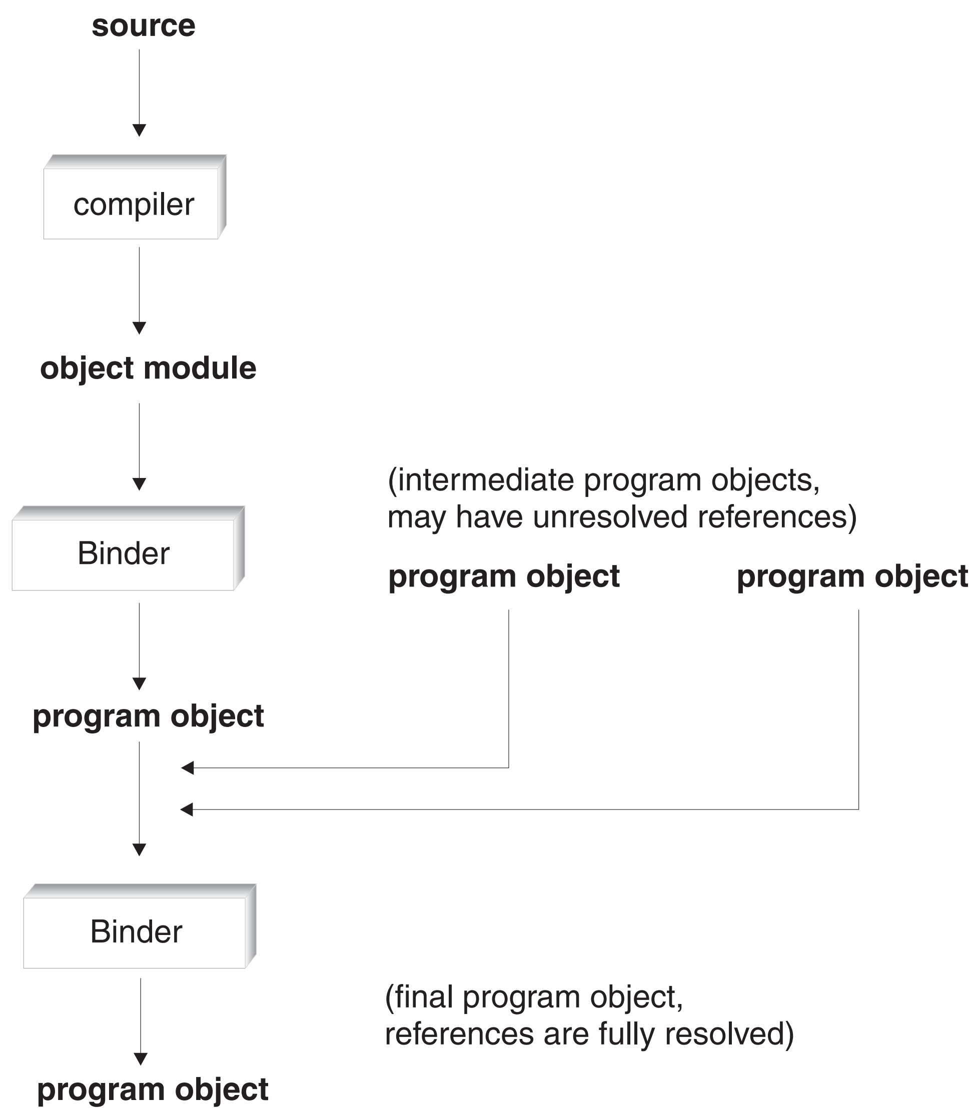

source
compiler
object module
Binder
(intermediate program objects, may have unresolved references)
program object
program object
program object
Binder
(final program object, references are fully resolved)
program object

Figure 30. Bind each compile unit

Build and use a DLL

You can use the method that is shown in Figure 31 on page 405 to build a DLL. To build a DLL, the code that you compile must contain symbols which indicate that they are exported. You can use the compiler option EXPORTALL or the **#pragma export** directive to indicate symbols in your C or C++ code that are to be exported. For C++, you can also use the `_Export` keyword.

When you build the DLL, the bind step generates a DLL and a file of IMPORT control statements which lists the exported symbols. This file is known as a definition side-deck. The binder writes one IMPORT control statement for each exported symbol. The file that contains IMPORT control statements indicates symbol names which may be imported and the name of the DLL from which they are imported.

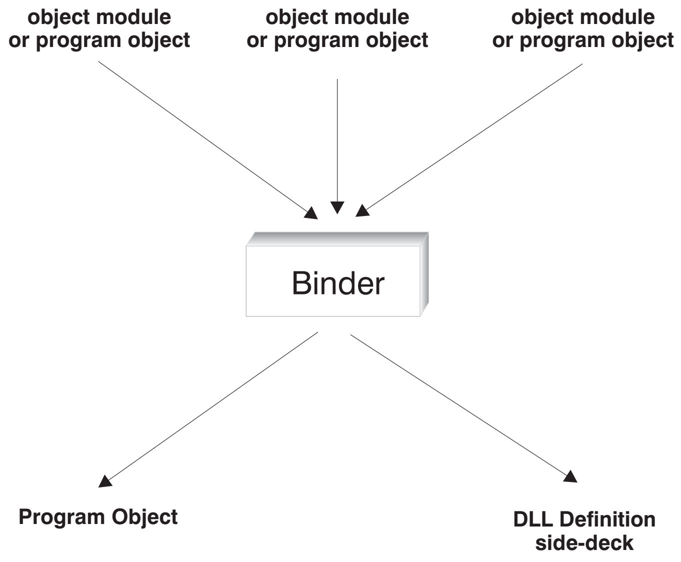

object module or program object
object module or program object
object module or program object
Binder
Program Object
DLL Definition side-deck

Figure 31. Build a DLL

You can use the method that is shown in Figure 32 to build an application that uses a DLL. To build a program which dynamically links symbols from a DLL during application run time, you must have C++ code, or C code that is compiled with the DLL option. This allows you to import symbols from a DLL. You must have an IMPORT control statement for each symbol that is to be imported from a DLL. The IMPORT control statement controls which DLL will be used to resolve an imported function or variable reference during execution. The bind step of the program that imports symbols from the DLL must include the definition side-deck of IMPORT control statements that the DLLs build generated.

The binder does not take an incremental approach to the resolution of DLL-linkage symbols. When binding or rebinding a program that uses a DLL, you must always specify the DYNAM(DLL) option, and must provide all IMPORT control statements. The binder does not retain these control statements for subsequent binds.

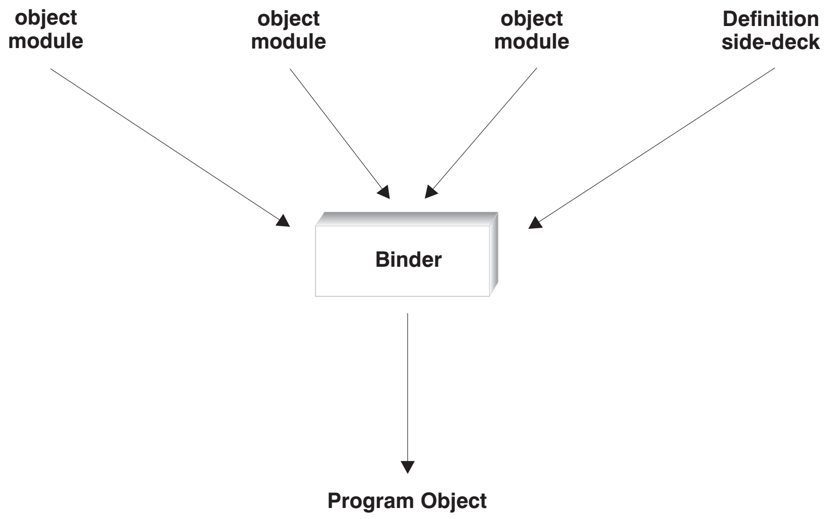


Figure 32. Build an application that uses a DLL

Rebind a changed compile unit

You can use the method shown in Figure 33 to rebind an application after making changes to a single compile unit. Compile your changed source file and then rebind the resultant object module with the complete program object of your application. This will replace the binder sections that are associated with the changed compile unit in the program.

You can use this method to maintain your application. For example, you can change a source file and produce a corresponding object module. You can then ship the object module to your customer, who can bind the new object module with the complete program object for the application. If you use this method, you have fewer files to maintain: just the program object for the application and your source code.

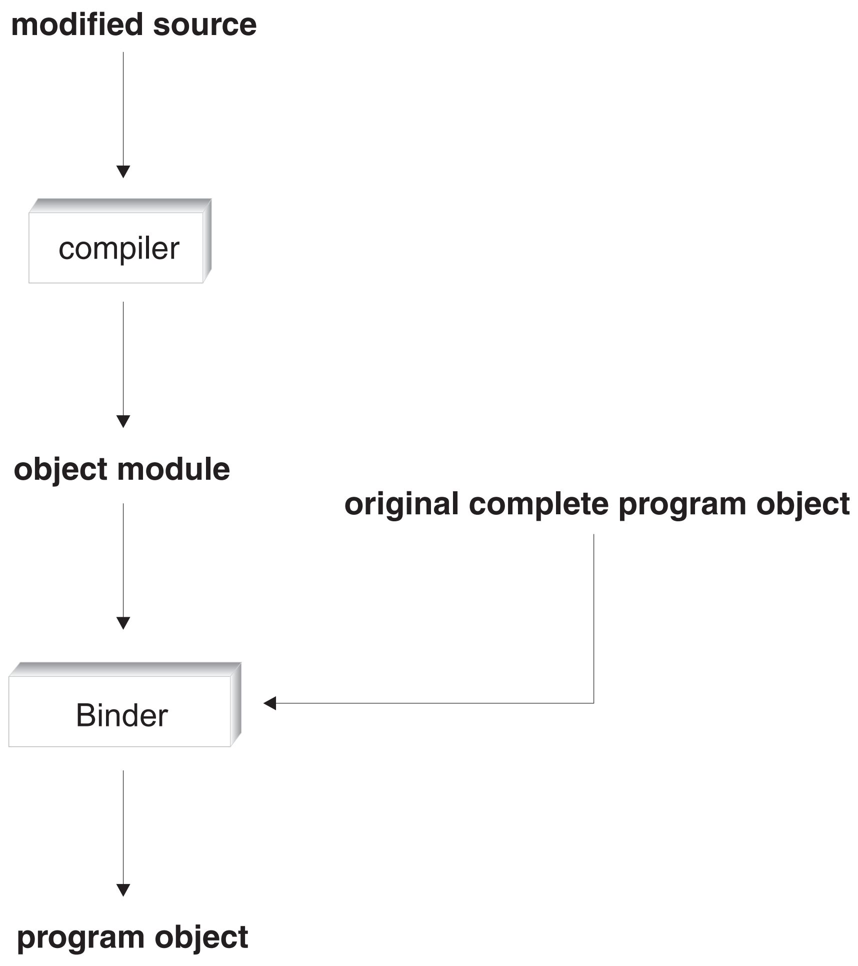


Figure 33. Rebinding a changed compile unit

Binding under z/OS UNIX System Services

The c89 and xlc utilities are the interface to the compiler and the binder for z/OS UNIX System Services C/C++ applications. You can use all supported command names, for example, **c89**, **c++**, and **xlc**, to compile, to compile and bind a program in one step, or to bind application object modules after compilation.

The default, for these utilities, is to invoke the binder alone, without first invoking the prelinker. That is, since the OS/390 V2R4 Language Environment release and

DFSMS 1.4, if the output file (**-o** executable) is not a PDS member, then the binder will be invoked. To modify your environment to run the prelinker, refer to the description of the *prefix*_STEPS environment variable in "Environment variables" on page 550.

Typically, you invoke the c89 and c++ utilities from the z/OS shell. For more information on these utilities, see Chapter 21, "c89 — Compiler invocation using host environment variables," on page 535 or the *z/OS UNIX System Services Command Reference*.

To bind your XPLINK module, specify **-Wl,xplink** on the c89/c++ command.

z/OS UNIX System Services example

The example source files unit0.c, unit1.c, and unit2.c that are shown in Figure 34, are used to illustrate all of the z/OS UNIX System Services examples that follow.

```
/* file: unit0.c */
#include <stdio.h>
extern int f1(void);
extern int f4(void);
int main(void) {
  int rc1;
  int rc4;
  rc1 = f1();
  rc4 = f4();
  if (rc1 != 1) printf("fail rc1 is %d\n",rc1);
  if (rc4 != 40) printf("fail rc4 is %d\n",rc4);
  return 0;
}

/* file: unit1.c */
int f1(void) { return 1; }

/* file: unit2.c */
int f2(void) { return 20;}
int f3(void) { return 30;}
int f4(void) { return f2()*2; /* 40 */ }
```

Figure 34. Example source files

Steps for single final bind using c89

Before you begin: Compile each source file and then perform a single final bind.

Perform the following steps to perform a single final bind using **c89**:

1. Compile each source file to generate the object modules unit0.o, unit1.o, and unit2.o as follows:

   ```
   c89 -c -W c,"CSECT(myprog)"  unit0.c
   c89 -c -W c,"CSECT(myprog)"  unit1.c
   c89 -c -W c,"CSECT(myprog)"  unit2.c
   ```

2. Perform a final single bind to produce the executable program myprog. Use the c89 utility as follows:

   ```
   c89 -o myprog unit0.o unit1.o unit2.o
   ```

The **-o** option of the c89 command specifies the name of the output executable. The c89 utility recognizes from the file extension `.o` that `unit0.o`, `unit1.o` and `unit2.o` are not to be compiled but are to be included in the bind step.

Example: The following is an example of a makefile to perform a similar build:

```
PGM = myprog
SRCS = unit0.c unit1.c unit2.c
OBJS = $(SRCS:b:+".o")
COPTS = -W c,"CSECT(myprog)"
$(PGM) : ($OBJS)
        c89 -o $(PGM) $(OBJS)
%.o : %.c
        c89 -c -o $@ $(COPTS) $<
```

For more information on makefiles, see *z/OS UNIX System Services Programming Tools*.

Advantage

This method is simple, and is consistent with existing methods of building applications, such as makefiles.

Steps for binding each compile unit using c89

Before you begin: Compile each source file and also bind it.

Perform the following steps to complete a final bind of all the partially bound units:

1. Compile each source file to its object module (.tmp). Bind each object module into a partially bound program object (.o), which may have unresolved references. In this example, references to `f1()` and `f4()` in `unit0.o` are unresolved. When the partially bound programs are created, remove the object modules as they are no longer needed. Use **c89** to compile each source file, as follows:

   ```
   c89 -c -W c,"CSECT(myprog)" -o unit0.tmp unit0.c
   c89 -r -o unit0.o unit0.tmp
   rm unit0.tmp

   c89 -c -W c,"CSECT(myprog)" -o unit1.tmp unit1.c
   c89 -r -o unit1.o unit1.tmp
   rm unit1.tmp

   c89 -c -W c,"CSECT(myprog)" -o unit2.tmp unit2.c
   c89 -r -o unit2.o unit2.tmp
   rm unit2.tmp
   ```

   The **-r** option supports rebindability by disabling autocall processing.

---

2. Perform the final single bind to produce the executable program `myprog` by using **c89**:

   ```
   c89 -o myprog unit0.o unit1.o unit2.o
   ```

Example: The following is an example of a makefile for performing a similar build:

```
_C89_EXTRA_ARGS=1
.EXPORT : _C89_EXTRA_ARGS       1
PGM = myprog                           2
SRCS = unit0.c unit1.c unit2.c         3
OBJS = $(SRCS:b:+".o")                 4
COPTS = -W c,"CSECT(myprog)"
$(PGM) : $(OBJS)                       5
        c89 -o $(PGM) $(OBJS)
%.tmp : %.c        6
        c89 -c -o $@ $(COPTS) $<
%.o : %.tmp                            7
        c89 -r -o $@ $<
```

1 Export the environment variable _C89_EXTRA_ARGS so c89 will process files with non-standard extensions. Otherwise c89 will not recognize `unit0.tmp`, and the makefile will fail

2 name of executable

3 list of source files

4 list of partly bound parts

5 executable depends on parts

6 make `.tmp` file from `.c`

7 make `.o` from `.tmp`

In this example, **make** automatically removes the intermediate .tmp files after the makefile completes, since they are not marked as PRECIOUS. For more information on makefiles, see *z/OS UNIX System Services Programming Tools*.

Advantage

Binding a set of partially bound program objects into a fully bound program object is faster than binding object modules into a fully bound program object for NOGOFF objects. For example, a central build group can create the partially bound program objects. Developers can then use these program objects and their changed object modules to create a development program object.

Steps for building and using a DLL using c89

Before you begin: Build `unit1.c` and `unit2.c` into DLL `onetwo`, which exports functions `f1()`, `f2()`, `f3()`, and `f4()`. Then build `unit0.c` into a program which dynamically links to functions `f1()` and `f4()` defined in the DLL.

Perform the following steps to build and use a DLL using **c89**:

1. Compile `unit1.c` and `unit2.c` to generate the object modules `unit1.o` and `unit2.o` which have functions to be exported. Use the c89 utility as follows:
   ```
   c89 -c -W c,"EXPORTALL,CSECT(myprog)" unit1.c
   c89 -c -W c,"EXPORTALL,CSECT(myprog)" unit2.c
   ```

2. Bind `unit1.o` and `unit2.o` to generate the DLL `onetwo`:
   ```
   c89 -Wl,dll -o onetwo unit1.o unit2.o
   ```
   When you bind code with exported symbols, you should specify the DLL binder option (**-W l,dll**).

   In addition to the DLL `onetwo` being generated, the binder writes a list of IMPORT control statements to `onetwo.x`. This list is known as the definition side-deck. One IMPORT control statement is written for each exported symbol.

These generated control statements will be included later as input to the bind step of an application that uses this DLL, so that it can import the symbols.

3. Compile `unit0.c` with the DLL option **-W c,DLL**, so that it can import unresolved symbols. Bind the object module, with the definition side-deck `onetwo.x` from the DLL build:

```
c89 -c -W c,DLL unit0.c
c89 -o dll12usr unit0.o onetwo.x
```

Advantage

The bind-time advantage of using DLLs is that you only need to rebuild the DLL with the changed code in it. You do not need to rebuild all applications that use the DLL in order to use the changed code.

Steps for rebinding a changed compile unit using c89

Before you begin: Rebuild an application after making a change to a single source file.

Perform the following steps to rebind a changed compile unit using **c89**:

1. Recompile the single changed source file. Use the compile time option CSECT so that each section is named for purposes of rebindability. For example, assume that you have made a change to `unit1.c`. Recompile `unit1.c` by using **c89** as follows:

```
c89 -o unit1.o -W c,"CSECT(myprog)" unit1.c
```

2. Rebind only the changed compile unit into the executable program, which replaces its corresponding binder sections in the program object:

```
cp -m myprog myprog.old
c89 -o myprog unit1.o myprog
```

The **cp** command is optional. It saves a copy of the old executable in case the bind fails in such a way as to damage the executable. `myprog` is overwritten with the result of the bind of `unit1.o`. Like-named sections in `unit1.o` replace those in the `myprog` executable.

The following is an example of a makefile that performs a similar build:

```
_C89_EXTRA_ARGS=1
.EXPORT : _C89_EXTRA_ARGS     1
SRCS = unit0.c unit1.c unit2.c       2
myprog.PRECIOUS : $(SRCS)   3
        @if [ -e $@ ]; then OLD=$@; else OLD=; fi;\
        CMD="$(CC) -Wc,csect $(CFLAGS) $(LDFLAGS) -o $@ $? $$OLD";\  4

        echo $$CMD; $$CMD;
        -@rm -f $(?:b+"$O")
```

1 allow filenames with non-standard suffixes

2 list of source files

3 do not delete `myprog` if the make fails

4 compile source files newer than the executable, and bind

The attribute `.PRECIOUS` prevents such parts from being deleted if `make` fails. `$?` are the dependencies which are newer than the target.

Note:

- You need the `.PRECIOUS` attribute to avoid removing the current executable, since you depend on it as subsequent input.
- If more than one source part changes, and any compiles fail, then on subsequent makes, all compiles are redone.

For a complete description of all **c89** options see Chapter 21, “c89 — Compiler invocation using host environment variables,” on page 535. For a description of **make**, see *z/OS UNIX System Services Command Reference* and for a **make** tutorial, see *z/OS UNIX System Services Programming Tools*.

Advantage

Rebinds are fast because most of the program is already bound. Also, none of the intermediate object modules need to be retained because they are available from the program itself.

Using the non-XPLINK version of the Standard C++ Library and c89

A non-XPLINK Standard C++ Library DLL is available that provides Standard C++ Library support for CICS and IMS. The CICS subsystem does not support XPLINK linkage, rendering the XPLINK Standard C++ Library DLL supplied with the compiler inoperable under this subsystem. The non-XPLINK Standard C++ Library DLL allows support for the Standard C++ Library in the CICS and IMS subsystems, as of z/OS V1R2. Since CICS does not support XPLINK linkage, a non-XPLINK DLL enables the Standard C++ Library under these subsystems.

Note: XPLINK 31-bit applications are supported under the IMS environment.

To use the non-XPLINK Standard C++ Library DLL, you must first link your object modules with the non-XPLINK system definition side-deck. Use the _CXX_PSYSIX environment variable to pass the non-XPLINK side deck information to c++/cxx. The _CXX_PSYSIX environment variable specifies the system definition side-deck list to be used to resolve symbols during the non-XPLINK link-editing phase. The following concatenation should be used:

```
export _CXX_PSYSIX=\
"_CXX_PLIB_PREFIX.SCEELIB(C128N)":\
"_CXX_CLIB_PREFIX.SCLBSID(IOSTREAM,COMPLEX)"
```

where `_CXX_PLIB_PREFIX` and `_CXX_CLIB_PREFIX` are set to a default (for example, CEE and CBC, respectively) during custom installation, or using user overrides.

It is only necessary to specify _CXX_PSYSIX in order to use the non-XPLINK side deck with IPA. Corresponding non-XPLINK IPA link step environment variables default to the value of _CXX_PSYSIX. To run a program with the non-XPLINK DLL, ensure that the SCEERUN data set containing the non-XPLINK DLL is in the MVS search path; that is, either specified in your STEPLIB or already loaded into LPA.

Performance

Due to peformance differences between XPLINK and non-XPLINK linkages, it is expected that an XPLINK program using the XPLINK Standard C++ Library DLL will outperform a non-XPLINK program using the non-XPLINK Standard C++ Library DLL.

It is possible to use the non-XPLINK DLL with an XPLINK application, although this is not preferred. A call to a function of different linkage than the callee will result in a performance degradation due to the overhead cost required to swap from one stack type to the other.

Using the non-XPLINK version of the Standard C++ Library and xlc

To use the non-XPLINK Standard C++ Library DLL with **xlc**, the `exportlist` attribute in the configuration file must include the c128n (instead of c128) member of the CEE.SCEELIB data set.

Performance

Due to performance differences between XPLINK and non-XPLINK linkages, it is expected that an XPLINK program using the XPLINK Standard C++ Library DLL will outperform a non-XPLINK program using the non-XPLINK Standard C++ Library DLL.

It is possible to use the non-XPLINK DLL with an XPLINK application, although this is not preferred. A call to a function of different linkage than the callee will result in a performance degradation due to the overhead cost required to swap from one stack type to the other.

Binding under z/OS batch

You can use the following procedures, which the z/OS XL C/C++ compiler supplies, to invoke the binder:

Procedure name	Description
CEEXL	C bind an XPLINK 32-bit program
CEEXLR	C bind and run an XPLINK 32-bit program
EDCCB	C compile and bind a non-XPLINK 32-bit program
EDCCBG	C compile, bind, and run a non-XPLINK 32-bit program
EDCXCB	C compile and bind an XPLINK 32-bit program
EDCXCBG	C compile, bind, and run an XPLINK 32-bit program
EDCXLDEF	Create C Source from a locale, compile, and bind the XPLINK 32-bit program
CBCB	C++ bind a non-XPLINK 32-bit program
CBCBG	C++ bind and run a non-XPLINK 32-bit program
CBCCB	C++ compile and bind a non-XPLINK 32-bit program
CBCCBG	C++ compile, bind, and run a non-XPLINK 32-bit program
CBCXB	C++ bind an XPLINK 32-bit program
CBCXBG	C++ bind and run an XPLINK 32-bit program
CBCXCB	C++ compile and bind an XPLINK 32-bit program
CBCXCBG	C++ compile, bind, and run an XPLINK 32-bit program
CCNPD1B	C or C++ bind an object compiled using the IPA(PDF1) and NOXPLINK options
CCNXPD1B	C or C++ bind an object compiled using the IPA(PDF1) and XPLINK options
EDCQB	C bind a 64-bit program
EDCQBG	C bind and run a 64-bit program
EDCQCB	C compile and bind a 64-bit program
EDCQCBG	C compile, bind, and run a 64-bit program
CBCQB	C++ bind a 64-bit program
CBCQBG	C++ bind and run a 64-bit program
CBCQCB	C++ compile and bind a 64-bit program
CBCQCBG	C++ compile, bind, and run a 64-bit program

Procedure name	Description
CCNQPD1B	C or C++ bind a 64-bit object compiled using the IPA(PDF1) and LP64 options
CBCLG	Prelink, link, and run a 31-bit non-XPLINK program

If you want to generate DLL code, you must use the binder DYNAM(DLL) option. All the z/OS XL C/C++ supplied cataloged procedures that invoke the binder use the DYNAM(DLL) option. For C++, these cataloged procedures use the DLL versions of the IBM-supplied class libraries by default; the IBM-supplied definition side-deck data set for class libraries, SCLBSID, is included in the SYSLIN concatenation.

z/OS batch example

Figure 35 shows the example source files `USERID.PLAN9.C(UNIT0)`, `USERID.PLAN9.C(UNIT1)`, and `USERID.PLAN9.C(UNIT2)`, which are used to illustrate all of the z/OS batch examples that follow.

```
/* file: USERID.PLAN9.C(UNIT0) */
#include <stdio.h>
extern int f1(void);
extern int f4(void);
int main(void) {
int rc1;
int rc4;
rc1 = f1();
rc4 = f4();
if (rc1 != 1) printf("fail rc1 is %d\n",rc1);
if (rc4 != 40) printf("fail rc4 is %d\n",rc4);
return 0;
}

/* file: USERID.PLAN9.C(UNIT1) */
int f1(void) { return 1; }

/* file: USERID.PLAN9.C(UNIT2) */
int f2(void) { return 20;}
int f3(void) { return 30;}
int f4(void) { return f2()*2; /* 40 */ }
```

Figure 35. Example source files

Steps for single final bind under z/OS batch

Before you begin: Compile each source file.

Perform the following steps to complete a final single bind of everything:

1. Compile each source file to generate the object modules `USERID.PLAN9.OBJ(UNIT0)`, `USERID.PLAN9.OBJ(UNIT1)`, and `USERID.PLAN9.OBJ(UNIT2)`. Use the EDCC procedure as follows:

```
//COMP0  EXEC EDCC,
//       INFILE='USERID.PLAN9.C(UNIT0)',
//       OUTFILE='USERID.PLAN9.OBJ,DISP=SHR',
//       CPARM='LONG,RENT'
//COMP1  EXEC EDCC,
//       INFILE='USERID.PLAN9.C(UNIT1)',
//       OUTFILE='USERID.PLAN9.OBJ,DISP=SHR',
//       CPARM='LONG,RENT'
//COMP2  EXEC EDCC,
//       INFILE='USERID.PLAN9.C(UNIT2)',
//       OUTFILE='USERID.PLAN9.OBJ,DISP=SHR',
//       CPARM='LONG,RENT'
```

2. Perform a final single bind to produce the executable program `USERID.PLAN9.LOADE(MYPROG)`. Use the CBCB procedure as follows:

```
//BIND EXEC CBCB,OUTFILE='USERID.PLAN9.LOADE,DISP=SHR'
//OBJECT DD DSN=USERID.PLAN9.OBJ,DISP=SHR
//SYSIN DD *
  INCLUDE OBJECT(UNIT0)
  INCLUDE OBJECT(UNIT1)
  INCLUDE OBJECT(UNIT2)
  NAME MYPROG(R)
/*
```

The OUTFILE parameter along with the NAME control statement specify the name of the output executable to be created.

Advantage

This method is simple, and is consistent with existing methods of building applications, such as makefiles.

Steps for binding each compile unit under z/OS batch

Before you begin: Compile each source file and also bind it.

Perform the following steps to complete a final bind of all the partially bound units:

1. Compile and bind each source file to generate the partially bound program objects `USERID.PLAN9.LOADE(UNIT0)`, `USERID.PLAN9.LOADE(UNIT1)`, and `USERID.PLAN9.LOADE(UNIT2)`, which may have unresolved references. In this example, references to `f1()` and `f4()` in `USERID.PLAN9.LOADE(UNIT0)` are unresolved. Compile and bind each unit by using the EDCCB procedure as follows:

```
//COMP0  EXEC EDCCB,
//       CPARM='CSECT(MYPROG)',
//       BPARM='LET,CALL(NO),ALIASES(ALL)',
//       INFILE='USERID.PLAN9.C(UNIT0)',
//       OUTFILE='USERID.PLAN9.LOADE(UNIT0),DISP=SHR'
//COMP1  EXEC EDCCB,
//       CPARM='CSECT(MYPROG)',
//       BPARM='LET,CALL(NO),ALIASES(ALL)',
//       INFILE='USERID.PLAN9.C(UNIT1)',
//       OUTFILE='USERID.PLAN9.LOADE(UNIT1),DISP=SHR'
//COMP2  EXEC EDCCB,
//       CPARM='CSECT(MYPROG)',
//       BPARM='LET,CALL(NO),ALIASES(ALL)',
//       INFILE='USERID.PLAN9.C(UNIT2)',
//       OUTFILE='USERID.PLAN9.LOADE(UNIT2),DISP=SHR'
```

The CALL(NO) option prevents autocall processing.

---

2. Perform the final single bind to produce the executable program MYPROG by using the CBCB procedure:

   You have two methods for building the program.

   a. Explicit include: In this method, when you invoke the CBCB procedure, you use include cards to explicitly specify all the program objects that make up this executable. Automatic library call is done only for the non-XPLINK data sets CEE.SCEELKED, CEE.SCEELKEX, and CEE.SCEECPP because those are the only libraries pointed to by ddname SYSLIB. Using CBCXB for XPLINK, automatic library is done only for CEE.SCEEBND2. For example:

```
//BIND   EXEC CBCB,
//         OUTFILE='USERID.PLAN9.LOADE,DISP=SHR'
//INPGM  DD DSN=USERID.PLAN9.LOADE,DISP=SHR
//SYSIN DD *
   INCLUDE INPGM(UNIT0)
   INCLUDE INPGM(UNIT1)
   INCLUDE INPGM(UNIT2)
   NAME MYPROG(R)
/*
```

   b. Library search: In this method, you specify the compile unit that contains your `main()` function, and allocate your object library to ddname SYSLIB. The binder performs a library search and includes additional members from your object library, and generates the output program object. You invoke the binder as follows:

```
//BIND   EXEC CBCB,
//       OUTFILE='USERID.PLAN9.LOADE,DISP=SHR'
//INPGM  DD DSN=USERID.PLAN9.LOADE,DISP=SHR
//SYSLIB DD
//         DD
//         DD
//         DD DSN=USERID.PLAN9.LOADE,DISP=SHR
//SYSIN DD *
   INCLUDE INPGM(UNIT0)
   NAME MYPROG(R)
/*
```

Advantage

Binding a set of partially bound program objects into a fully bound program object is faster than binding object modules into a fully bound program object. For example, a central build group can create the partially bound program objects. You can then use these program objects and their changed object modules to create a development program object.

Steps for building and using a DLL under z/OS batch

Perform the following steps to build `USERID.PLAN9.C(UNIT1)` and `USERID.PLAN9.C(UNIT2)` into DLL `USERID.PLAN.LOADE(ONETWO)`, which exports functions `f1()`, `f2()`, `f3()` and `f4()`. Build `USERID.PLAN9.C(UNIT0)` into a program which dynamically links to functions `f1()` and `f4()` defined in the DLL build and use a DLL under z/OS batch.

1. Compile `USERID.PLAN9.C(UNIT1)` and `USERID.PLAN9.C(UNIT2)` to generate the object modules `USERID.PLAN9.OBJ(UNIT1)` and `USERID.PLAN9.OBJ(UNIT2)`, which define the functions to be exported. Use the EDCC procedure as follows:

```
//* Compile UNIT1
//CC1 EXEC EDCC,
//        CPARM='OPTF(DD:OPTIONS)',
//        INFILE='USERID.PLAN9.C(UNIT1)',
//        OUTFILE='USERID.PLAN9.OBJ(UNIT1),DISP=SHR'
//COMPILE.OPTIONS DD *
   LIST RENT LONGNAME EXPORTALL
*/
//* Compile UNIT2
//CC2 EXEC EDCC,
//        CPARM='OPTF(DD:OPTIONS)',
//         INFILE='USERID.PLAN9.C(UNIT2)',
//         OUTFILE='USERID.PLAN9.OBJ(UNIT2),DISP=SHR'
//COMPILE.OPTIONS DD *
   LIST RENT LONGNAME EXPORTALL
*/
```

2. Bind `USERID.PLAN9.OBJ(UNIT1)` and `USERID.PLAN9.OBJ(UNIT2)` to generate the DLL `ONETWO`:

```
//* Bind the DLL
//BIND1    EXEC CBCB,
//          BPARM='CALL,DYNAM(DLL)',
//          OUTFILE='USERID.PLAN9.LOADE(ONETWO),DISP=SHR'
//INOBJ    DD DISP=SHR,DSN=USERID.PLAN9.OBJ
//SYSDEFSD DD DISP=SHR,DSN=USERID.PLAN9.IMP(ONETWO)
//SYSLIN   DD *
   INCLUDE INOBJ(UNIT1)
   INCLUDE INOBJ(UNIT2)
   NAME ONETWO(R)
/*
```

   When you bind code with exported symbols, you must specify the binder option DYNAM(DLL). You must also allocate the definition side-deck DD SYSDEFSD to define the definition side-deck where the IMPORT control statements are to be written.

   In addition to the DLL being generated, a list of IMPORT control statements is written to DD SYSDEFSD. One IMPORT control statement is written for each exported symbol. These generated control statements will be included later as input to the bind step of an application that uses this DLL, so that it can import the symbols.

3. Compile `USERID.PLAN9.C(UNIT0)` so that it may import unresolved symbols, and bind with the file of IMPORT control statements from the DLLs build:

```
//* Compile the DLL user
//CC1 EXEC EDCC,
//         CPARM='OPTF(DD:OPTIONS)',
//         INFILE='USERID.PLAN9.C(UNIT0)',
//         OUTFILE='USERID.PLAN9.OBJ(UNIT0),DISP=SHR'
//COMPILE.OPTIONS DD *
   LIST RENT LONGNAME DLL
/*
//* Bind the DLL user with input IMPORT statements from the DLL build
//BIND1    EXEC CBCB,
//         BPARM='CALL,DYNAM(DLL)',
//         OUTFILE='USERID.PLAN9.LOADE,DISP=SHR'
//INOBJ    DD DISP=SHR,DSN=USERID.PLAN9.OBJ
//IMP      DD DISP=SHR,DSN=USERID.PLAN9.IMP
//SYSLIN   DD *
  INCLUDE INOBJ(UNIT0)
  INCLUDE IMP(ONETWO)
  ENTRY CEESTART
  NAME DLL12USR(R)
/*
```

Advantage

The bind time advantage of using DLLs is that you only need to rebuild the DLL with the changed code in it. You do not need to rebuild all applications that use the DLL in order to use the changed code.

Build and use a 64-bit application under z/OS batch

Creating a 64-bit application under z/OS batch is similar to creating a 31-bit application. There are, however, some subtle differences, which the following C++ example demonstrates.

As of z/OS C/C++ V1R6, new PROCs are available for binding and running with 64-bit applications. There are no new PROCs for a 64-bit compile (without binding or running) but you can use the previously existing C and C++ PROCs, along with the LP64 compiler option, to create 64-bit object files that can then be used with the new 64-bit enabled PROCs. Then, rather than using the regular binding PROCs (such as CBCB and EDCCBG), you need to use the new 64-bit PROCs for binding; for example, CBCQB and EDCQCBG.

Example: The following example shows how to implement these instructions. In this example, we use the CBCC PROC and the LP64 compiler option for our first 64-bit compile, and the CBCQCBG PROC to compile another source file in 64-bit mode, bind it (along with the first object file we produced), and finally run the resulting load module.

```
#include <iostream>
void lp64_function() {
#ifdef _LP64
  std::cout << "Hello World, z/OS has 64-bit programs now!" << std::endl;
#else
  std::cout << "Uh oh, someone didn't compile this file with LP64" << std::endl;
#endif
 }
```

```
HELLO2.C
void lp64_function();
int main() {
  lp64_function();
}
```

```
//USERID   JOB (641A,2317),'Programmer Name',REGION=128M,
//         CLASS=B,MSGCLASS=S,NOTIFY=&SYSUID;,MSGLEVEL=(1,1)
//ORDER JCLLIB ORDER=(CBC.SCCNPRC)
//*-------------------------------------------------------------------
//* C++ Compile using LP64 compiler option
//*-------------------------------------------------------------------
//COMPILE  EXEC CBCC,
//         INFILE='USERID.LP64.SOURCE(HELLO1)',
//         OUTFILE='USERID.LP64.OBJECT(HELLO1),DISP=SHR',
//         CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS  DD DATA,DLM='/>'
  LP64
/>
//*-------------------------------------------------------------------
//* C++ 64-bit Compile, Bind, and Go Step
//*-------------------------------------------------------------------
//COBINDGO EXEC CBCQCBG,
//         INFILE='USERID.LP64.SOURCE(HELLO2)',
//         OUTFILE='USERID.LP64.LOAD(HELLO),DISP=SHR'
//BIND.SYSIN DD DATA,DLM='/>'
 INCLUDE OBJECT(HELLO1)
/>
//OBJECT   DD DSN=USERID.LP64.OBJECT,DISP=SHR
```

Build and use a 64-bit application with IPA under z/OS batch

Example: This example shows you how to IPA Compile both a C source file and a C++ source file in 64-bit mode, then IPA Link them, bind them (in 64-bit mode), and run the resulting load module.

This example also shows that when you want to create an IPA optimized program that makes use of calls to standard library functions, you need to explicitly let IPA know where to find the libraries that it will link with. The location of the standard library functions is not included by default in the IPA Link PROCs because if you do not actually ever call a standard library function, IPA will spend time analyzing the unused libraries before realizing your program does not need them, thereby unnecessarily slowing down your compilation time. If you are building a C++ program and do not tell IPA where to find the libraries it needs at IPA Link time, the IPA Linker will complain about the unresolved symbols it cannot find. You can tell IPA where the standard libraries are by adding the following lines to the CBCQI or EDCQI job steps in your JCL:

```
//SYSIN    DD DATA,DLM='/>'
 INCLUDE OBJECT(HELLO)
 INCLUDE SYSLIB(C64,IOSX64)
 INCLUDE SYSLIB(CELQSCPP,CELQS003)
/>
//OBJECT   DD DSN=USER.TEST.OBJECT,DISP=SHR
```

Note: The `USER.TEST.OBJECT` data set and the `HELLO` PDS member are meant to represent the object file(s) for your application, which you should have created using a previous IPA Compile step.

Example: The following example shows how to implement these instructions.

```
//USERID  JOB (641A,2317),'Programmer Name',REGION=128M,
//         CLASS=B,MSGCLASS=S,NOTIFY=&SYSUID;,MSGLEVEL=(1,1)
//ORDER JCLLIB ORDER=(CBC.SCCNPRC)
//*-------------------------------------------------------------------
//* 64-bit C IPA Compile
//*-------------------------------------------------------------------
//IPACOMP1 EXEC EDCC,
//         OUTFILE='USERID.IPA.LP64.OBJECT(OBJECT1),DISP=SHR',
//         CPARM='OPTFILE(DD:OPTIONS)'
//SYSIN    DD DATA,DLM='/>'

#include <time.h>
#include <string.h>
int get_time_of_day(char* output) {

  time_t     time_val;
  struct tm* time_struct;
  char*      time_string;


  if ( -1 != time(&time_val;) ) {

    time_struct = localtime(&time_val;);

    if ( NULL != time_struct ) {

      time_string = asctime(time_struct);

      if ( NULL != time_string ) {

        strcpy(output, time_string);
        output[strlen(output) - 1] = 0;

        return 0;
      }
    }
  }

  return 1;
}
/>
//OPTIONS  DD DATA,DLM='/>'
 IPA(NOOBJECT,NOLINK) LP64 LONGNAME OPT
/>
//*-------------------------------------------------------------------
//* 64-bit C++ IPA Compile with very high optimization
//*-------------------------------------------------------------------
//IPACOMP2 EXEC CBCC,
//         OUTFILE='USERID.IPA.LP64.OBJECT(OBJECT2),DISP=SHR',
//         CPARM='OPTFILE(DD:OPTIONS)'
//SYSIN    DD DATA,DLM='/>'

#include <iostream>
#include <string>

using std::cout;
using std::endl;
using std::string;

extern "C" int get_time_of_day(char*);

int main() {

  char* tod;

  tod = new char[100];
  if ( 0 == get_time_of_day(tod) ) {

    cout << "The current time is: " << tod << endl;

  } else {

    cout << "Error: Could not determine the time" << endl;

  }

  delete tod;

  return 0;
}
/>
//OPTIONS  DD DATA,DLM='/>'
 IPA(NOOBJECT,NOLINK) LP64 OPT(3)
/>
//*-------------------------------------------------------------------
//* 64-bit C++ IPA Link
//*-------------------------------------------------------------------
//IPALINK  EXEC CBCQI,
//         OUTFILE='USERID.IPALINK.LP64.OBJECT(IPAOBJ),DISP=SHR',
//         IPARM='IPA(LEVEL(2),MAP) LONGNAME'
//SYSIN    DD DATA,DLM='/>'
 INCLUDE OBJECT(OBJECT1)
 INCLUDE OBJECT(OBJECT2)
 INCLUDE SYSLIB(C64,IOSX64)
 INCLUDE SYSLIB(CELQSCPP,CELQS003)
/>
//OBJECT   DD DSN=USERID.IPA.LP64.OBJECT,DISP=SHR
//*-------------------------------------------------------------------
//* C++ 64-bit Bind and Go Step
//*-------------------------------------------------------------------
//BINDGO   EXEC CBCQBG,
//         INFILE='USERID.IPALINK.LP64.OBJECT(IPAOBJ)',
//         OUTFILE='USERID.LP64.LOAD(FINALEXE),DISP=SHR'
//SYSIN    DD DATA,DLM='/>'
 INCLUDE OBJECT(IPAOBJ)
/>
//OBJECT   DD DSN=USERID.IPA.LP64.OBJECT,DISP=SHR
```

Using the non-XPLINK version of the Standard C++ Library and z/OS batch

A non-XPLINK Standard C++ Library DLL is available that provides Standard C++ Library support for CICS and IMS. The CICS subsystem does not support XPLINK linkage, rendering the XPLINK Standard C++ Library DLL supplied with the compiler inoperable under this subsystem. The non-XPLINK Standard C++ Library DLL allows support for the Standard C++ Library in the CICS and IMS subsystems, as of z/OS V1R2. Since CICS does not support XPLINK linkage, a non-XPLINK DLL enables the Standard C++ Library under these subsystems.

Note: XPLINK 31-bit applications are supported under the IMS environment.

All non-XPLINK C++ PROCs containing bind and pre-link steps need to be invoked with the STDLIBSD PROC variable set to `c128n`, or overridden in order to use the non-XPLINK Standard C++ Library DLL. These PROCs are: CBCB, CBCBG, CBCCB, CBCCBG, CBCCL, CBCCLG, CBCL, CBCLG and CCNPD1B.

The appropriate DD statements in these PROCs must be overridden:

- For a bind step, the non-XPLINK side deck must override the XPLINK side-deck or the SYSLIN concatenation.

- For a pre-link step, the non-XPLINK side deck must override the XPLINK side deck or the SYSIN concatenation.

The following concatenations added to the calling JCL will override the appropriate DD statement of the corresponding CBC PROC:

CBCB, CBCBG

```
//SYSLIN   DD
//         DD  DSN=&LIBPRFX..SCEELIB(C128N),DISP=SHR
```

CBCCB, CBCCBG

```
//BIND.SYSLIN   DD
//         DD  DSN=&LIBPRFX..SCEELIB(C128N),DISP=SHR
```

CBCL, CBCLG

```
//SYSLIN   DD
//         DD  DSN=&LIBPRFX..SCEELIB(C128N),DISP=SHR
```

CBCCL, CBCCLG

```
//PLKED.SYSLIN   DD
//         DD  DSN=&LIBPRFX..SCEELIB(C128N),DISP=SHR
```

The following concatenation added to the calling JCL will override the appropriate DD statement of the corresponding CCN PROC. Note that CICS does not support PDF.

CCNPD1B

```
//SYSLIN   DD
//         DD  DSN=&LIBPRFX..SCEELIB(C128N),DISP=SHR
```

Restrictions concerning use of non-XPLINK Standard C++ Library DLL

The following is a list of restrictions:

- No enhanced ASCII functionality support:

  The non-XPLINK Standard C++ Library DLL does not provide enhanced ASCII functionality support as ASCII run-time functions require XPLINK linkage. Classes and functions sensitive to character encoding are provided in EBCDIC alone in the non-XPLINK DLL.
- No PDF PROC support for CICS:

  CICS does not support Profile Directed Feedback (PDF). The non-XPLINK PDF PROC, CCNPD1B, cannot be used with CICS. The XPLINK PDF CCNXPD1B PROC and the 64-bit PDF CCNQPD1B PROC cannot be used with CICS as well.

Steps for rebinding a changed compile unit under z/OS batch

Before you begin: Make a change to a single source file and rebuild the application.

Perform the following steps to recompile the single changed source file and make a replacement of its binder sections in the program:

1. Recompile the single changed source file. Use the CSECT compiler option so that each section is named for purposes of rebindability. For example, assume that you have made a change to `USERID.PLAN9.C(UNIT1)`. Recompile the source file using the EDCC procedure as follows:

```
//* Compile UNIT1 user
//CC EXEC EDCC,
//            CPARM='OPTF(DD:OPTIONS)',
//            INFILE='USERID.PLAN9.C(UNIT1)',
//            OUTFILE='USERID.PLAN9.OBJ(UNIT1),DISP=SHR'
//COMPILE.OPTIONS DD *
   LIST RENT LONGNAME DLL CSECT(MYPROG)
/*
```

2. Rebind only the changed compile unit into the executable program, which replaces its corresponding binder sections in the program object:

```
//BIND   EXEC CBCB,
//       OUTFILE='USERID.PLAN9.LOADE,DISP=SHR'
//OLDPGM  DD DSN=USERID.PLAN9.LOADE,DISP=SHR
//NEWOBJ  DD DSN=USERID.PLAN9.OBJ,DISP=SHR
//SYSIN DD *
   INCLUDE NEWOBJ(UNIT1)
   INCLUDE OLDPGM(MYPROG)
   NAME NEWPGM(R)
/*
```

Advantage

Rebinds are fast because most of the program is already bound, and none of the intermediate object modules are retained.

Writing JCL for the binder

You can use cataloged procedures rather than supply all the JCL required for a job step. However, you can use JCL statements to override the statements of the cataloged procedure.

Use the EXEC statement in your JCL to invoke the binder. The EXEC statement to invoke the binder is:

```
 //BIND EXEC PGM=IEWL
```

Use PARM parameter for the EXEC statement to select one or more of the optional facilities that the binder provides.

Example: You can specify the OPTIONS option on the PARM parameter to read binder options from the ddname OPTS, as follows:

```
//BIND1 EXEC PGM=IEWL,PARM='OPTIONS=OPTS'
//OPTS     DD *
     AMODE=31,MAP
     RENT,DYNAM=DLL
     CASE=MIXED,COMPAT=CURR
/*
//SYSLIB   DD DISP=SHR,DSN=CEE.SCEELKEX
//         DD DISP=SHR,DSN=CEE.SCEELKED
//         DD DISP=SHR,DSN=CEE.SCEECPP
//SYSLIN   DD DISP=SHR,DSN=USERID.PLAN9.OBJ(P1)
//         DD DISP=SHR,DSN=CBC.SCLBSID(IOSTREAM)
//SYSLMOD  DD DISP=SHR,DSN=USERID.PLAN9.LOADE(PROG1)
//SYSPRINT DD SYSOUT=*
```

In this example, object module P1, which was compiled NOXPLINK, is bound using the IOSTREAM DLL definition side-deck. The Language Environment non-XPLINK run-time libraries SCEELKED, SCEELKEX, and SCEECPP are statically bound to produce the program object PROG1.

Example: If the object module P1 was compiled XPLINK, then the JCL would be:

```
//BIND1 EXEC PGM=IEWL,PARM='OPTIONS=OPTS'
//OPTS DD *
AMODE=31,MAP
RENT,DYNAM=DLL
CASE=MIXED,COMPAT=CURR
LIST=NOIMP
/*
//SYSLIB DD DSN=CEE.SCEEBND2,DISP=SHR
//SYSLIN DD DSN=USERID.PLAN9.OBJ(P1),DISP=SHR
// DD DSN=CEE.SCEELIB(CELHSCPP),DISP=SHR
// DD DSN=CEE.SCEELIB(CELHS003),DISP=SHR
// DD DSN=CEE.SCEELIB(CELHS001),DISP=SHR
// DD DISP=SHR,DSN=CBC.SCLBSID(IOSTREAM)
//SYSLMOD DD DISP=SHR,DSN=USERID.PLAN9.LOADE(PROG1)
//SYSPRINT DD SYSOUT=*
```

Example: If the object module P1 was compiled LP64, then the JCL would be:

```
//BIND1 EXEC PGM=IEWL,PARM='OPTIONS=OPTS'
//OPTS DD *
AMODE=64,MAP
RENT,DYNAM=DLL
CASE=MIXED,COMPAT=CURR
LIST=NOIMP
/*
//SYSLIB DD DSN=CEE.SCEEBND2,DISP=SHR
//SYSLIN DD DSN=USRID.PLAN9.OBJ(P1),DISP=SHR
// DD DSN=CEE.SCEELIB(CELQHSCPP),DISP=SHR
// DD DSN=CEE.SCEELIB(CELQHS003),DISP=SHR
// DD DSN=CBC.SCLBSID(IOSX64),DISP=SHR
// DD DISP=SHR,DSN=CBC.SCLBSID(IOSTREAM)
//SYSLMOD DD DISP=SHR,DSN=USERID.PLAN9.LOADE(PROG1)
//SYSPRINT DD SYSOUT=*
```

For more information on the files given, please refer to "LP64 libraries" on page 433.

The binder always requires three standard data sets. You must define these data sets on DD statements with the ddnames SYSLIN, SYSLMOD, and SYSPRINT.

Example: A typical sequence of job control statements for binding an object module into a program object is shown below. In the following non-XPLINK example, the binder control statement NAME puts the program object into the PDSE USER.LOADE with the member name PROGRAM1.

```
//BIND     EXEC  PGM=IEWL,PARM='MAP'
//SYSPRINT DD *                           << out: binder listing
//SYSDEFSD DD DUMMY                       << out: generated IMPORTs
//SYSLMOD  DD DISP=SHR,DSN=USERID.PLAN9.LOADE << out: PDSE of executables
//SYSLIB   DD DISP=SHR,DSN=CEE.SCEELKED   << in:  autocall libraries to search
//         DD DISP=SHR,DSN=CEE.SCEELKEX
//         DD DISP=SHR,DSN=CEE.SCEECPP
//INOBJ    DD DISP=SHR,DSN=USERID.PLAN.OBJ       << in: compiler object code
//SYSLIN DD*
   INCLUDE INOBJ(UNIT0)
   INCLUDE INOBJ(UNIT1)
   INCLUDE INOBJ(UNIT2)
   ENTRY CEESTART
   NAME  PROGRAM1(R)
/*
```

You can explicitly include members from a data set like USERID.PLAN.OBJ, as shown in this example. If you want to be more flexible and less explicit, include only one member, typically the one that contains the entry point (e.g. `main()`). Then you can add USERID.PLAN.OBJ to the SYSLIB concatenation so that a library search brings in the remaining members.

Binding under TSO using CXXBIND

This topic describes how to bind your z/OS XL C++ or z/OS XL C program in TSO by invoking the CXXBIND REXX EXEC. This REXX EXEC invokes the binder and creates an executable program object.

Note: This REXX EXEC does not support 64-bit binding. You must use the PROCs or **c89**, **cc**, **c++**, or **cxx** commands under z/OS UNIX System Services to perform 64-bit binding.

If you specify a data set name in an option, and the high-level qualifier of the data set is not the same as your user prefix, you must use the fully qualified name of the data set and place single quotation marks around the entire name.

If you specify a z/OS UNIX file name in an option, it must be an absolute file name; it must begin with a slash (/). You can include commas and special characters in file names, but you must enclose file names that contain special characters or commas in single quotation marks. If a single quotation mark is part of the file name, you must specify the quotation mark twice.

The syntax for the CXXBIND EXEC is:

```
                   ┌─,──────────────────┐
►►──CXXBIND──OBJ──(──▼─┬─input-object───┬─┴──)─────────────────────────────►
                       └─'input-object'─┘

►──┬──────────────────────────────────┬────────────────────────────────────►
   │             ┌─,──────────────┐   │
   └─OPT──(──────▼──binder_option;─┴──)─┘
```

```
►─┬──────────────────────────────────────────────┬──────────────────────►
  │           ┌─,─────────────────────────┐       │
  │           ▼                           │       │
  └─LIB──(──┬─search-library-name───────┬─┴──)─┘
            └─'search-library-name'─────┘

►─┬─────────────────────────────────────────────────┬───────────────────►
  └─LOAD──(──┬─output_program_object───┬──)─┘
             └─'output_program_object'─┘

►─┬─────────────────────────────────────────────────────┬───────────────►
  └─IMP──(──┬─file_of_generated_imports───┬──)─┘
            └─'file_of_generated_imports'─┘

►─┬──────────────────────────────────────┬──┬──────────┬────────────────►◄
  └─LIST──(──┬─output_listing───┬──)─┘  └─XPLINK─┘
             └─'output_listing'─┘
```

`OBJ`
: You must always specify the input file names by using the OBJ keyword parameter. Each input file must be one of the following:

- An object module that can be a PDS member, a sequential data set, or a z/OS UNIX file
- A load module that is a PDS member
- A program object that can be a PDSE member or a z/OS UNIX file
- A text file that contains binder statements. The file can be a PDS member, a sequential data set, or a z/OS UNIX file

`OPT`
: Use the OPT keyword parameter to specify binder options. For example, if you want the binder to use the MAP option, specify the following:

```
CXXBIND OBJ(PLAN9.OBJ(PROG3)) OPT('MAP')...
```

`LIB`
: Use the LIB keyword parameter to specify the PDS and PDSE libraries that the binder should search to resolve unresolved external references during a library search of the DD SYSLIB.

The default libraries that are used when the XPLINK option is not specified are the CEE.SCEELKED, CEE.SCEELKEX, and CEE.SCEECPP C/C++ libraries and the CBC.SCLBSID C++ class library. The default libraries that are used when the XPLINK option is specified are the CEE.SCEEBND2 and CEE.SCEELIB C/C++ libraries and the CBC.SCLBSID C++ class library . The default library names are added to the ddname SYSLIB concatenation if library names are specified with the LIB keyword parameter.

`LOAD`
: Use the LOAD keyword parameter to specify where the resultant executable program object (which must be a PDSE member, or a z/OS UNIX file) should be stored.

`IMP`
: Use the IMP keyword parameter to specify where the generated IMPORT control statements should be written.

`LIST`	Use the LIST keyword parameter to specify where the binder listing should be written. If you specify *, the binder directs the listing to your console.
`XPLINK`	Use the XPLINK keyword parameter when you are building an XPLINK executable program object. Specifying XPLINK will change the default libraries as described under the LIB option.

TSO example

Figure 36 shows the example source files `PLAN9.C(UNIT0)`, `PLAN9.C(UNIT1)`, and `PLAN9.C(UNIT2)`, that are used to illustrate all of the TSO examples that follow.

```
/* file: USERID.PLAN9.C(UNIT0) */
#include <stdio.h>
extern int f1(void);
extern int f4(void);
int main(void) {
int rc1;
int rc4;
rc1 = f1();
rc4 = f4();
if (rc1 != 1) printf("fail rc1 is %d\n",rc1);
if (rc4 != 40) printf("fail rc4 is %d\n",rc4);
return 0;
}

/* file: USERID.PLAN9.C(UNIT1) */
int f1(void) { return 1; }

/* file: USERID.PLAN9.C(UNIT2) */
int f2(void) { return 20;}
int f3(void) { return 30;}
int f4(void) { return f2()*2; /* 40 */ }
```

Figure 36. Example Source Files

Steps for single final bind under TSO

Before you begin: Compile each source file.

Perform the following steps to complete a single final bind of everything:

1. Compile each unit to generate the object modules `PLAN9.OBJ(UNIT0)`, `PLAN9.OBJ(UNIT1)`, and `PLAN9.OBJ(UNIT2)`. Use the CC REXX exec as follows:
   ```
   CC PLAN9.C(UNIT0) OBJECT(PLAN9.OBJ) CSECT(MYPROG)
   CC PLAN9.C(UNIT1) OBJECT(PLAN9.OBJ) CSECT(MYPROG)
   CC PLAN9.C(UNIT2) OBJECT(PLAN9.OBJ) CSECT(MYPROG)
   ```

2. Perform a final single bind to produce the executable program `PLAN9.LOADE(MYPROG)`. Use the CXXBIND REXX exec as follows:
   ```
   CXXBIND OBJ(PLAN9.OBJ(UNIT0),PLAN9.OBJ(UNIT1),PLAN9.OBJ(UNIT2))
      LOAD(PLAN9.LOADE(MYPROG))
   ```

Advantage

This method is simple, and is consistent with existing methods of building applications, such as makefiles.

Steps for binding each compile unit under TSO

Before you begin: Compile and bind each source file.

Perform the following steps to complete a final bind of all the partially bound units:

1. Compile and bind each source file to generate the partially bound program objects `PLAN9.LOADE(UNIT0)`, `PLAN9.LOADE(UNIT1)`, and `PLAN9.LOADE(UNIT2)`, which may have unresolved references. In this example, references to `f1()` and `f4()` in `PLAN9.LOADE(UNIT0)` are unresolved. Compile and bind each unit by using the CC and CXXBIND REXX execs as follows:

   ```
   CC PLAN9.C(UNIT0) OBJECT(PLAN9.OBJ) CSECT(MYPROG)
   CXXBIND OBJ(PLAN9.OBJ(UNIT0)) OPT('LET,CALL(NO)')
      LOAD(PLAN9.LOADE(UNIT0))

   CC PLAN9.C(UNIT1) OBJECT(PLAN9.OBJ) CSECT(MYPROG)
   CXXBIND OBJ(PLAN9.OBJ(UNIT1)) OPT('LET,CALL(NO)')
      LOAD(PLAN9.LOADE(UNIT1))

   CC PLAN9.C(UNIT2) OBJECT(PLAN9.OBJ) CSECT(MYPROG)
   CXXBIND OBJ(PLAN9.OBJ(UNIT2)) OPT('LET,CALL(NO)')
      LOAD(PLAN9.LOADE(UNIT1))
   ```

   The CALL(NO) option prevents autocall processing.

2. Perform the final single bind to produce the executable program `MYPROG` by using the CXXBIND REXX exec:

   ```
   CXXBIND OBJ(PLAN9.LOADE(UNIT0), PLAN9.LOADE(UNIT1), PLAN9.LOADE(UNIT2))
      LOAD(PLAN9.LOADE(MYPROG))
   ```

Advantage

Binding a set of partially bound program objects into a fully bound program object is faster than binding object modules into a fully bound program object. For example, a central build group can create the partially bound program objects. You can then use these program objects and their changed object modules to create a development program object.

Steps for building and using a DLL under TSO

Perform the following steps to build `PLAN9.C(UNIT1)` and `PLAN9.C(UNIT2)` into DLL `PLAN9.LOADE(ONETWO)` which exports functions `f1()`, `f2()`, `f3()` and `f4()`. Then build `PLAN9.C(UNIT0)` into a program which dynamically links to functions `f1()` and `f4()` defined in the DLL.

1. Compile `PLAN9.C(UNIT1)` and `PLAN9.C(UNIT2)` to generate the object modules `PLAN9.OBJ(UNIT1)` and `PLAN9.OBJ(UNIT2)` which have functions to be exported. Use the CC REXX exec as follows:

   ```
   CC PLAN9.C(UNIT1) OBJECT(PLAN9.OBJ) EXPORTALL,LONGNAME,DLL,CSECT(MYPROG)
   CC PLAN9.C(UNIT2) OBJECT(PLAN9.OBJ) EXPORTALL,LONGNAME,DLL,CSECT(MYPROG)
   ```

2. Bind `PLAN9.OBJ(UNIT1)` and `PLAN9.OBJ(UNIT2)` to generate the DLL `PLAN9.LOADE(ONETWO)`:

   ```
   CXXBIND OBJ(PLAN9.LOADE(UNIT0), PLAN9.LOADE(UNIT1)) IMP (PLAN9.IMP(ONETWO))
         LOAD(PLAN9.LOADE(ONETWO))
   ```

   When you bind code with exported symbols, you must specify the binder option DYNAM(DLL). You must also use the CXXBIND IMP option to define the definition side-deck where the IMPORT control statements are to be written.

3. Compile `PLAN9.C(UNIT0)` so that it may import unresolved symbols, and bind with `PLAN9.IMP(ONETWO)`, which is the definition side-deck containing IMPORT control statements from the DLL build:

```
CC PLAN9.C(UNIT0) OBJECT(PLAN9.OBJ) CSECT(MYPROG),DLL
CXXBIND OBJ(PLAN9.LOADE(UNIT0), PLAN9.IMP(ONETWO)) LOAD(PLAN9.LOADE(DLL12USR))
```

Advantage

The bind time advantage of using DLLs is that you only need to rebuild the DLL with the changed code in it. You do not need to rebuild all applications that use the DLL in order to use the changed code.

Steps for rebinding a changed compile unit under TSO

Before you begin: Make a change to a single source file and rebuild the application.

Perform the following steps to recompile the single changed source file and make a replacement of its binder sections in the program:

1. Recompile the single changed source file. Use the CSECT compiler option to ensure that each section is named for purposes of rebindability. For example, assume that you have made a change to `PLAN9.C(UNIT1)`. Recompile `PLAN9.C(UNIT1)` by using the CC REXX exec as follows:

```
CC PLAN9.C(UNIT1) OBJECT(PLAN9.OBJ) CSECT(MYPROG)
```

2. Rebind only the changed source file into the executable program, which replaces its corresponding binder sections in the program object:

```
CXXBIND OBJ(PLAN9.OBJ(UNIT1), PLAN9.LOADE(MYPROG))
     LOAD(PLAN9.LOADE(NEWPROG)
```

Advantage

Rebinds are fast because most of the program is already bound, and none of the intermediate object modules are retained.

Chapter 10. Binder processing

You can bind any z/OS XL C/C++ object module or program object.

Object files with long name symbols, reentrant writable static symbols, and DLL-style function calls require additional processing to build global data for the application. You can always rebind if you don't require this additional processing. You can also re-bind if you used the binder for this additional processing and produced a program object (in other words, you didn't use the prelinker). If you used the prelinker and performed this additional processing, you cannot later rebind. If you have done additional processing and output it to a PDS, you cannot rebind it. For further information, refer to "About prelinking, linking, and binding" on page 8.

Various limits have been increased from the linkage-editor; for example, the z/OS V1R6 binder supports variable and function names up to 32767 characters long.

For the Writable Static Area (WSA), the binder assigns relative offsets to objects in the Writable Static Area and manages initialization information for objects in the Writable Static Area. The Writable Static Area is not loaded with the code. Language Environment run time requests it.

For C++, the binder collects constructor calls and destructor calls for static C++ objects across multiple compile units. C++ linkage names appear with the full signature in the binder listing. A cross reference of mangled versus demangled names is also provided.

For DLLs, the binder collects static DLL initialization information across multiple compile units. It then generates a function descriptor in the Writable Static Area for each DLL-referenced function, and generates a variable descriptor for each DLL-referenced variable. It accepts IMPORT control statements in its input to resolve dynamically linked symbols, and generates an IMPORT control statement for each exported function and variable.

The C++ compiler may generate internal symbols that are marked as exported. These symbols are for use by the run-time environment only and are not required by any user code. When these symbols are generated, if the binder option is DYNAM=DLL and the definition side-deck is not defined for the binder, the binder issues a message indicating the condition. If you are not building a DLL, you can use DYNAM=NO or you can ignore the message; or you can define a dummy side deck for the binder and then ignore the generated side deck.

Note: When binding a DLL in z/OS UNIX System Services, specify **-Wl,DLL** on the command line.

The z/OS UNIX file system support allows a library search of archive libraries that were created with the ar utility. UNIX files can be specified on binder control statements or specified directly on the compiler invocation command line.

C/C++ code is rebindable, provided all the sections are named. You can use the CSECT compiler option or the **#pragma csect** directive to name a section. If the GOFF option is active, then your CSECTs will automatically be named. See "CSECT | NOCSECT" on page 88.

© Copyright IBM Corp. 1998, 2009

Note: If you do not name all the sections and you try to rebind, the binder cannot replace private or unnamed sections. The result is a permanent accumulation of dead code and of duplicate functions.

The RENAME control statement may rename specified unresolved function references to a definition of a different name. This is especially helpful when matching function names that should be case insensitive. The RENAME statement does not apply to rebinds. If you rebind updated code with the original name, you will need another RENAME control statement to make references match their definitions.

The binder starts its processing by reading object code from primary input (DD SYSLIN). It accepts the following inputs:
- Object modules (compiler output from C/C++ and other languages)
- Load modules (previously link-edited by the Linkage-Editor)
- Program Objects (previously bound by the binder)
- Binder control statements
- Generalized Object File Format (GOFF) files

During the processing of primary input, control statements can control the binder processing. For example, the INCLUDE control statement will cause the binder to read and include other code.

Among other processing, the binder records whether or not symbols (external functions and variables) are currently defined. During the processing of primary input, the AUTOCALL control statement causes a library to be immediately searched for members that contain a definition for an unresolved symbol. If such a member is found, the binder reads it as autocall input before it processes more primary or secondary input.

After the binder processes primary input, it searches the libraries that are included in DD SYSLIB for definitions of unresolved symbols, unless you specified the options NOCALL or NORES. This is final autocall processing. The binder may read library members that contain the sought definition as autocall input.

Final autocall processing drives DD SYSLIB autocall resolution one or two times. After the first DD SYSLIB autocall resolution is complete, symbols that are still unresolved are subject to renaming. If renaming is done, DD SYSLIB autocall is driven a second time to resolve the renamed symbols.

After the binder completes final autocall (if autocall takes place), it processes the IMPORT control statements that were read in to match unresolved DLL type references. It then marks those symbols as being resolved from DLLs.

Finally, the binder generates an output program object. It stores the program object in a z/OS UNIX file, or as a member of the program library (PDSE) specified on the DD SYSLMOD statement. The Program Management Loader can load this program object into virtual storage to be run. The binder can generate a listing. It can also generate a file of IMPORT control statements for symbols exported from the program that are to be used to build other applications that use this DLL.

Linkage considerations

The binder will check that a statically bound symbol reference and symbol definition have compatible attributes. If a mismatch is detected, the binder will issue a diagnostic message. This attribute information is contained within the binder input files, such as object files, program objects, and load modules.

For C and C++, the default attribute is based on the XPLINK, NOXPLINK, LP64 and ILP32 options.

The attributes can also be set for assembly language. Refer to the *HLASM Language Reference*, SC26-4940 for further information.

Primary input processing

The binder obtains its primary input from the contents of the data sets that are defined by the DD SYSLIN.

Primary input to the binder can be a sequential data set, a member of a partitioned data set, or an instream data set. The primary input must consist of one or more separately compiled program objects, object modules, load modules or binder control statements.

C or C++ object module as input

The binder accepts object modules generated by the XL C or XL C++ compiler (as well as other compilers or assemblers) as input. All initialization information and relocation information for both code and the Writable Static Area is retained, which makes each compile unit fully rebindable.

Secondary input processing

Secondary input to the binder consists of files that are not part of primary input but are included as input due to the INCLUDE control statement.

The binder obtains its secondary input by reading the members from libraries of object modules (which may contain control statements), load modules, or program objects.

Load module as input

The binder accepts a load module that was generated by the Linkage-Editor input, and converts it into program object format on output.

Note: Object modules that define or refer to writable static objects that were processed by the prelinker and link-edited into a load module do not contain relocation information. You cannot rebind these compile units, or use them as input to the IPA Link step. See “Code that has been prelinked” on page 453 for more information on prelinked code and the binder.

Program object as input

The binder accepts previously bound program objects as input. This means that you can recompile only a changed compile unit, and rebind it into a program without needing other unchanged compile units. See “Rebind a changed compile unit” on page 406 and “Rebindability” on page 447.

You can compile and bind each compile unit to a program object, possibly with unresolved references. To build the full application, you can then bind all the separate program objects into a single executable program object.

Autocall input processing (library search)

The library search process is also known as *automatic library call*, or *autocall*. Unresolved symbols, including unresolved DLL-type references, may have their definitions within a library member that is searched during library search processing.

The library member that is expected to contain the definition is read. This may resolve the expected symbol, and also other symbols which that library member may define. Reading in the library member may also introduce new unresolved symbols.

Incremental autocall processing (AUTOCALL control statement)

Traditionally, autocall has been considered part of the final bind process. However, through the use of the AUTOCALL control statement, you can invoke autocall at any time during the include process.

The binder searches the libraries that occur on AUTOCALL control statements immediately for unresolved symbols and DLL references, before it processes more primary or secondary input. See *z/OS MVS Program Management: User's Guide and Reference* for further information on the AUTOCALL control statement. After processing the AUTOCALL statement, if new unresolved symbols are found that cannot be resolved from within the library being processed, the library will not be searched again. To search the library again, another AUTOCALL statement or SYSLIB must indicate the same library.

Final autocall processing (SYSLIB)

The binder performs final autocall processing of DD SYSLIB in addition to incremental autocall. It performs this processing after it completes the processing of DD SYSLIN.

DD SYSLIB defines the libraries of object modules, load modules, or program objects that the binder will search after it processes primary and secondary input.

The binder searches each library (PDS or PDSE) in the DD SYSLIB concatenation in order. The rules for searching for a symbol definition in a PDS or PDSE are as follows:
- If the library contains a C370LIB directory (`@@DC370$ or @@DC390$`) that was created using the C/C++ Object Library Utility, and the directory points to a member containing the definition for the symbol, that member is read.
- If the library has a member or alias with the same name as the symbol that is being searched, that member of the library is read.

You can use the LIBRARY control statement to suppress the search of SYSLIB for certain symbols, or to search an alternate library.

Non-XPLINK libraries

The libraries described here are to be used only for binding non-XPLINK program modules.

For C and C++, you should include CEE.SCEELKEX and CEE.SCEELKED in your DD SYSLIB concatenation when binding your program. Those libraries contain the

Language Environment resident routines, which include those for callable services, initialization, and termination. CEE.SCEELKED has the uppercase (NOLONGNAME), 8-byte-or-less versions of the standard C library routines, for example PRINTF and `@@PT@C`. CEE.SCEELKEX has the equivalent case-sensitive long-named routines; for example, `printf`, `pthread_create`.

For C++, you should also include the C++ base library in data set CEE.SCEECPP in your DD SYSLIB concatenation when binding your program. It contains the C++ base routines such as global operator `new`. The C++ class libraries are contained in the C128N member of the CEE.SCEELIB data set.

XPLINK libraries

The libraries described here are to be used only for binding XPLINK program modules.

For C and C++, you must include CEE.SCEEBND2 in your DD SYSLIB concatenation when binding your program. This library contains the Language Environment resident routines, which include those for initialization and termination.

XPLINK C run-time and C++ base libraries are packaged as DLLs. Therefore, the bindings for those routines resolve dynamically. This is accomplished by providing definition side-decks (object modules containing IMPORT control statements). This is done using INCLUDE control statements in the binder primary or secondary input. Language Environment side decks reside in the CEE.SCEELIB data set.

The Language Environment routine definitions for callable services are contained in the CELHS001 member of the data set CEE.SCEELIB. For example, CEEGTST is contained here.

The C run-time library routine definitions for 32-bit programs are contained in the CELHS003 member of the data set CEE.SCEELIB, which contains NOLONGNAME and case-sensitive long-named routines (for example, `printf`, PRINTF, and `pthread_create` are contained here). It also contains the C run-time library global variables; for example, `environ`.

For 32-bit C++ programs, you should also include the C++ base library side deck (member CELHSCPP in data set CEE.SCEELIB). It contains the C++ base routines such as global operator `new`.

The C++ class libraries are contained in the C128 member of the CEE.SCEELIB data set.

LP64 libraries

The libraries described in this topic are to be used only for binding LP64 program modules. LP64 is built upon the XPLINK linkage, which means that:

- In the simple XPLINK case, you must include CEE.SCEEBND2 in your DD SYSLIB concatenation when binding your programs.
- The 64-bit C++ libraries are packaged as DLLs, so INCLUDE statements must be used to resolve C and C++ run-time references.
- The 64-bit side decks are in the CEE.SCEELIB dataset.
- For 64-bit modules, the C run-time library definitions are contained in the CELQS003 member of the CEE.SCEELIB data set.
- The C++ base library side deck member for 64-bit is the CELQSCPP member of the CEE.SCEELIB data set.

- The C++ class libraries are contained in the C64 member of the CEE.SCEELIB data set.
- There is no 64-bit equivalent for the CELHS001 member.

Rename processing

Rename processing is performed at the end of the first pass of final autocall processing of DD SYSLIB, when all possible references have been resolved with the names as they were on input. The binder renaming logic permits the conversion of unresolved non-DLL external function references and drives the final autocall process again.

The binder maps names according to the following hierarchy:
1. If the name has ever been mapped due to a pragma map in C++ code, the name is not renamed.
2. If the name has ever been mapped due to a pragma map in C code that was compiled with the LONGNAME option, the name is not renamed.
3. If a valid RENAME control statement was read for an unresolved function name, new-name specified on the applied RENAME statement is chosen, provided that old-name did not already appear on an applied RENAME statement as either a new or old name. Syntactically correct RENAME control statements that are not applied are ignored. See *z/OS MVS Program Management: User's Guide and Reference* for more information on RENAME control statements.
4. If the name corresponds to a Language Environment function, the binder may map the name according to C/C++ run-time library rules.
5. If the UPCASE(YES) option is in effect and the name is 8 bytes or less, and not otherwise renamed by any of the previous rules, the name chosen is the same name but with all alphabetic characters mapped to uppercase, and '_' mapped to '@'. The binder maps names with the initial characters IBM, CEE, or PLI to initial characters of IB$, CE$, and PL$, respectively. All names that are different only in case will map to the same name.

If renamed, the original name is replaced. The original name and the generated new name appear in the rename table of the binder listing. See "Renamed Symbol Cross Reference" on page 439.

Generating aliases for automatic library call (library search)

For library search purposes, a member of a library (PDS, PDSE, or archive) can be an object module, a load module, or a program object. It has one member name, but may define multiple symbols (variables or functions) within it. To make library search successful, you must expose these defined symbols as aliases to the binder. When the binder searches for an unresolved reference, it can find, through the member name or an alias, the member which contains the definition. It then reads that member.

You can create aliases in the following ways:
- ALIAS binder control statement
- ALIASES(ALL) binder option
- ar utility for object module archives
- EDCALIAS utility for object module PDS and PDSEs

Note: Aliases that the EDCALIAS utility generates are supported only for migration purposes. Use the EDCALIAS utility only if you need to provide autocall libraries to both prelinker and binder users. Otherwise, you should use the ALIASES(ALL) option, and bind separate compile units.

Dynamic Link Library (DLL) processing

The binder supports the code that is generated by C++, and by C with the DLL compiler option, as well as code that is generated by C and C++ with the XPLINK option. Code generated with the XPLINK compiler option, like code generated by C++ and code generated by C with the DLL option, is always DLL-enabled (that is, references can be satisfied by IMPORT control statements). The binder option DYNAM(DLL) controls DLL processing. You must specify DYNAM(DLL) if the program object is to be a DLL, or if it contains DLL-type references. This topic assumes that you specified the DYNAM(DLL) option. See *z/OS MVS Program Management: User's Guide and Reference* for more information on the DYNAM(DLL) binder option. You must also specify CASE(MIXED) in order to preserve the case sensitivity of symbols on IMPORT control statements.

If you are building an application that imports symbol definitions from a DLL, you must include an IMPORT control statement for each symbol to which your application expects to dynamically link. Typically, the input to your bind step for your application should include the definition side-deck of IMPORT control statements that the binder generated when the DLL was built. For compatibility, the binder accepts definition side-decks of IMPORT control statements that the Language Environment Prelinker generated. To use the definition side-decks that are distributed with IBM Class libraries, you must specify the binder option CASE(MIXED).

After final autocall processing of DD SYSLIB is complete, all DLL-type references that are not statically resolved are compared to IMPORT control statements. Symbols on IMPORT control statements are treated as definitions, and cause a matching unresolved symbol to be considered dynamically rather than statically resolved. A dynamically resolved symbol causes an entry in the binder class B_IMPEXP to be created. If the symbol is unresolved at the end of DLL processing, it is not accessible at run time.

Addresses of statically bound symbols are known at application load time, but addresses of dynamically bound symbols are not. Instead, the run-time library that loads the DLL that exports those symbols finds their addresses at application run time. The run-time library also fixes up the linkage blocks (descriptors) for the importer in C_WSA during program execution.

The binder builds tables of imported and exported symbols in the class B_IMPEXP, section IEWBCIE. This element contains the necessary information about imported and exported symbols to support run-time library dynamic linking and loading.

Statically bound functions

For each DLL-referenced function, the binder will generate a function linkage block (descriptor) of the same name as a part in the class C_WSA.

Some of the linkage descriptors for XPLINK code are generated by the compiler rather than the binder. Compiler-generated descriptors are not visible as named entities at bind time. For XPLINK:

- Functions, which are referenced exclusively in the compilation unit, have descriptors which are generated by the compiler and have no visible names.
- Functions, which are possibly referenced outside of the compilation unit (either by function pointer, or because they are exported), have descriptors which are generated by Language Environment functions when the DLL is loaded. They are not part of C_WSA. There will be a pointer to the function descriptor in C_WSA.

- For all other DLL-referenced functions, function descriptors are generated by the binder as a part with the same name in the class C_WSA (with the exception that for NORENT compiles, the descriptor will be in B_DESCR rather than C_WSA).

All C++ code and XPLINK code generate DLL references. C code generates DLL references if you used the DLL compiler option. If a DLL reference to an external function is resolved at the end of final autocall processing, the binder generates a function linkage block of the same name in the Writable Static Area, and initializes it to point to the resolved function. If the DLL reference is to a static function, the binder generates a function linkage block with a private name, which is initialized to point to the resolved static function.

Imported variables

For each DLL-referenced external variable in C_WSA that is unresolved at the end of final autocall processing (DD SYSLIB), if a matching IMPORT control statement was read in, the variable is considered to be resolved via dynamic linking from the DLL named on the IMPORT control statement. The binder will generate a variable linkage block (descriptor) of the same name, as a part in the class C_WSA.

Imported functions

For each DLL-referenced external function that is unresolved at the end of final autocall processing, if a matching IMPORT control statement was read in, the function is considered to be resolved via dynamic linking from the DLL named on the IMPORT control statement. The binder will generate a function linkage block (descriptor) of the same name, as a part in the class C_WSA.

Output program object

The DD SYSLMOD defines where the binder stores its output program object. You can store the output program object in one of the following:

- A PDSE member, where the binder stores a single program object
- A PDSE where the binder stores its output program objects (one program object for each NAME control statement)
- A z/OS UNIX System Services file or directory

The PDSE must have the attribute RECFM=U.

Output IMPORT statements

The DD SYSDEFSD defines the output sequential data set where the binder writes out IMPORT control statements. The binder writes one control statement for each exported external symbol (function or variable), if you specify the option DYNAM(DLL). The data set must have the attributes `RECFM=F` or `RECFM=FB`, and `LRECL=80`.

You can mark symbols for export by using the **#pragma export** directive or the EXPORTALL compiler option, or the C++ `_Export` keyword.

Output listing

This topic contains an overview of the binder output listing. The binder creates the listing when you use the LIST binder option. It writes the listing to the data set that you defined by the DD SYSPRINT.

The listing consist of a number of categories. Some categories always appear in the listing, and others may appear depending on the options that you selected, or that were in effect.

Names that the binder generated appear as $PRIVxxxxxx rather than $PRIVATE. Private names that appear in the binder listing do not actually have that name in the program object. Their purpose in the listing is to permit association between various occurrences of the same private name within the listing. For purposes of rebindability, it is crucial that no sections have private names.

C++ names that appear in messages and listings are mangled names.

For the example listings in this topic, the files `USERID.PLAN9.OBJ(CU1)` and `/u/userid/plan9/cu2.o` were bound together using the JCL shown in Figure 38 on page 438. Figure 37 shows the corresponding source files:

```
/* file: USERID.PLAN9.C(CU1) */
/* compile with: LONGNAME RENT EXPORTALL CSECT("cu1")*/
#include <stdio.h>
int Ax=10;                                                  /* exported */
int ALongNamedThingVVWhichIsExported=11;      /* exported */
static int Az=12;
static int A1(void) {
   return Ax;
}
int ALongNamedThingFFWhichIsExported(void) {   /* exported */
  return Ax;
}
int A3(void) {             /* exported */
  return Ax + Az;
}
extern int b1(void);   /* statically bound, defined in plan9/cu2.C */
main() {
  int i;
  i = b1() + call_a3() + call_b1_in_cu2();
  printf("now returning\n");     /* printf statically bound from SCEELKEX */
  return i;
}
```

```
 /* file: cu2.C (C++ file) */
 /* compile with: CSECT(PROJ9) */
 extern b2(void);
 extern "C" c2(void);                                /* imported from DLLC */
 extern     c3(void);                                 /* imported from DLLC */
 extern "C" int b1(void) {                          /* called from cu1.c */
   return b2();
 }
 int b2(void) {
   return c2() + c3();
 }
```

Figure 37. Source files for listing example

```
//BIND1    EXEC CBCB,
//         BPARM='LIST(ALL),MAP,XREF',
//         OUTFILE='USERID.PLAN9.LOADE(HELLO1),DISP=SHR'
//INOBJ    DD DISP=SHR,DSN=USERID.PLAN9.OBJ
//SYSDEFSD DD DISP=SHR,DSN=USERID.PLAN9.IMP
//SYSPRINT DD DISP=SHR,DSN=USERID.PLAN9.LISTINGS(CU1CU2R)
//SYSLIN   DD *
 INCLUDE INOBJ(CU1)
 INCLUDE '/u/userid/plan9/cu2.o'
 IMPORT CODE,DLLC,c1
 IMPORT CODE,DLLC,c2
 IMPORT CODE,DLLC,c3__Fv
 RENAME 'call_a3' 'A3'
 RENAME 'call_b1_in_cu2' 'b1'
 ENTRY CEESTART
 NAME CU1CU2(R)
/*
```

Figure 38. Listing example JCL

Header

The heading always appears at the top of each page. It contains the product number, the binder version and release number, the date and the time the bind step began, and the entry point name. The heading also appears at the top of each section.

Input Event Log

This section is a chronological log of events that took place during the input phase of binding. The binder LIST option controls its presence. See *z/OS MVS Program Management: User's Guide and Reference* for more information on the LIST option.

```
IEW2278I B352 INVOCATION PARAMETERS - AMODE=31,MAP,RENT,DYNAM=DLL,CASE=MIXED,
COMPAT=CURR,ALIASES=ALL,LIST(ALL),MAP,XREF
IEW2322I 1220  1    INCLUDE INOBJ(CU1)
IEW2308I 1112 SECTION CEESTART HAS BEEN MERGED.
IEW2308I 1112 SECTION PROJ9#CU1#C HAS BEEN MERGED.
IEW2308I 1112 SECTION ALongNamedThingVVWhichIsExported HAS BEEN MERGED.
IEW2308I 1112 SECTION Ax HAS BEEN MERGED.
IEW2308I 1112 SECTION PROJ9#CU1#S HAS BEEN MERGED.
IEW2308I 1112 SECTION CEEMAIN HAS BEEN MERGED.
IEW2308I 1112 SECTION PROJ9#CU1#T HAS BEEN MERGED.
IEW2322I 1220  2    INCLUDE '/u/userid/plan9/cu2.o'
IEW2308I 1112 SECTION PROJ9#cu2.C#C HAS BEEN MERGED.
IEW2308I 1112 SECTION PROJ9#cu2.C#S HAS BEEN MERGED.
IEW2308I 1112 SECTION PROJ9#cu2.C#T HAS BEEN MERGED.
IEW2322I 1220  3    IMPORT CODE 'DLLC' 'c1'
IEW2322I 1220  4    IMPORT CODE 'DLLC' 'c2'
IEW2322I 1220  5    IMPORT CODE 'DLLC' 'c3__Fv'
IEW2322I 1220  6    RENAME 'call_a3' 'A3'
IEW2322I 1220  7    RENAME 'call_b1_in_cu2' 'b1'
IEW2322I 1220  8    ENTRY CEESTART
IEW2322I 1220  9    NAME CU1CU2(R)
 :
 :
```

Module Map

The Module Map is printed only if you specify the binder MAP option. It displays the attributes of each loadable binder class, along with the storage layout of the parts in that class.

For C/C++ programmers who use constructed reentrancy, two classes are of special interest: C_CODE and C_WSA. For LP64, the class names are C_CODE64 and C_WSA64. The C_CODE class exists if C++ code is encountered or if C code is compiled with LONGNAME or RENT. The C_WSA class exists if any defined writable static objects are encountered.

```
                         *** M O D U L E  M A P ***

---------------
CLASS  C_CODE            LENGTH =      5E4  ATTRIBUTES = CAT,   LOAD, RMODE=ANY
---------------

 SECTION    CLASS                                      ------- SOURCE --------
  OFFSET   OFFSET  NAME                TYPE    LENGTH  DDNAME   SEQ  MEMBER

                0  PROJ9#CU1#C        CSECT       330  INOBJ     01  CU1
       0        0     PROJ9#CU1#C        LABEL
      D0       D0     ALongName-ported   LABEL
     190      190     A3                 LABEL
     248      248     main               LABEL




---------------
CLASS  C_WSA             LENGTH =       68  ATTRIBUTES = MRG, DEFER , RMODE=ANY
---------------

      CLASS
     OFFSET  NAME               TYPE          LENGTH

          0  c3()               DESCRIPTOR      20
         20  c2                 DESCRIPTOR      20
         40  ALongName#000001   PART             4
         44  Ax                 PART             4
         48  $PRIV000011        PART            18
         60  $PRIV000014        PART             8
```

Data Set Summary

The Module Map ends with a Data Set Summary table, which associates input files with a corresponding ddname name and concatenation number.

The binder creates a dummy ddname for each unique z/OS UNIX System Services file when it processes path names from control statements. For example, on an INCLUDE control statement. The dummy ddname has the format ″/nnnnnnn″, where nnnnnnn is an integer assigned by binder, and appears in messages and listings in place of the z/OS UNIX file name.

```
          ***  DATA SET SUMMARY  ***

DDNAME    CONCAT   FILE IDENTIFICATION
/0000001    01     /u/userid/plan9/cu2.o
INOBJ       01     USERID.PLAN9.OBJ
SYSLIB      01     CEE.SCEELKEX
SYSLIB      02     CEE.SCEELKED
SYSLIB      03     CEE.SCEECPP
```

Renamed Symbol Cross Reference

The Renamed Symbol Cross Reference is printed only if a name was renamed for library search purposes, and you specified the MAP binder option.

The binder normally processes symbols exactly as received. However, it may remove certain symbolic references if they are not resolved by the original name during autocall. See "Rename processing" on page 434. During renaming, the original reference is replaced. Such replacements, whether resolved or not, appear in the Rename Table.

The rename table is a listing of each generated new name and its original old name.

```
***  RENAMED SYMBOL CROSS REFERENCE  ***
---------------------
RENAMED SYMBOL
      SOURCE SYMBOL
---------------------

A3
      call_a3

b1
      call_b1_in_cu2

***  END OF RENAMED SYMBOL CROSS REFERENCE  ***

*** E N D   O F   M O D U L E   M A P ***
```

Cross Reference Table

The listing contains a Cross-Reference Table of the program object if you specify the XREF binder option. Each line in the table contains one address constant in the program object. The left half of the table shows the location (OFFSET) and reference type (TYPE) within a defined part (SECT/PART) where a reference occurs. The right half of the table describes the symbol being referenced.

```
                                   C R O S S - R E F E R E N C E  T A B L E
                                   _________________________________________


TEXT CLASS = C_CODE

--------------- R E F E R E N C E --------------------------  T A R G E T -------------------------------
CLASS                          ELEMENT      |                                                ELEMENT
OFFSET SECT/PART(ABBREV)        OFFSET  TYPE | SYMBOL(ABBREV)   SECTION (ABBREV)              OFFSET CLASS NAME
                                             |
    68 PROJ9#CU1#C                  68 Q-CON | Ax               $NON-RELOCATABLE                  44 C_WSA
    70 PROJ9#CU1#C                  70 A-CON | CEESTART         CEESTART                           0 B_TEXT
   138 PROJ9#CU1#C                 138 Q-CON | Ax               $NON-RELOCATABLE                  44 C_WSA
   204 PROJ9#CU1#C                 204 Q-CON | $PRIV000011      $NON-RELOCATABLE                  48 C_WSA
   208 PROJ9#CU1#C                 208 Q-CON | Ax               $NON-RELOCATABLE                  44 C_WSA
   2E4 PROJ9#CU1#C                 2E4 Q-CON | $PRIV000011      $NON-RELOCATABLE                  48 C_WSA
   2E8 PROJ9#CU1#C                 2E8 V-CON | b1               PROJ9#cu2.C#C                      0 C_CODE
   2EC PROJ9#CU1#C                 2EC V-CON | A3               PROJ9#CU1#C                      190 C_CODE
   2F0 PROJ9#CU1#C                 2F0 V-CON | b1               PROJ9#cu2.C#C                      0 C_CODE
   2F4 PROJ9#CU1#C                 2F4 V-CON | printf           printf                             0 B_TEXT
   33C CEEMAIN                       4 A-CON | main             PROJ9#CU1#C                      248 C_CODE
   340 CEEMAIN                       8 A-CON | EDCINPL          EDCINPL                            0 B_TEXT
   3C8 PROJ9#cu2.C#C                78 V-CON | b2()             PROJ9#cu2.C#C                     E0 C_CODE
   3D0 PROJ9#cu2.C#C                80 A-CON | CEESTART         CEESTART                           0 B_TEXT
   4CA PROJ9#cu2.C#C               17A Q-CON | $PRIV000014      $NON-RELOCATABLE                  60 C_WSA
   588 PROJ9#cu2.C#C               238 Q-CON | $PRIV000014      $NON-RELOCATABLE                  60 C_WSA
   58C PROJ9#cu2.C#C               23C Q-CON | c2               $NON-RELOCATABLE                  20 C_WSA
   590 PROJ9#cu2.C#C               240 Q-CON | c3()             $NON-RELOCATABLE                   0 C_WSA
```

Imported and Exported Symbols Listing

The Imported and Exported Symbols Listing is part of the Module Summary Report, and is printed before other module summary information. This section will not appear if you do not specify the DYNAM(DLL) option, or if you are not importing or exporting any symbols.

This section follows the cross-reference table in the binder map. The listing shows the imported or exported symbols, and whether they name code or data. It also shows the DLL member name for imported symbols.

Descriptors are identified as such in the listing. One of the following generates an object module that exports symbols:

- Code that is compiled with the C, C++, or COBOL EXPORTALL compiler option
- C/C++ code that contains the **#pragma export** directive
- C++ code that contains the `_Export` keyword

The listing format is shown below. All imported symbols appear first, followed by all exported symbols. Within each group, symbol names appear in alphabetical order. There are some differences between the two groups:

- The member name or z/OS UNIX System Services file name for IMPORT is derived from the IMPORT control statement.
- The member name for exports is always the same as the DLL member name and does not appear in the listing.
- Symbol and member names that are longer than 16 bytes are abbreviated in the listing, using a hyphen. If there are duplicates, they are abbreviated using a number sign and a number. The abbreviation table shows the mapping from the abbreviated names to the actual names. See “Long Symbol Abbreviation Table” on page 444.

In the example below, you can see that `c2` and `c3` are to be dynamically linked from a DLL named DLLC. Also, this program exports variables `Ax` and `ALongNamedThingVVWhichIsExported`, and functions `A3` and `ALongNamedThingFFWhichIsExported`.

```
  ***  I M P O R T E D   A N D   E X P O R T E D   S Y M B O L S  ***

 IMPORT/EXPORT     TYPE    NAME                MEMBER
 -------------     ----    ----------------    ----------------
    IMPORT         CODE    c2                  DLLC
    IMPORT         CODE    c3()                DLLC

    EXPORT         DATA    Ax
    EXPORT         CODE    ALongName-ported
    EXPORT         DATA    ALongName#000001
    EXPORT         CODE    A3

                              *** END OF IMPORT/EXPORT ***
```

Mangled to Demangled Symbol Cross Reference

The Mangled to Demangled Symbol Cross Reference table is similar to the rename table. It cross-references demangled C++ names in object modules with their corresponding mangled names.

Note: Mangling is name encoding for C++, which provides type safe linkage. Demangling is decoding of a mangled name into a human readable format.

```
  ***  SHORT MANGLED NAMES ***
  ---------------------
  MANGLED NAME
        DE-MANGLED NAME
  ---------------------

  b2__Fv
        b2()

  c3__Fv
        c3()

  ***  END OF MANGLED TO DEMANGLED CROSS REFERENCE ***
```

The following example is for long mangled names.

```
              **  A B B R E V I A T I O N  /  D E M A N G L E D   N A M E S **

ABBR/MANGLE NAME    LONG SYMBOL


__javCls1-ension            :=
__javCls18_java/awt/Dimension
  $$DEMANGLED$$    ==    java.awt.Dimension

__javCls1-nuItem          :=
__javCls17_java/awt/MenuItem
  $$DEMANGLED$$    ==    java.awt.MenuItem

__jav15_j-ame()V          :=
__jav15_java/awt/Button9_buildName()V
  $$DEMANGLED$$    ==    void java.awt.Button.buildName()
```

Processing Options

The Processing Options section of the module summary lists values of the binder options that were in effect during the bind process.

```
PROCESSING OPTIONS:

   ALIASES            ALL
   ALIGN2             NO
   AMODE              31
   CALL               YES
   CASE               MIXED
   COMPAT             PM3
   DCBS               NO
   DYNAM              DLL
   :
   :
   ***END OF OPTIONS***
```

Save Operation Summary

The Save Operation Summary for a save to a program object lists the `blocksize` of the target PDSE. If you specified DYNAM(DLL), and are exporting symbols, the Save Operation Summary shows the data set name or the z/OS UNIX System Services path name of the side file. For example:

```
SAVE OPERATION SUMMARY:

   MEMBER NAME         CU1CU2
   LOAD LIBRARY        USERID.PLAN9.LOADE
   PROGRAM TYPE        PROGRAM OBJECT(FORMAT 3)
   VOLUME SERIAL       M06001
   DISPOSITION         REPLACED
   TIME OF SAVE        11.13.40  JUN  3, 1997
   SIDEFILE            USERID.PLAN9.IMP(CU1CU2)
```

Save Module Attributes

The Save Module Attributes section displays the attributes of the program object. These attributes are saved in the PDSE directory along with the program name, or saved in the z/OS UNIX System Services file.

```
SAVE MODULE ATTRIBUTES:

   AC                  000
   AMODE                31
   DC                  NO
   EDITABLE            YES
   EXCEEDS 16MB        NO
   EXECUTABLE          YES
   MIGRATABLE          NO
   OL                  NO
   OVLY                NO
   PACK,PRIME          NO,NO
   PAGE ALIGN          NO
   REFR                NO
   RENT                YES
   REUS                YES
   RMODE               ANY
   SCTR                NO
   SSI
   SYM GENERATED       NO
   TEST                NO
   XPLINK              NO
   MODULE SIZE (HEX)   00001360
```

Entry Point and Alias Summary

The Entry Point and Alias Summary will show an entry type of ″HIDDEN″ for hidden aliases. Hidden aliases may not be visible to some system utilities, and are marked as ″not executable″, to prevent an unintentional load and execution. They are for autocall purposes only. If you specify the option ALIASES(ALL), the binder generates hidden aliases.

```
ENTRY POINT AND ALIAS SUMMARY:

NAME:            ENTRY TYPE AMODE C_OFFSET CLASS NAME        STATUS

CEESTART          MAIN_EP      31 00000000 B_TEXT
b1                HIDDEN          00000350 C_CODE           REASSIGNED
b2()              HIDDEN          00000430 C_CODE           REASSIGNED
main              HIDDEN          00000248 C_CODE           REASSIGNED
Ax                HIDDEN          00000044 C_WSA            REASSIGNED
ALongName-ported  HIDDEN          000000D0 C_CODE           REASSIGNED
ALongName#000001  HIDDEN          00000040 C_WSA            REASSIGNED
A3                HIDDEN          00000190 C_CODE           REASSIGNED
CEEMAIN           HIDDEN          00000338 C_CODE           REASSIGNED
PROJ9#cu2.C#C     HIDDEN          00000350 C_CODE           REASSIGNED
PROJ9#cu2.C#S     HIDDEN          000005D8 C_CODE           REASSIGNED
PROJ9#cu2.C#T     HIDDEN          000005E0 C_CODE           REASSIGNED
PROJ9#CU1#C       HIDDEN          00000000 C_CODE           REASSIGNED
PROJ9#CU1#S       HIDDEN          00000330 C_CODE           REASSIGNED
PROJ9#CU1#T       HIDDEN          00000348 C_CODE           REASSIGNED

                         ***** E N D   O F   R E P O R T *****
```

Long Symbol Abbreviation Table

The Long Symbol Abbreviation Table lists symbol names that do not fit in the space that is allocated to them in the listing. This is a cross reference of abbreviations to the actual name. The abbreviation table is printed for symbols greater than 16 bytes in length, if you specify the MAP(YES) and XREF(YES) binder options.

```
*** L O N G   S Y M B O L   A B B R E V I A T I O N   T A B L E ***

     ABBREVIATION        LONG SYMBOL

     ALongName-ported := ALongNamedThingFFWhichIsExported
     ALongName#000001 := ALongNamedThingVVWhichIsExported


*** E N D   O F   L O N G   S Y M B O L   A B B R E V .   T A B L E ***
```

DDname vs Pathname Cross Reference Table

This section appears only if you specified path names on control statements.

The binder creates a dummy ddname for each unique z/OS UNIX System Services file when it processes z/OS UNIX file system path names from control statements. For example, on an INCLUDE control statement. The dummy ddname has the format ″/nnnnnnn″, where nnnnnnn is an integer assigned by the binder. The integer nnnnnnn appears in messages and listings in place of the z/OS UNIX file name.

The DDname vs Pathname Cross Reference Table shows the correspondence between the dummy ddname and its corresponding z/OS UNIX file name. The table appears only if there is a generated ddname. Pathnames that you specified on JCL have user-assigned ddnames, and do not appear in this table. The following is the format of the DDname vs Pathname Cross Reference Table.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
|D D N A M E   V S   P A T H N A M E   C R O S S   R E F E R E N C E   |
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

 DDNAME   PATHNAME
--------  --------------------------------------------------------------

/0000001  /u/userid/plan9/cu2.o

                          *** END OF DDNAME VS PATHNAME ***
```

Message Summary Report

The binder generates a Message Summary Report at the conclusion of each bind operation. The summary contains information on the types and severity of the messages that were issued during the bind process. You can search other parts of the listing to find where the messages were issued.

```
----------------------
MESSAGE SUMMARY REPORT
----------------------
 SEVERE MESSAGES        (SEVERITY = 12)
 NONE

 ERROR MESSAGES         (SEVERITY = 08)
 NONE

 WARNING MESSAGES       (SEVERITY = 04)
 NONE

 INFORMATIONAL MESSAGES (SEVERITY = 00)
 2008  2278  2308  2322


 **** END OF MESSAGE SUMMARY REPORT ****
```

Binder processing of C/C++ object to program object

The binder recognizes C/C++ object modules and performs special processing for them.

C/C++ categorizes reentrant programs as natural or constructed. The binder supports both natural reentrancy and C/C++ constructed reentrancy. However, programs that contain constructed reentrancy need additional run-time library for support while executing.

C code is naturally reentrant if it contains no data in the Writable Static Area. Modifiable data can be one of the following:
- External variables
- Static variables
- Writable strings
- DLL linkage blocks (descriptors) for variables
- DLL linkage blocks (descriptors) for functions

C++ code always has DLL type references for all function references that require a function descriptor in C_WSA. This means that all C++ programs are made reentrant via constructed reentrancy.

Programs with constructed reentrancy have two areas:
- A modifiable area that contains modifiable objects, seen in the binder class C_WSA
- A constant or reentrant area that contains executable code and constant data, seen in the binder classes B_TEXT or C_CODE.

Each user running the program receives a private copy of the C_WSA demand load class, which is mapped by the binder and is loaded by the run-time library. Multiple spaces or sessions can share the second part only if it is installed in the link pack area (LPA) or extended link pack area (ELPA). You must install PDSEs dynamically in the LPA.

To generate reentrant C/C++ code, follow these steps:
1. Compile your source files to generate code with constructed reentrancy as follows:
   - Compile your C source files with the RENT compiler option to generate code with constructed reentrancy.
   - Compile your C++ source files with whatever options you require. The compiler will generate C++ code with constructed reentrancy.
2. Use the binder to combine all input object modules into a single output program object.

Each compile unit maps to a number of sections, which belong to the C_CODE, C_WSA, or B_TEXT binder classes. Named binder sections may be replaced and make the code potentially rebindable. You can name your C/C++ sections with either the CSECT compiler option, or with the use of the **#pragma csect** directive. The name of a section should not be the same as one of your functions or variables, as this will cause duplicate symbols.

Each section owns one or more parts. The names of the parts are the names that resolve references. The names of functions appear as labels, which also resolve references. Some parts that are owned by a section may be unnamed. Each part belongs to a binder class.

Each externally named object in the Writable Static Area appears as a part that is owned by a section of the same name in the program object. Such parts belong to the C_WSA binder class. The binder section that owns an object also owns the initialization information for the object in the Writable Static Area. A rebind replaces this initialization information.

The code parts belong to the binder class of C_CODE or B_TEXT. The code parts consist of assembly instructions, constants and literals, and potentially read only variables that are not in the Writable Static Area. The following example will produce two sections, `i` and `CODE1`:

```
#pragma code(csect,"CODE1")
int i=10;
int foo(void) { return i; }
```

- The section named `i` is in class C_WSA, and has associated with it the initialization information to initialize `i` to 10.
- The section named `CODE1` is in class C_CODE, and has associated with it the entry point for function `foo()` and the machine instructions for the function.

When rebound, both sections `i` and `CODE1` are replaced along with any information that is associated with them.

The names in the C_WSA class and in the C_CODE class are in the same namespace. A variable and a function cannot have the same name.

C++ constructor calls and destructor calls that need to be collected across compile units are collected in the class C_@@STINIT.

DLL initialization information, which needs to be collected across compile units, is collected in the class C_@@DLLI.

Note: The information in this section is applicable to GOFF object modules and is not applicable to XOBJ.

Rebindability

If the binder processes duplicate sections, it keeps *only the first one*. This feature is particularly important when rebinding. You must include the changed parts first and the old program object second. This is how you replace the changed sections.

The binder can process each object module separately so that you only need to recompile and rebind the modules that you have modified. You do not need to recompile or include the object module for any unchanged modules.

When the binder replaces a named section, it also replaces all of its parts (named or unnamed). If a section does not have the name you desire, you can change it with the **#pragma csect** directive or with the CSECT compiler option. Unnamed parts typically come from the following:

- Unnamed modifiable static parts in C_WSA (static variables, strings)
- Unnamed static parts in C_CODE that may not be modifiable (static variables, strings)
- Unnamed code, static, or test part in C_CODE

You should name all sections if you want to rebind. If a section is unnamed (has a private name) and you attempt to replace it on a rebind, the unnamed section is not replaced by the updated but corresponding unnamed section. Instead, the binder keeps both the old and new unnamed sections, causing the program module to grow in size. All references to functions that are defined by both the old section and the new section are resolved first to functions in the new section. The program may run correctly, but you will get warnings about duplicate function definitions at bind time. These duplicates will never go away on future rebinds because you cannot replace or delete unnamed sections. You will also accumulate dead code in the duplicate functions which can never be accessed. This is why it is important to name all sections if you want to rebind your code.

Example: Suppose that our DLL consists of two compile units, `cu3.c` and `cu4.c`, that are bound using the JCL in Figure 39 on page 448:

```
/* file: cu3.c */
/* compile with: LONGNAME RENT EXPORTALL*/
#pragma csect(code,"CODE3")
func3(void) { return 4; }
int int3 = 3;
```

```
/* file: cu4.c */
/* compile with: LONGNAME RENT EXPORTALL */
#pragma csect(code,"CODE4")
func4(void) { return 4; }
int int4 = 4;
```

```
//BIND1    EXEC CBCB,
//         BPARM='CALL,MAP,DYNAM(DLL)',
//         OUTFILE='USERID.PLAN9.LOADE,DISP=SHR'
//INOBJ    DD DISP=SHR,DSN=USERID.PLAN9.OBJ
//SYSLIN   DD *
 INCLUDE INOBJ(CU3)
 INCLUDE INOBJ(CU4)
 ENTRY CEESTART
 NAME BADEXE(R)
/*
```

Figure 39. JCL to bind cu3.c and cu4.c

Later, you discover that `func3` is in error and should return 3. Change the source code in `cu3.c` and recompile. Rebind as follows:

```
//BIND1    EXEC CBCB,
//         BPARM='LIST(ALL),CALL,XREF,LET,MAP,DYNAM(DLL)',
//         OUTFILE='USERID.PLAN9.LOADE,DISP=SHR'
//INOBJ    DD DISP=SHR,DSN=USERID.PLAN9.OBJ
//INPOBJ   DD DISP=SHR,DSN=USERID.PLAN9.LOADE
//SYSLIN   DD *
 INCLUDE INOBJ(CU3)
 INCLUDE SYSLMOD(BADEXE)
 ENTRY CEESTART
 NAME GOODEXE(R)
/*
```

The input event log in the binder listing shows:

```
IEW2322I 1220  1    INCLUDE INOBJ(CU3)
IEW2308I 1112 SECTION CODE3 HAS BEEN MERGED.
IEW2308I 1112 SECTION int3 HAS BEEN MERGED.
IEW2322I 1220  2    INCLUDE INPOBJ(BADEXE)
IEW2308I 1112 SECTION CODE4 HAS BEEN MERGED.
IEW2308I 1112 SECTION int4 HAS BEEN MERGED.
IEW2308I 1112 SECTION CEESTART HAS BEEN MERGED.
IEW2308I 1112 SECTION CEESG003 HAS BEEN MERGED.
IEW2308I 1112 SECTION CEEBETBL HAS BEEN MERGED.
IEW2308I 1112 SECTION CEEBPUBT HAS BEEN MERGED.
IEW2308I 1112 SECTION CEEBTRM HAS BEEN MERGED.
IEW2308I 1112 SECTION CEEBLLST HAS BEEN MERGED.
IEW2308I 1112 SECTION CEEBINT HAS BEEN MERGED.
IEW2308I 1112 SECTION CEETGTFN HAS BEEN MERGED.
IEW2308I 1112 SECTION CEETLOC HAS BEEN MERGED.
IEW2322I 1220  3    ENTRY CEESTART
IEW2322I 1220  4    NAME GOODEXE(R)
```

`BADEXE` defines sections `int3`, `CODE3`, `int4`, and `CODE4`. If the binder sees duplicate sections, it uses the first one that it reads. Since `CU3` defines sections `CODE3` and `int3`, and is included before `BADEXE`, both sections are replaced by the newer ones in `CU3` when program object `GOODEXE` is created.

DLL considerations

Any IMPORT control statements used in the original bind must also be input to the re-bind, unless the dynamic resolution information is available via an INCLUDE statement.

Error recovery

This topic describes common errors in binding.

Unresolved symbols

Inconsistent reference vs. definition types

A common error is to compile one part of the code with RENT and another with NORENT. A RENT type reference (Q-CON in the binder listing) must be resolved by a Writable Static Area definition of a PART or a DESCRIPTOR in class C_WSA. A NORENT reference (V-CON or A-CON in the binder listing) must be resolved by CSECT or a LABEL typically in class C_CODE or B_TEXT.

Check the binder map to ensure that objects appear as parts in the expected classes (C_CODE, B_TEXT, C_WSA ...).

Inconsistent name usage

Another problem is the case sensitivity of the symbol names. Objects in the Writable Static Area cannot be renamed, but unresolved function references may be renamed to find a definition of a different name. See “Rename processing” on page 434. Such inconsistencies arise from inconsistent usage of the LONGNAME and NOLONGNAME compiler options, and from multi-language programs that make symbol names uppercase.

Example: Compile the file `main.c` with the options LONG, NORENT, and `other.c` with the options NOLONG, RENT:

```
/* file: main.c */
/* compile with LONG, NORENT */
extern int I2;
extern int func2(void);
main() {
  int i;
  i = i2 + func2();
  return i;
}
```

```
/* file: other.c */
/* compile with NOLONG,RENT */
int I2 = 2;
int func2(void) { return 2; }
```

When you bind the object modules together, the following errors will occur:

- An inconsistent use of the RENT | NORENT C compiler option causes symbol `I2` to be unresolved. The definition of `I2` from `other.c` is a writable static object because of the RENT option. But a writable static object cannot resolve the reference to `I2` from `main.c` because it is a NORENT reference. The binder messages show:

  ```
  IEW2308I 1112 SECTION I2 HAS BEEN MERGED.

  IEW2456E 9207 SYMBOL I2 UNRESOLVED.
  ```

- An inconsistent use of the LONG | NOLONG C compiler option causes the symbol `func2` to be unresolved. The function definition in `other.c` is in uppercase because of the NOLONG option. But the reference to `func2` from `main.c` is in lowercase because of the LONG option. The binder listing shows that `FUNC2` is a LABEL, that is a defined entry point; yet the binder messages show:

  ```
  IEW2456E 9207 SYMBOL func2 UNRESOLVED.
  ```

Significance of library search order

The order in which the libraries in SYSLIB are concatenated is significant.

Example: Suppose that functions `f1()` and `f4()` are resolved from SYSLIB:

```
  /* file: unit0.c */
  extern int f1(void);  /* from member UNIT1 of library LIB1 */
  extern int f4(void);  /* from member UNIT2 of library LIB2 */
  int main() {
    int rc1, rc4;
    rc1 = f1();
    rc4 = f4();
    if (rc1 !=  1) printf("fail rc1 is %d-n", rc1);
    if (rc4 != 40) printf("fail rc1 is %d-n", rc4);
    return 0;
  }
```

SYSLIB defines the libraries USERID.LIB1 with members UNIT1 and UNIT2, and USERID.LIB2 with members of the same name but different contents.

The library members are compiled from the following:

```
/* member UNIT1 of library LIB1 */
int f1(void) { return 1; }

/* member UNIT2 of library LIB1 */
int f2(void) { return 2; }

/* member UNIT1 of library LIB2 */
int f1(void) { return 10; }

/* member UNIT2 of library LIB2 */
int f2(void) { return 20; }
int f3(void) { return 30; }
int f4(void) { return f2()*2; /* 40 */ }
```

When bound with ALIASES(ALL), or when the EDCALIAS utility is used, all defined symbols are seen in a library directory as aliases that indicate the library member that contains their definition.

There are two definitions of f1(), but library search of SYSLIB for f1 searches library LIB1 first, and finds alias f1 of member UNIT1. It reads in that member, and the call to f1() returns 1. Library search of SYSLIB for f4 searches LIB1 first, and does not find a definition. It then searches LIB2, and finds alias f4 of member UNIT2 of library LIB2. So UNIT2 of library LIB2 is read in resolving not only f4, but also f2 and f3, and the call to f4() returns 40. UNIT2 of library LIB1 is not read by mistake because an alias indicates not only the member name, but also the library in which that member resides.

If the order of LIB1 and LIB2 is reversed, LIB2 is searched first, and f1() is obtained from LIB2 instead.

If changing the library search order cannot work for you, use the LIBRARY control statement. See *z/OS MVS Program Management: User's Guide and Reference* for further information on the LIBRARY control statement.

Duplicates

If the binder processes duplicate sections, it keeps the first one and ignores subsequent ones, without giving a warning. This feature is used to replace named sections when rebinding by replacing only changed sections.

If the binder processes functions that have duplicate names, it keeps all definitions, but all references resolve to the first one. An exception is in the case of C++

template instantiation. The binder takes the first user-defined function (if any) of the same signature rather than the first compiler-generated definition via template instantiation.

Example: Compile the following source files `doit1.c` and `doit2.c`:

```
#include <stdio.h>
/* file: doit1.c */
int int1 = 1;
#pragma csect(code,"DO1")
int func2(void) { return 2; }
int func3(void) { return 3; }
extern int func4(void);
int main() {
  int i1,i2,i3,i4;
  i1 = int1;
  i2 = func2();
  i3 = func3();
  i4 = func4();
  printf("%d %d %d %d\n",i1,i2,i3,i4);
  return 0;
}
```

```
/* file: doit2.c */
int int1 = 11;
#pragma csect(code,"DO2")
int func3(void) { return 33; }
int func4(void) { return 44; }
```

Use the LONGNAME compiler option, and bind. The binder sections are `int1`, `DO1` and `int1`, `DO2`. The binder keeps one of the duplicate sections, `int1`, and does not issue a warning. But uniquely named sections contain the functions. Section `DO1` contains the functions `func2` and `func3`. Section `DO2` contains the functions `func3` and `func4`. The binder retains both sections `DO1` and `DO2`, but because both sections contain function `func3`, it issues a warning message as follows:

```
IEW2480W A711 EXTERNAL SYMBOL func3 OF TYPE LD WAS ALREADY DEFINED AS A
SYMBOL OF TYPE LD IN SECTION DO1.
```

It is easier to find the object code with the duplicate if you use multiple INCLUDE statements rather than DD concatenation.

Example: If you use:

```
  //INOBJ  DD DISP=SHR,DSN=USERID.PLAN9.OBJ
  //SYSLIN DD *
   INCLUDE INOBJ(DOIT1)
   INCLUDE INOBJ(DOIT2)
   ENTRY CEESTART
  /*
```

The members in the binder listing are separated logically. The messages in the binder listing are:

```
  :
  :
IEW2322I 1220  1    INCLUDE INOBJ(DOIT1)
IEW2308I 1112 SECTION CEESTART HAS BEEN MERGED.
IEW2308I 1112 SECTION DO1 HAS BEEN MERGED.
IEW2308I 1112 SECTION int1 HAS BEEN MERGED.
IEW2308I 1112 SECTION CEEMAIN HAS BEEN MERGED.
IEW2322I 1220  2    INCLUDE INOBJ(DOIT2)
IEW2480W A711 EXTERNAL SYMBOL func3 OF TYPE LD WAS ALREADY DEFINED AS A
SYMBOL OF TYPE LD IN SECTION DO1.
IEW2308I 1112 SECTION DO2 HAS BEEN MERGED.
```

From the informational messages, it is clear that section `DO1` is from `INOBJ(DOIT1)`, and that `DO2` is from `INOBJ(DOIT2)`.

Example: But if you use DD concatenation as follows:

```
  //INOBJ  DD DISP=SHR,DSN=USERID.PLAN9.OBJ
  //SYSLIN DD DISP=SHR,DSN=USERID.PLAN9.OBJ(DOIT1)
  //       DD DISP=SHR,DSN=USERID.PLAN9.OBJ(DOIT2)
  //       DD *
   ENTRY CEESTART
  /*
  :
  :
```

Now the messages are:

```
IEW2308I 1112 SECTION CEESTART HAS BEEN MERGED.
IEW2308I 1112 SECTION DO1 HAS BEEN MERGED.
IEW2308I 1112 SECTION int1 HAS BEEN MERGED.
IEW2308I 1112 SECTION CEEMAIN HAS BEEN MERGED.
IEW2480W A711 EXTERNAL SYMBOL func3 OF TYPE LD WAS ALREADY DEFINED AS A
SYMBOL OF TYPE LD IN SECTION DO1.
IEW2308I 1112 SECTION DO2 HAS BEEN MERGED.
```

It is no longer clear which input file defines which section, and this makes tracking down duplicates to the originating compile unit more difficult.

Duplicate functions from autocall

If a library member that is expected to contain the definition of a symbol is read, it may resolve the expected symbol. It may also resolve other symbols because the library member may define multiple functions. These unexpected definitions that are pulled in through library search may cause duplicates. Since you cannot always be sure which one of the duplicate symbols you will resolve with, you should remedy the situation that is causing the duplicate symbols.

Hunting down references to unresolved symbols

Unresolved requests generate error or warning messages in the binder listing. If a function or variable is unresolved at the end of binder processing, it can be resolved at a later rebind.

Example: If you did not expect a symbol to remain unresolved, you can look at the binder listing to see which parts reference the symbol. If your DD SYSLIN has a large concatenation, the input is logically concatenated before the binder processes it. Since the compile units are not logically separated, it is hard to tell which compile unit defines the part that has the reference; for example:

```
  //SYSLIN DD DISP=SHR,DSN=USERID.PLAN9.OBJ(MEM1)
  //       DD DISP=SHR,DSN=USERID.PLAN9.OBJ(MEM2)
  //       DD DISP=SHR,DSN=USERID.PLAN9.OBJ(MEM3)
```

Example: You should consider using multiple INCLUDE control statements, which will logically separate the compile units for the binder informational messages in the listing. You can then find the compile unit with the unresolved reference (similar to finding duplicate function definitions); for example:

```
//INOBJ  DD DISP=SHR,DSN=USERID.PLAN9.OBJ
//SYSLIN DD *
 INCLUDE INOBJ(DOIT1)
 INCLUDE INOBJ(DOIT2)
 ENTRY CEESTART
/*
```

Incompatible linkage attributes

The binder will check that a statically bound symbol reference and symbol definition have compatible attributes. If a mismatch is detected, the binder will issue a diagnostic message. This attribute information is contained within the binder input files, such as object files, program objects, and load modules.

For C and C++, the default attribute is based on the XPLINK and NOXPLINK options. Individual symbols can have a different attribute than the default by using the #pragma OS_UPSTACK, #pragma OS_DOWNSTACK, and #pragma OS_NOSTACK.

The attributes can also be set for assembly language. Refer to the *HLASM Language Reference*, SC26-4940 for further information.

Non-reentrant DLL problems

If you bind a DLL with the REUS(NONE) option, each load of the DLL causes a separate load of the code area and the data area (C_WSA). If you split a statically bound program into mutually dependent DLLs, you will probably not get the desired result. Function pointers that used to compare the same may not be the same anymore, because the multiple loads of a DLL have more than one copy of the function in memory.

The same is true for data. A separate copy of C_WSA is loaded. So, data objects that are exported from a DLL and modified are not seen as modified by the new program that uses the DLL. You should bind all DLLs with REUS(RENT), or REUS(SERIAL) so that a new C_WSA is loaded only once per enclave.

Code that has been prelinked

You cannot bind code that refers to objects in the Writable Static Area and has been prelinked, and code which refers to objects in the Writable Static Area and has not been prelinked, in the same program object. This is because the z/OS prelinker and the binder use different methods to manage the Writable Static Area. The z/OS prelinker removes relocation information about objects in the Writable Static Area, making them invisible to the binder. The binder keeps relocation information and manages the Writable Static Area in the binder class C_WSA.

Chapter 11. Running a C or C++ application

This information gives an overview of how to run z/OS XL C/C++ programs under z/OS batch, TSO, and z/OS UNIX System Services.

The Language Environment element provides a common run-time environment for C, C++, COBOL, PL/I, and FORTRAN. For detailed instructions on running existing and new z/OS XL C/C++ programs under the Language Environment run time, refer to *z/OS Language Environment Programming Guide*. *z/OS XL C/C++ Programming Guide* also describes how to run z/OS XL C/C++ programs in a CICS environment.

Setting the region size for z/OS XL C/C++ applications

Prior to running your applications, ensure that you have the required region size to run the compiler and to run your application.

Note: The minimum region size for invoking the compiler is 148 MB. Depending on your program and the degree of optimization you are using (for example, OPT(2) and IPA), you may require significantly more space.

If your installation does not change the IBM-supplied default limits in the IEALIMIT or IEFUSI exit routine modules, different values for the region size have the following results:

Region Size Value	Result
`0K` or `0M`	Provides the job step with all the storage that is available below and above 16 MB. The resulting size of the region below and above 16 MB is unpredictable.
< `0M` ≤ `REGION` < `16M`	Establishes the size of the private area below 16 MB. If the region size specified is not available below 16 MB, the job step terminates abnormally. The extended region size is the default value of 32 MB.
< `16M` ≤ `REGION` ≤ `32M`	Provides the job step all the storage available below 16 MB. The resulting size of the region below 16 MB is unpredictable. The extended region size is the default value of 32 MB.
< `32M` ≤ `REGION` < `2047M`	Provides the job step all the storage available below 16 MB. The resulting size of the region below 16 MB is unpredictable. The extended region size is the specified value. If the region size specified is not available above 16 MB, the job step abnormally terminates.

Assuming that you do not use your own IEFUSI exit to override this, a specification of `REGION=4M` provides 4 MB below 16 MB, and a default of 32 MB above 16 MB for a total of 36 MB of available virtual memory and not just 4 MB.

Specifying `REGION=40M` provides all available private virtual memory below 16 MB, most likely around 8 MB to 10 MB, and 40 MB above 16 MB for a total of around 48 MB. This means that a JCL change from `REGION=4M` to `REGION=40M` does not change the virtual storage available to the compiler from 4 MB to 40 MB, but rather from 36 MB to 48 MB. If the only storage use increase is above 16 MB, then the actual increase is 8 MB.

© Copyright IBM Corp. 1998, 2009

For information about using the IEFUSI installation exit to set process limits, see *z/OS UNIX System Services Planning*.

Running an application under z/OS batch

You must have the Language Environment Library SCEERUN available before you try to run your application under z/OS batch.

If your application was compiled using the XPLINK compiler option you must have the Language Environment Library SCEERUN2 available before you try to run your application under z/OS batch.

If your application was bound with the DLL Class Libraries, you must supply SCLBDLL2 at run time. As of z/OS V1R2, the version of the DLL library is in CBC.SCLBDLL2. The DLL data set(s) can be in the system libraries, your JOBLIB statement, or your CBC.SCLBDLL2 statement.

The search sequence for library files is in the following order: STEPLIB, JOBLIB, LINKPACK, and LINKLIST.

Specifying run-time options under z/OS batch

When you run a C or C++ application, you can override the default values for a set of z/OS XL C/C++ run-time options. These options affect the execution of your application, including its performance, its error-handling characteristics, and its production of debugging and tuning information.

For your application to recognize run-time options, either the EXECOPS compiler option, or the **#pragma runopts(execops)** directive must be in effect. The default compiler option is EXECOPS.

You can specify run-time options under z/OS batch as follows:

- In your JCL, in the PARM parameter of the EXEC statement. For more information, refer to “Specifying run-time options in the EXEC statement” on page 457.
- On the GPARM parameter of the cataloged procedures that are supplied by IBM. Refer to “Using cataloged procedures” on page 457.
- The **#pragma runopts** statement in your source code.
- The CEEUOPT facility that is provided by the Language Environment services.
- In the assembler user exit. For more information on the assembler user exit, refer to the *z/OS XL C/C++ Programming Guide*.

If EXECOPS is in effect, use a slash ’/’ to separate run-time options from arguments passed to the application. For example:

```
GPARM='STORAGE(FE,FE,FE)/PARM1,PARM2,PARM3'
```

Language Environment services interpret the character string that precedes the slash as run-time options. The character string following the slash is passed to the `main()` function of your application as arguments. If a slash does not separate the arguments, Language Environment services interpret the entire string as an argument.

If the NOEXECOPS option is in effect, none of the preceding run-time options will take effect. In fact, any arguments and options that you specify in the parameter string (including the slash, if present) are passed as arguments to the `main()` function. For a description of run-time options see "Specifying run-time options" on page 333.

You should establish the required settings of the options for all z/OS XL C/C++ programs that you execute on a production basis. Each time the program is run, the default run-time options that were selected during z/OS XL C/C++ installation apply, unless you override them by using one of the following:
- Coding a **#pragma runopts** directive in your source
- Creating a CEEUOPT csect with the CEEXOPT macro and linking this csect into the program module.
- Specifying run-time options in the EXEC or GPARM statements

Example: The following example shows you how to run your program under z/OS batch. Partitioned data set member `MEDICAL.ILLNESS.LOAD(SYMPTOMS)` contains your z/OS XL C/C++ executable program. The program was compiled with the EXECOPS compiler option in effect. If you want to use the run-time option RPTOPTS(ON), and to pass `TESTFUNCT` as an argument to the function, use the JCL stream as follows:

```
//JOBname  JOB...
//STEP1    EXEC  PGM=SYMPTOMS,PARM='RPTOPTS(ON)/TESTFUNCT'
   ⋮
//STEPLIB  DD    DSN=MEDICAL.ILLNESS.LOAD,DISP=SHR
//         DD    DSN=CEE.SCEERUN,DISP=SHR
```

Figure 40. Running your program under z/OS batch

Specifying run-time options in the EXEC statement

Example: You can specify run-time options in the PARM parameter of the EXEC statement as follows:

```
//[stepname] EXEC PGM=program_name,
//            PARM='[run-time options/][program parameters]'
```

Example: If you want to generate a storage report and run-time options report for the application `PROGRAM1`, specify the run-time option RPTOPTS(ON) as follows:

```
//GO1   EXEC PGM=PROGRAM1,PARM='RPTOPTS(ON) / '
```

Note that the run-time options that are passed to the main routine are followed by a slash (/) to separate them from program parameters.

Using cataloged procedures

You can use one of the following cataloged procedures that are supplied with the z/OS XL C/C++ compiler to run your program. Each procedure listed below includes an execution step:

For z/OS XL C programs:

EDCCBG	Compile, bind, and run a C program
EDCCLG	Compile, link, and run a C program
EDCCLG	Compile, pre-link, link, and run a C program
EDCQBG	Bind and run a 64-bit C program
EDCQCBG	Compile, bind, and run a 64-bit C program
EDCXCBG	Compile, bind, and execute an XPLINK C Program

For z/OS XL C++ programs:

CBCBG	Bind and run a C++ program

CBCCB	Compile, bind, and run a C++ program
CBCCLG	Compile, prelink, link, and run a C++ program
CBCG	Run a C++ program
CBCLG	Prelink, link, and run a C++ program
CBCQBG	Bind and run a 64-bit C++ program
CBCQCBG	Compile, bind, and run a 64-bit C++ program
CBCXBG	Bind and run an XPLINK C++ program
CBCXCBG	Compile, bind, and run an XPLINK C++ program
CBCXG	Run an XPLINK C++ program

For more information on these cataloged procedures, see Chapter 12, "Cataloged procedures and REXX EXECs," on page 465.

Example: If you are using an IBM-supplied cataloged procedure, you must specify the run-time options on the GPARM parameter of the EXEC statement. Ensure that the EXECOPS run-time option is in effect.

```
//STEP   EXEC EDCCBG,INFILE='...',
//            GPARM='STACK(10K)/'
```

Example: You can also use the GPARM parameter to pass arguments to the z/OS XL C/C++ `main()` function. Place the argument, preceded by a slash, after the run-time options; for example:

```
//GO  EXEC EDCCBG,INFILE=...,
//         GPARM='STACK(10K)/ARGUMENT'
```

Example: If you want to pass an argument without specifying run-time options and EXECOPS is in effect (this is the default), precede it with a slash; for example:

```
//GO  EXEC  EDCCBG,...GPARM='/ARGUMENT'
//GO  EXEC  ,...GPARM='/z/OS UNIX System Services file:/u/mike/cloudy.C'
```

Example: If you want to pass parameters which contain slashes, and you are not providing run-time options, you must precede the parameters with a slash, as follows:

```
//GO  EXEC  EDCCBG,...GPARM='/z/OS UNIX System Services file:/u/mike/cloudy.C'
```

See also "Specifying run-time options" on page 333.

Running an application under TSO

Before you run your program under TSO, you must have access to the run-time library CEE.SCEERUN. To ensure that you have access to the run-time library, do one of the following:

- If you are running under ISPF in the foreground, concatenate the libraries to ISPLLIB.
- Have your system programmer add the libraries to the LPALST or LPA.
- Have your system programmer add the libraries to the LNKLST.
- Have your system programmer change the LOGON PROC so the libraries are added to the STEPLIB for the TSO session.
- If your application was compiled using the XPLINK compiler option, you must have the Language Environment Library SCEERUN2 available before you try to run your application under TSO.

The TSO `CALL` command runs a load module under TSO. If *data-set-name* is the partitioned data set member that holds the load module, the command to load and run a specified load module is:

```
CALL 'data-set-name' ['parameter-string'];
```

For example, if the load module is stored in partitioned data set member `SAMPLE.CPGM.LOAD(TRICKS)`, and the default run-time options are in effect, run your program as follows:

```
CALL 'SAMPLE.CPGM.LOAD(TRICKS)'
```

If you specify the unqualified name of the data set, the system assumes the descriptive qualifier LOAD. If you do not specify a member name, the system assumes the name TEMPNAME.

You do not need to use the CALL command if the STEPLIB ddname includes the data set that contains your program. For example, you could call a program `PROG1` with two required parameters `PARM1` and `PARM2` from the command line:

```
PROG1 PARM1 PARM2
```

See the appropriate document listed in *z/OS Information Roadmap* for more information on STEPLIB.

Specifying run-time options under TSO

You can specify run-time options in a **#pragma runopts** directive or in the *parameter-string* of the TSO CALL command. The *parameter-string* contains two fields that are separated by a slash(/), and takes the form:

```
'[run-time options/][arguments to main]'
```

The first field is passed to the program initialization routine as a run-time option list; the second field is passed to the `main()` function.

To allow your application to recognize run-time options, EXECOPS must be in effect. You can specify your additional run-time options on the command line as follows: specify the options followed by a slash (/), followed by the parameters you want to pass to the `main()` function.

For example, to run a load module that is stored in the partitioned data set member `GINGER.HOURLY.LOAD(CHECK)`, with the run-time option RPTOPTS(ON), use the following command:

```
CALL 'GINGER.HOURLY.LOAD(CHECK)' 'RPTOPTS(ON)/'
```

If the NOEXECOPS compiler or run-time option is in effect, what you specify on the command line (including the slash, if present) is passed as arguments to the `main()` function. For a description of run-time options see “Specifying run-time options” on page 333.

If you want to pass your parameters as mixed case, you must use the ASIS run-time option. See “Passing arguments to the z/OS XL C/C++ application” for more information on passing mixed case parameters.

Passing arguments to the z/OS XL C/C++ application

The arguments passed to `main()` are `argc` and `argv`. `argc` is an integer whose value is the number of arguments that are given when the program is run. `argv` is an array of pointers to null terminated character strings, which contain the arguments for the program. The first argument is the name of the program being run on the TSO command line. For more information on `argc`, `argv`, and `main()` see “ARGPARSE | NOARGPARSE” on page 72 or the description in *z/OS XL C/C++ Language Reference*.

The case of the characters in `argv` depends on you invoked how your z/OS XL C/C++ program, as shown in the following table.

Table 42. Case sensitivity of arguments under TSO

How the z/OS XL C/C++ program is invoked	**Example**	**Case of argument**
As TSO command	`program args`	Mixed case (However, if you pass the arguments entirely in upper case, the argument will be changed to lower case.)
By CALL command (with or without ASIS)	`CALL program args`	Lower case
By CALL command with control arguments ASIS	`CALL program Args ASIS`	Mixed case (However, if you pass the arguments entirely in upper case, the argument will be changed to as lower case.)
By CALL command with control ASIS	`CALL program ARGS ASIS`	The arguments will be changed to lower case following ISO C standards.

Running an application under z/OS UNIX System Services

This information discusses how to run your z/OS UNIX System Services XL C/C++ application.

You must have the Language Environment Library SCEERUN available before you try to run your application under z/OS UNIX System Services. If your application was compiled using the XPLINK compiler option you must have the Language Environment Library SCEERUN2 available before you try to run your application under z/OS UNIX System Services. If your application was bound with the DLL Class Libraries, you must supply SCLBDLL2 at run time. As of z/OS V1R2, the version of the DLL library is in CBC.SCLBDLL2.

z/OS UNIX System Services application environments

You can run your z/OS UNIX System Services XL C/C++ application programs from the following environments:

- z/OS shell
- z/OS ISPF Shell (ISHELL)
- TSO/E

  To call an application program that resides in a z/OS UNIX file from the TSO/E READY prompt, you must use the BPXBATCH utility.
- z/OS batch

  To run an application program that resides in a z/OS UNIX file, you must use the BPXBATCH utility with the JCL EXEC statement.
- z/OS shell through z/OS batch or TSO

By using the IBM-supplied BPXBATCH program, you can run an application program that resides in a z/OS UNIX file. You supply the name of the program as an argument to the BPXBATCH program, which invokes the shell environment. The BPXBATCH runs under the z/OS batch environment or under TSO.

Specifying run-time options under z/OS UNIX System Services

When invoking a program from the z/OS shell, slash-separated run-time options arguments syntax is not used. All the arguments always go to the `main()` routine. Specify run-time options by using the exported environment variable _CEE_RUNOPTS. The run-time will only use _CEE_RUNOPTS if the EXECOPS option is in effect.

Restriction on using 24-bit AMODE programs

You cannot run a 24-bit AMODE z/OS XL C/C++ application program that resides in a z/OS UNIX System Services file. Any programs you intend to run from the file system must be 31-bit or 64-bit AMODE, problem program state, PSW key 8 programs. If you plan to run a 24-bit AMODE z/OS XL C/C++ program from within an application, ensure that the executable resides in a PDS or PDSE member.

Any new XL C/C++ applications you develop for z/OS UNIX System Services should be 31-bit or 64-bit AMODE.

Copying applications between a PDS and z/OS UNIX System Services

If you have a XL C/C++ application as a PDS member and want to place it in the z/OS UNIX file system, you can use the z/OS UNIX System Services TSO/E command OPUTX to copy the member into a z/OS UNIX file.

If you have a XL C/C++ application as a z/OS UNIX file and want to place it in a PDS, you can use the z/OS UNIX TSO/E command OGETX to copy the z/OS UNIX file into a PDS.

You can also bind directly into a data set member with the c89 or `c++` utility by specifying a data set member name on the **-o** option, as in:

```
c89 -o"//loadlib(foo)"
```

For a description of these commands, see Chapter 21, "c89 — Compiler invocation using host environment variables," on page 535. For examples of using these commands to copy data sets to z/OS UNIX files, see *z/OS UNIX System Services User's Guide*.

Running a data set member from the z/OS shell

If your z/OS UNIX System Services XL C/C++ program resides in data sets and you must run the executable member from within the shell, you can pass a call to the program to TSO/E. Type the TSO/E CALL command with the name of the executable data set member on the shell command line and press the TSO/E function key to pass the command to TSO/E. Alternatively, you can use the **tso** command from the shell. Just precede the CALL with **tso** on the command line and press the ENTER key.

When the program completes, the shell session is restored.

Running z/OS UNIX System Services applications under z/OS batch

Using the BPXBATCH utility

Use the IBM-supplied BPXBATCH program to run a XL C/C++ application under z/OS batch from a z/OS UNIX System Services file. You can invoke the BPXBATCH utility from TSO/E, or by using JCL. The BPXBATCH utility submits a batch job and performs an initial user login to run a specified program from the shell environment.

Before you invoke BPXBATCH, you must have the appropriate authority to read from and write to z/OS UNIX files. For writing program output such as error messages, you should allocate z/OS UNIX files to `stdout` and `stderr`. Allocate the standard files using the PATH options on the TSO/E ALLOCATE command or the JCL DD statement.

For more information on the BPXBATCH program, refer to Chapter 19, “BPXBATCH utility,” on page 527.

Invoking BPXBATCH from TSO/E

From TSO/E, you can invoke BPXBATCH several ways:
- From the TSO/E READY prompt
- From a CALL command
- From a REXX EXEC

Figure 41 shows a REXX EXEC that does the following:
1. Runs the application program `/myap/base_comp` from your user ID
2. Directs output to the file `/myap/std/my.out`
3. Writes error messages to the file `/myap/std/my.err`
4. Copies the output and error data to data sets

```
/* base_comp REXX exec */
"Allocate File(STDOUT) Path('/u/myu/myap/std/my.out')
         Pathopts(OWRONLY,OCREAT,OTRUNC) Pathmode(SIRWXU) Pathdisp(DELETE,DELETE)"
"Allocate File(STDERR) Path('/u/myu/myap/std/my.err')
         Pathopts(OWRONLY,OCREAT,OTRUNC) Pathmode(SIRWXU) Pathdisp(DELETE,DELETE)"

"BPXBATCH PGM /u/myu/myap/base_comp"

"Allocate File(output1) Dataset
('MYAPPS.STD(BASEOUT)')"
"Ocopy Indd(STDOUT) Outdd(output1) Text Pathopts(OVERRIDE)"

"Allocate File(output2) Dataset('MYAPPS.STD(BASEERR)')"
"Ocopy Indd(STDERR) Outdd(output2) Text Pathopts(OVERRIDE)"
```

Figure 41. REXX EXEC to Run a Program

To invoke BPXBATCH, enter the name of the REXX EXEC from the TSO/E READY prompt. When the REXX EXEC completes, the files allocated to `stdout` and `stderr` are deleted.

Invoking BPXBATCH using JCL

To invoke BPXBATCH using JCL, submit a job that executes an application program and allocates the standard files using DD statements. For example, to run the application program `/myap/base_comp` from your user ID, direct its output to the file `/myap/std/my.out`, write error messages to the file `/myap/std/my.err`, and code the JCL statements as follows:

```
//jobname  JOB ...
//stepname EXEC PGM=BPXBATCH,PARM='PGM /u/myu/myap/base_comp'
//STDOUT    DD PATH='/u/myu/myap/std/my.out',
//          PATHOPTS=(OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU
//STDERR    DD PATH='/u/myu/myap/std/my.err',
//          PATHOPTS=(OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU
```

Submitting a non-z/OS UNIX System Services executable to run under batch

If your program requires z/OS UNIX System Services, but has been link-edited into a load module (PDS member) or bound into a non-z/OS UNIX program object (PDSE member), it can be executed in the z/OS batch environment. Use the JCL EXEC statement to submit the executable to run under the batch environment. You must have the run-time option POSIX in effect, either as **#pragma runopts(POSIX(ON))**, or as PARM='POSIX(ON)/'.

Chapter 12. Cataloged procedures and REXX EXECs

This information describes the REXX EXECs (TSO) and cataloged procedures that the z/OS XL C/C++ compiler provides in conjunction with the Language Environment services, to call the various z/OS XL C/C++ utilities.

When you specify a data set name without enclosing it in single quotation marks ('), your user prefix will be added to the beginning of the data set name. If you enclose the data set name in quotation marks, it is treated as a fully qualified name.

For more information on the REXX EXECs and EXECs that Language Environment services provide, and on the cataloged procedures that do not contain a compile step, see *z/OS Language Environment Programming Guide*.

For a description of CXXBIND see Chapter 9, "Binding z/OS XL C/C++ programs," on page 401. For a description of CXXMOD see "Prelinking and linking under TSO" on page 626. For a list of the old syntax REXX EXECs, see "Other z/OS XL C utilities" on page 477.

Name	Task Description
REXX EXECs for z/OS 31-bit C and C++	
C370LIB	Maintain an object library under TSO
CXXBIND	Generate an executable module under TSO using the binder
CXXMOD	Generate an executable module under TSO using the pre-link and link
CDADBGLD	Generate a module level debug side file
Cataloged procedures for z/OS XL C and z/OS XL C++	
EDCLIB	Maintain an object library
CCNPD1B	Bind C or C++ object compiled using the IPA(PDF1) and NOXPLINK options
CCNQPD1B	Bind C or C++ object compiled using the IPA(PDF1) and LP64 options
CCNXPD1B	Bind C or C++ object compiled using the IPA(PDF1) and XPLINK options
REXX EXECs for z/OS XL C	
CC	Compile (new syntax - recommended approach) Note: It applies to 31-bit only.
CDSECT	Run DSECT utility
CPLINK	Interactively prelink and link a C program (31-bit only)
GENXLT	Generate a translate table
ICONV	Run the character conversion utility
LOCALEDEF	Produce a locale object
Catalogued procedures for z/OS XL C	
CDAASMC	Compile Common Debug Architecture assembler code to generate both DWARF and ADATA debug information, by default.

Name	Task Description
EDCC	Compile a 31-bit program
EDCCB	Compile and bind a 31-bit program
EDCCBG	Compile, bind, and run a 31-bit program
EDCCL	Compile and link-edit a 31-bit program
EDCCLG	Compile, link-edit, and run a 31-bit program
EDCCLIB	Compile and maintain an object library
EDCI	Run IPA Link step for a 31-bit program
EDCPL	Prelink and link-edit a 31-bit program
EDCCPLG	Compile, prelink, link-edit, and run a 31-bit program
EDCDSECT	Run the DSECT Conversion Utility
EDCGNXLT	Generate a translate table
EDCICONV	Run the character conversion utility
EDCLDEF	Produce a locale object
EDCQB	Bind a 64-bit program
EDCQBG	Bind and run a 64-bit program
EDCQCB	Compile and bind a 64-bit program
EDCQCBG	Compile, bind, and run a 64-bit program
EDCXCB	Compile, and bind an XPLINK 31-bit program
EDCXCBG	Compile, bind, and run an XPLINK 31-bit program
EDCXI	Run IPA Link step for an XPLINK 31-bit or 64-bit program
EDCXLDEF	Create z/OS XL C source from a locale, compile, and bind the XPLINK program to produce an XPLINK locale object
REXX EXECs for z/OS XL C++	
CXX	Compile under TSO
Cataloged procedures for z/OS XL C++	
CBCC	Compile a 31-bit program
CBCCB	Compile and bind a 31-bit program
CBCCBG	Compile, bind and run a 31-bit program
CBCB	Bind a 31-bit program
CBCBG	Bind and run a 31-bit program
CBCCL	Compile, prelink and link a 31-bit program
CBCCLG	Compile, prelink, link and run a 31-bit program
CBCG	Run a 31-bit program
CBCI	Run IPA Link step for a 31-bit program
CBCL	Prelink and link a 31-bit program
CBCLG	Prelink, link and run a 31-bit program
CBCQB	Bind a 64-bit program
CBCQBG	Bind and run a 64-bit program

Name	Task Description
CBCQCB	Compile and bind a 64-bit program
CBCQCBG	Compile, bind, and run a 64-bit program
CBCXB	Bind an XPLINK program
CBCXBG	Bind and run an XPLINK program
CBCXCB	Compile and bind an XPLINK program
CBCXCBG	Compile, bind, and run an XPLINK program
CBCXG	Run a 31-bit or 64-bit program
CBCXI	Run IPA Link step for an XPLINK 31-bit or 64-bit program

Tailoring cataloged procedures, REXX EXECs, and EXECs

A system programmer must modify the cataloged procedures, and REXX EXECs before they are used.

The following data sets contain the cataloged procedures and REXX EXECs that are to be modified:
- CBC.SCCNPRC
- CBC.SCCNUTL
- CEE.SCEEPROC
- CEE.SCEECLST

Most customization for REXX EXECs is in CBC.SCCNUTL(CCNCCUST) and CEE.SCEECLST(CEL4CUST).

The system programmer can make the following changes to REXX EXEC CCNCCUST by editing member CCNCCUST, which resides in data set CBC.SCCNUTL:
- Change or add more binder options by modifying BINDOPTS parameter.
- Change the prefix for Language Environment LIBPRFX from the IBM-supplied default to the high-level qualifier that you chose.
- Change the prefix for XL C/C++ Base Compiler LNGPRFX from the IBM-supplied default to the high-level qualifier that you chose.
- Change the prefix for Run–Time Library Extensions CLBPRFX from the IBM-supplied default to the high-level qualifier that you chose.
- To use Japanese prelinker messages, change PLANG from EDCPMSGE to EDCPMSGK.
- Change the unit parameter TUNIT if the default SYSDA does not suit your system.
- If you need to use the old syntax of CC, change CCSYNTAX to OLD or BOTH.

  | We also have kept the support for the old syntax for compatibility with IBM
  | C/C++ for MVS. It is highly recommended to choose the new syntax, especially
  | for customers who do not have IBM C/C++ for MVS installed or who do not have
  | any dependency on the old syntax. The new syntax allows more flexibility. Refer
  | to the *z/OS XL C/C++ Compiler and Run-Time Migration Guide for the*
  | *Application Programmer* for more information.

- (Optional) You can supply your run-time options for the compiler by modifying variable CBCRTOPT. To use Japanese compiler messages, change NATLANG(ENU) to NATLANG(JPN).
- (Optional) Specify C compiler options in the CBCCCOPT parameter.
- (Optional) Specify C++ compiler options in the CBCCXOPT parameter.

The members in the following table reside in the CBC.SCCNPRC data set.

Table 43. Customization modifications

MEMBER	LNGPRFX	LIBPRFX	CLBPRFX	PLANG
CBCB		✓	✓	
CBCBG		✓	✓	
CBCC	✓	✓	✓	
CBCCB	✓	✓	✓	
CBCCBG	✓	✓	✓	
CBCCL	✓	✓	✓	✓
CBCCLG	✓	✓	✓	✓
CBCG		✓	✓	
CBCI	✓	✓	✓	
CBCL		✓	✓	✓
CBCLG		✓	✓	✓
CBCQB		✓	✓	
CBCQBG		✓	✓	
CBCQCB	✓	✓	✓	
CBCQCBG	✓	✓	✓	
CBCXB		✓	✓	
CBCXBG		✓	✓	
CBCXCB	✓	✓	✓	
CBCXCBG	✓	✓	✓	
CBCXG		✓	✓	
CBCXI	✓	✓	✓	
CCNPD1B	✓	✓	✓	
CCNQPD1B	✓	✓	✓	
CCNXPD1B	✓	✓	✓	
CXXFILT	✓	✓		
EDCC	✓	✓		
EDCCB	✓	✓		
EDCCBG	✓	✓		
EDCCL	✓	✓		
EDCCLG	✓	✓		
EDCCLIB	✓	✓		✓
EDCCPLG	✓	✓		✓
EDCDSECT	✓	✓		
EDCI	✓	✓		

Table 43. Customization modifications (continued)

MEMBER	LNGPRFX	LIBPRFX	CLBPRFX	PLANG
EDCQB		✓		
EDCQBG		✓		
EDCQCB	✓	✓		
EDCQCBG	✓	✓		
EDCXCB	✓	✓		
EDCXCBG	✓	✓		
EDCXI	✓	✓		

The IBM-supplied cataloged procedures provide many parameters to allow each site to customize them easily. The table below describes the commonly used parameters. Use only those parameters that apply to the cataloged procedure you are using. For example, if you are only compiling (EDCC), do not specify any binder parameters.

Parameter	Description
INFILE	For compile procedures, the input z/OS XL C/C++ source file name, PDS name of source files, or directory name of source files. For IPA Link procedures (for example, EDCI, and CBCI), the input IPA object. For prelink, link and bind procedures, the input object. If you do not specify the input data set name, you must use JCL statements to override the appropriate SYSIN DD statement in the cataloged procedure.
OUTFILE	Output module name and file characteristics. For the cataloged procedures ending in a link-edit, bind or go step, specify the name of the file where the load module is to be stored. For most other cataloged procedures, specify the name of the file where the object module is to be stored. If you do not specify an OUTFILE name, a temporary data set will be generated.
CPARM	Compiler options: If two contradictory options are specified, the last is accepted and the first ignored.
STDLIBSD	Enables procedures that contain a bind or prelink/link step to use C128N (NOXPLINK version of the C++ Standard Library).
BPARM	Bind utility options: If two contradictory options are specified, the last is accepted and the first ignored. The set of default binder options passed on the binder invocation does not include COMPAT=CURRENT. The COMPAT option is omitted so the binder default, which is COMPAT=MIN, is used unless it is explicitly overridden by specifying it with the BPARM proc option.
IPARM	IPA Link step options: If two contradictory options are specified, the last is accepted and the first ignored.
PPARM	Prelink utility options: If two contradictory options are specified, the last is accepted and the first ignored.
LPARM	Linkage-editor options: If two contradictory options are specified, the last is accepted and the first ignored.
GPARM	Language Environment run-time (Go step) options and parameters: If two contradictory Language Environment run-time options are specified, the last is accepted and the first ignored.

Parameter	Description
CRUN	Compile step execution run-time parameters for the z/OS XL C/C++ compiler.
IRUN	IPA Link step run-time parameters: for the z/OS XL C/C++ compiler.
OPARM	Object Library Utility parameters. Required for EDCLIB.
OBJECT	Object module to be added to the library. The data-set name (DSN=...) and any applicable keyword parameters (such as, DCB, DISP,) can be specified using this parameter. The default is OBJECT=DUMMY. OBJECT is required for EDCLIB if the ADD function is selected.
LIBRARY	Data-set name for the library for the requested function (ADD, DEL, MAP, or DIR). An example is `LIBRARY='FRED.LIB.OBJ'`. LIBRARY is required for EDCLIB and EDCCLIB.
MEMBER	Member of the library to contain the object module. An example is `MEMBER='MYPROG'`. In z/OS XL C, MEMBER is required for EDCCLIB.

Data sets used

The following table gives a cross-reference of the data sets that each job step requires, and a description of how the data set is used. Refer to the input/output topic of the *z/OS XL C/C++ Programming Guide* for more information about the attributes that are used when opening different types of files.

Table 44. Cross reference of data set used and job step

DD Statement	COMPILE	IPALINK	BIND	PLKED (Prelink)	LKED (Link-Edit)	GO (Run)	EDCALIAS (Object Library)
STEPLIB[1]	X	X	X	X		X	X
SYSCPRT	X	X					
SYSIN	X	X	X	X	X		X
SYSLIB	X	X	X	X	X		X
SYSLIN	X	X	X		X		
SYSLMOD			X		X		
SYSMOD				X			
SYSMSGS				X			X
SYSOUT	X	X		X			X
SYSPRINT		X		X	X	X	X
SYSUTx	X	X			X (SYSUT1)		
IPACNTL		X					

Note: [1] Optional data sets, if the compiler is in DLPA and the run-time library is in LPA, DLPA, or ELPA. To save resources (especially in z/OS UNIX System Services), do not unnecessarily specify data sets on the STEPLIB ddname.

Description of data sets used

The following table lists the data sets that the IBM-supplied cataloged procedures use. It describes the uses of the data set, and the attributes that it supports. You require compiler work data sets only if you specified NOMEM at compile time.

Notes:

1. You should check the defaults at your site for SYSOUT=*
2. The compiler does not directly deal with the SYSOUT DD statement. It uses `stderr`, which in turn is associated with SYSOUT. However, this is just a default ddname, which can be changed by specifying the MSGFILE run-time option. Since the compiler does not directly deal with the DD statement associated with the `stderr`, it cannot provide an alternate DD statement for SYSOUT. Applications that invoke the compiler using one of the documented assembler macros can affect the DD statement that is associated with the `stderr` only by specifying the MSGFILE run-time option in the parameter list, but not via an alternate DD statement.

Table 45. Data set descriptions for cataloged procedures

In Job Step	DD Statement	Description and Supported Attributes (You should check the defaults at your site for `SYSOUT=*`)
COMPILE	SYSIN	For a C++, C, or IPA compilation, the input data set containing the source program. `RECFM=VS, V, VB, VBS, F, FB, FBS, or FS, LRECL≤32760`. It can be a PDS.
COMPILE	SYSLIB	For a C++, C, or IPA compilation, the data set for z/OS XL C/C++ system header files for a source program. SYSLIB must be a PDS or PDSE (`DSORG=PO`) and `RECFM=VS, V, VB, VBS, F, FB  LRECL≤32760`. `RECFM` cannot be mixed. The `LRECLs` for `F` or `FB` `RECFM` must match. For more information on searching system header files, see "SEARCH | NOSEARCH" on page 219.
COMPILE	SYSLIN	Data set for object module. One of the following: • `RECFM=F` or `FS` • `RECFM=FB` or `FBS`. It can be a PDS. `LRECL=80`
COMPILE	SYSOUT	Data set for displaying compiler error messages. `LRECL=137, RECFM=VBA, BLKSIZE=882`. (Defaults for SYSOUT=*).
COMPILE	STEPLIB	Data set for z/OS XL C/C++ compiler and run-time library modules. STEPLIB must be a PDS or PDSE (`DSORG=PO`) with `RECFM=U`, `BLKSIZE=32760`, `LRECL=0`.

Table 45. Data set descriptions for cataloged procedures *(continued)*

In Job Step	DD Statement	Description and Supported Attributes (You should check the defaults at your site for SYSOUT=*)
COMPILE	SYSCPRT	Output data set for compiler listing. LRECL>=137, RECFM=VB,VBA, BLKSIZE=882 (default for SYSOUT=*) LRECL=133, RECFM=FB,FBA, BLKSIZE=133**n*(where *n* is an integer value)
COMPILE	SYSUT1	Obsolete work data set. LRECL=80 and RECFM=F or FB or FBS.
COMPILE	SYSUT5, SYSUT6, SYSUT7, SYSUT8, SYSUT10, SYSUT14, SYSUT16, and SYSUT17	Work data sets. LRECL=3200, RECFM=FB, and BLKSIZE=3200**n* (where *n* is an integer value).
COMPILE	SYSUT9	Work data set. LRECL=137, RECFM=VB, and BLKSIZE=137**n* (where *n* is an integer value) in z/OS XL C, or 882 in z/OS XL C++.
COMPILE	SYSUT10	PPONLY output data set. 72≤LRECL≤32760, RECFM=VS, V, VB, VBS, F, FB, FBS or FS (if not pre-allocated, V is the default). It can be a PDS.
COMPILE	SYSEVENT	Events output file. Must be allocated by the user. For a description of this file, see "EVENTS | NOEVENTS" on page 108 and Appendix E, "Layout of the Events file," on page 669.
COMPILE	TEMPINC (C++ only)	Template instantiation file. Must be a PDS or PDSE. 72≤LRECL≤32760, RECFM=VS, V, VB, VBS, F or FB (default is V).
COMPILE	USERLIB	User header files. Must be a PDS or PDSE. LRECL≤32760, and RECFM=VS, V, VB, VBS, F or FB. For more information on searching user header files, see "SEARCH | NOSEARCH" on page 219.
IPALINK	SYSIN	Data set containing object module for the IPA Link step. LRECL=80 and RECFM=F or FB.

Table 45. Data set descriptions for cataloged procedures (continued)

In Job Step	DD Statement	Description and Supported Attributes (You should check the defaults at your site for SYSOUT=*)
IPALINK	IPACNTL	IPA Link control file directives. RECFM=VS, V, VB, VBS, F, FB, FBS, or FS, LRECL≤32760.
IPALINK	SYSLIB	IPA Link step secondary input. SYSLIB can be a mix of two types of libraries: • Object module libraries. These can be PDSs (DSORG=PO) or PDSEs, with attributes RECFM=F or RECFM=FB, and LRECL=80. • Load module libraries. These must be PDSs (DSORG=PO) with attributes RECFM=U and BLKSIZE≤32760. SYSLIB member libraries must be cataloged.
IPALINK	SYSLIN	Data set for generated object module. One of the following: • RECFM=F or FS • RECFM=FB or FBS
IPALINK	SYSOUT	Data set for displaying compiler error messages. LRECL=137, RECFM=VBA, BLKSIZE=882. (Defaults for SYSOUT=*).
IPALINK	STEPLIB	Data set for z/OS XL C/C++ compiler/run-time library modules. STEPLIB must be a PDS or PDSE (DSORG=PO) with RECFM=U, BLKSIZE≤32760.
IPALINK	SYSCPRT	Output data set for IPA Link step listings. LRECL=137, RECFM=VBA, BLKSIZE=882 (default for SYSOUT=*).
IPALINK	SYSUT1	Obsolete work data set. LRECL=80 and RECFM=F or FB or FBS.
IPALINK	SYSUT5, SYSUT6, SYSUT7, SYSUT8, SYSUT10, SYSUT14, SYSUT16, and SYSUT17	Work data sets. LRECL=3200, RECFM=FB, and BLKSIZE=3200**n* (where *n* is an integer value).
IPALINK	SYSUT9	Work data set. LRECL=137, RECFM=VB, and BLKSIZE=137**n* (where *n* is an integer value).

Table 45. Data set descriptions for cataloged procedures (continued)

In Job Step	DD Statement	Description and Supported Attributes (You should check the defaults at your site for SYSOUT=*)
BIND	SYSDEFSD	Output from binding a DLL (an application that exports symbols). LRECL=80 and RECFM=F or FB or FBS
BIND	SYSIN	Data set for additional object for the binder. It defaults to Dummy. LRECL=80 and RECFM=F, FB or FBS.
BIND	SYSLIB	Data set for binder automatic call libraries.
BIND	SYSLIN	Primary input data set for the binder One of the following: RECFM=F or FS RECFM=FB or FBS.
BIND	SYSLMOD	Output Program Object Library. PDSE with RECFM=U and BLKSIZE<=32760.
BIND	SYSPRINT	Data set for listing of binder diagnostic messages. LRECL=137, RECFM=VBA, BLKSIZE=882. (Default attributes for SYSOUT=*).
PLKED	STEPLIB	Data set containing prelink utility modules. STEPLIB must be a PDS or PDSE (DSORG=PO) and RECFM=U and BLKSIZE≤32760.
PLKED	SYSDEFSD	Output from prelinking a DLL (an application that exports symbols). LRECL=80 and RECFM=F or FB or FBS
PLKED	SYSIN	Data set containing object module for the prelink utility. This is the primary input data set. LRECL=80 and RECFM=F, FB or FBS.
PLKED	SYSLIB	Data set for automatic call libraries to be used with the prelinker. SYSLIB must be cataloged and LRECL=80 and RECFM=F or FB or FBS. DSORG=PO
PLKED	SYSMOD	Data set for output of the prelink utility LRECL=80 and RECFM=F or FB or FBS.
PLKED	SYSMSGS	Data set containing prelink utility messages. LRECL=150, RECFM=F or FB or FBS and BLKSIZE=6150.

Table 45. Data set descriptions for cataloged procedures (continued)

In Job Step	DD Statement	Description and Supported Attributes (You should check the defaults at your site for SYSOUT=*)
PLKED	SYSOUT	Data set for the prelinker map. LRECL=80 and RECFM=F or FB or FBS
PLKED	SYSPRINT	Data set for listing of prelink utility diagnostic messages. LRECL=137, RECFM=VBA, BLKSIZE=882. (Default attributes for SYSOUT=*).
LKED	SYSLIB	Data set for z/OS XL C/C++ autocall library. SYSLIB must be a PDS or PDSE and have the attributes RECFM=U and BLKSIZE≤32760.
LKED	SYSLIN	Primary input data set for linkage editor One of the following: • RECFM=F or FS • RECFM=FB or FBS
LKED	SYSLMOD	Output load module library. RECFM=U and BLKSIZE≤32760.
LKED	SYSPRINT	Data set for listings and diagnostics produced by the linkage editor. One of the following: • LRECL=121, and RECFM=FA • LRECL=121, RECFM=FBA, and BLKSIZE=121**n* (where *n* is less than or equal to 40).
LKED	SYSIN	Data set for additional object for the binder. It defaults to Dummy. LRECL=80 and RECFM=F, FB or FBS.
LKED	SYSUT1	Work data set. The data set attributes will be supplied by the linkage editor.
GO	STEPLIB	Run-time libraries. STEPLIB must be one or more PDSes or PDSEs and have the attributes RECFM=U and BLKSIZE≤32760.
GO	CEEDUMP	Data set for error messages generated by Language Environment Dump Services. CEEDUMP must be a sequential data set and it must be allocated to SYSOUT, a terminal, or a unit record device, or a data set with the attributes RECFM=VBA, LRECL=125, and BLKSIZE=882.

Table 45. Data set descriptions for cataloged procedures (continued)

In Job Step	DD Statement	Description and Supported Attributes (You should check the defaults at your site for SYSOUT=*)
GO	SYSPRINT	Data set for listings and diagnostics from user program. LRECL=137, RECFM=VBA, BLKSIZE=882. (default attributes for SYSOUT=*).
OUTILITY	SYSIN	Input data set for object module to be added to the library. It can be sequential or partitioned (with a member name specified). LREL=80, RECFM=F or FB or FBS.
OUTILITY	SYSLIB	Library for which the member name is to be added (ADD); for which the member name is to deleted (DEL); which is to be listed (MAP); for which the C370LIB-directory is to be built. This DD must point to a single partitioned data set. Concatenations cannot be used. Member names must not be specified. LREL=80, RECFM=F or FB or FBS.
OUTILITY	SYSOUT	Output data set for the C370LIB-directory map. It can be sequential or partitioned (with a member name specified). LREL=80, RECFM=F or FB or FBS.
OUTILITY	SYSMSGS	Data set containing the input messages. LRECL=150, RECFM=F or FB or FBS.
OUTILITY	SYSPRINT	Data set diagnostics from the C370LIB program. The default is to SYSOUT=*. LRECL=137, RECFM=VBA, BLKSIZE=882

Examples using cataloged procedures

```
//*------------------------------------------------------------------
//* Compile a Partitioned Data Set program with various options
//*------------------------------------------------------------------
//EXAMPLE1 EXEC EDCC,
//          INFILE='PATRICK.TEST.PDSSRC(CPROG1)',
//          OUTFILE='PATRICK.TEST.OBJECT(CPROG1),DISP=SHR',
//          CPARM='OPT NOSEQ NOMAR LIST'
//COMPILE.USERLIB DD DSNAME=PATRICK.HDR.FILES,DISP=SHR
//*
//*------------------------------------------------------
//* Compile a Sequential program with various options
//*------------------------------------------------------
//EXAMPLE2 EXEC EDCC,
//          INFILE='PATRICK.TEST.SEQSRC.CPROG2',
//          OUTFILE='PATRICK.TEST.OBJECT(CPROG2),DISP=SHR',
//          CPARM='OPT SOURCE XREF FLAG(E)'
//COMPILE.USERLIB DD DSNAME=PATRICK.HDR.FILES,DISP=SHR
```

Figure 42. Example compilation for z/OS XL C using EDCC

```
//*
//CCMEM   EXEC CBCC,          * Compile C++ source member
//        INFILE='MIKE.CPP(ONLYONE)',
//        OUTFILE='MIKE.SAMPLE.OBJ(ONLYONE),DISP=SHR ',
//        CPARM='OPT SOURCE SHOWINC LIST'
//*
//CCPDS   EXEC CBCC,          * Compile C++ source PDS
//        INFILE='MIKE.CPP',
//        OUTFILE='MIKE.PROJECT.OBJ,DISP=SHR ',
//        CPARM='NOOPT'
```

Figure 43. Example Compilation for z/OS XL C++ Using CBCC

Other z/OS XL C utilities

| Starting with IBM C/C++ for MVS V3R2, several improvements were made to the REXX EXECs provided with the C/C++ compiler. The improved REXX EXECs use a different syntax, which we refer to as the *new syntax*. The *old syntax* is the syntax of the REXX EXECs prior to the C/C++ for MVS V3R2 release of the compiler. This topic describes the old syntax for these REXX EXECs, which is still supported. In the following table, we indicate the corresponding updated REXX EXECs which will provide new features and greater flexibility.

Name	Task Description	Substitute
CC (old syntax)	Compile	CC (new syntax)
CMOD	Generate an executable module	CXXMOD

Figure 44. Utilities for z/OS XL C

For a description of CXXMOD, see “Prelinking and linking under TSO” on page 626.

Using the old syntax for CC

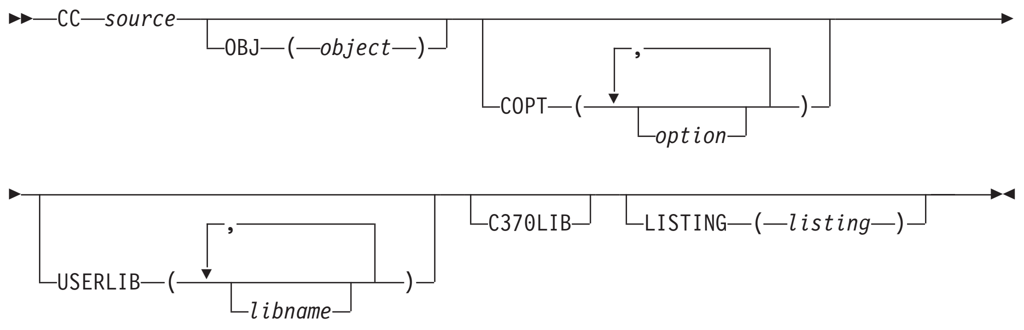
The CC command can now be invoked using a new syntax. At installation time, your system programmer can customize the CC EXEC to accept:
- Only the old syntax (the one supported by compilers prior to IBM C/MVS™ Version 3 Release 2)
- Only the new syntax

- Both syntaxes

The CC EXEC should be customized to accept only the new syntax. If you customize the CC EXEC to accept only the old syntax, keep in mind that it does not support z/OS UNIX System Services files. If you customize the CC EXEC to accept both the old and new syntaxes, you must invoke it using either the old syntax *or* the new syntax, but not a mixture of both. If you invoke this EXEC with the old syntax, it will not support z/OS UNIX files.

For information on the new syntax, see "Using the CC and CXX REXX EXECs" on page 346. Refer to the *z/OS Program Directory* for more information about installation and customization.

The old syntax for the CC REXX EXEC is:



You can override the default compiler options by specifying the options:
- In the `COPT` keyword parameter
- In a **#pragma options** directive in your source file
- By specifying them directly on the invocation line

However, any options specified on **#pragma options** directives are overridden by options specified on the invocation line.

The following rules apply when you use the old syntax for the CC REXX EXEC:
- When you are specifying a data set name, if the name is not enclosed in single quotation marks ('), your user prefix will be added to the beginning of the data set name. If the data set name is enclosed in single quotation marks, it will be treated as a fully qualified name.
- When you need to use spaces, commas, single quotation marks, or parentheses within a REXX EXEC option, the text must be placed inside a string using single quotation marks.
- If you want to use a single quotation mark inside a string, you must use two quotation marks in place of each quotation mark.

Example: The following example demonstrates these rules:

```
CC TEST.C(STOCK) COPT ('SEARCH(CLOTHES.H ''MARK.SUPPLY.C(ORDER)'')')
```

Using CMOD

The CMOD REXX EXEC makes a call to LINK with the appropriate library. The syntax of the CMOD REXX EXEC is:

```
                  ┌─────────────┐
►►──CMOD──OBJ──(──▼─object_deck─┴──)──┬────────────────────────┬──────────────────────────►
                                      │        ┌─────────┐     │
                                      └─LIB──(─▼─libname─┴──)──┘

►──┬─────────────────────┬──┬─────────────────────────────┬──────────────────────────────►◄
   └─LOAD──(──libname──)─┘  │           ┌─────────────┐   │
                            └─LOPT──(───▼─link_option─┴──)┘
```

OBJ Specifies the object decks that you want to link.

LIB Specifies the libraries that are to be used to resolve external entries.

LOAD Specifies the output library in which the load module is to be stored.

LOPT Specifies the options that you want to pass to the linkage editor. All options are passed to the TSO LINK command.

A non-zero return code indicates that an error has occurred. For diagnostic information, refer to Appendix C, "Diagnosing problems," on page 647. CMOD can also return the return code from LINK. See the appropriate document in your TSO library for more information on LINK.

Chapter 13. Object library utility

This information describes how to use the object library utility to process libraries of object modules. On z/OS, a library is a PDS or PDSE with object modules as members.

Object libraries (also called object library utility directories) provide convenient packaging of object modules in MVS data sets, in much the same way as the z/OS UNIX System Services ar utility packages object modules that reside in z/OS UNIX files. Using the object library utility, you can create libraries that contain object modules compiled with various combinations of compiler options, such as LONGNAME | NOLONGNAME, XPLINK | NOXPLINK, IPA | NOIPA and LP64 | ILP32.

The object library utility keeps track of the attributes of each of its members in two special members of the library, which are the Basic Directory Member (`@@DC370$`) and the Enhanced Directory Member (`@@DC390$`). The *Basic Directory Member* is used to maintain backwards compatibility with object library utility directories that were created with older versions of the object library utility (pre-z/OS V1R2). The *Enhanced Directory Member* was introduced to support object modules that were compiled with the IPA, XPLINK, or LP64 compiler options, as well as provide more detailed listing information. If you do have older object library utility directories at your site, you should consider upgrading them to include the Enhanced Directory Member by using the DIR command (described later in this information).

Commands for this utility allow you to add and delete object modules from a library, rebuild the Basic and Enhanced Directory Members, and to create a listing of all the contents in an object library utility directory.

You can create an object library under z/OS batch and TSO, but not from under z/OS UNIX.

Creating an object library under z/OS batch

Under z/OS batch, the following cataloged procedures include an object library utility step:

EDCLIB	Maintain an object library
EDCCLIB	Compile and maintain an object library (C only)

The EDCLIB cataloged procedure is located in the CEE.SCEEPROC data set. The EDCCLIB cataloged procedure is located in the CBC.SCCNPRC data set. For more information on the data sets that you use with the object library utility, see "Description of data sets used" on page 470.

To compile the z/OS XL C source file `WALTER.SOURCE(SUB1)` with the LONGNAME compiler option, and then add it to the preallocated PDS (or PDSE) data set `WALTER.SOURCE.LIB`, use the following JCL. If this is the first time the object library utility has been used to add an object module to `WALTER.SOURCE.LIB`, then the Basic and Enhanced Directory members will be created in this data set. If they already exist in this data set, then they will be updated to include the information for the object module created during the compilation.

```
//COMPILE EXEC EDCCLIB,INFILE='WALTER.SOURCE(SUB1)',CPARM='LO',
//      LIBRARY='WALTER.SOURCE.LIB',MEMBER='SUB1'
```

If you request a map for the library `WALTER.SOURCE.LIB`, use the following:

```
//OBJLIB EXEC EDCLIB,OPARM='MAP',LIBRARY='WALTER.SOURCE.LIB'
```

For z/OS XL C++, use the EDCLIB cataloged procedure. You can specify commands for the object library utility step on the OPARM parameter. You can specify options for the object library utility step. These options can generate a library directory, add members or delete members of a directory, or generate a map of library members and defined external symbols. This topic shows you how to specify these options under z/OS batch.

Example: The following example creates a new object library utility directory. If the directory already exists, it is updated.

```
//DIRDIR   EXEC EDCLIB,
//         LIBRARY='LUCKY13.CXX.OBJMATH',
//         OPARM='DIR'
```

Example: To create a listing of all the object files (members) in an object library utility directory:

```
//MAPDIR   EXEC EDCLIB,
//         LIBRARY='LUCKY13.CXX.OBJMATH',
//         OPARM='MAP'
```

To add new members to an object library, use the ADD option to update the directory.

Example: To add a new member named MA191:

```
//ADDDIR   EXEC EDCLIB,
//         LIBRARY='LUCKY13.CXX.OBJMATH',
//         OPARM='ADD MA191',
//         OBJECT='DSNAME=LUCKY13.CXX.OBJ(OBJ191),DISP=SHR'
```

To delete a member from an object library, use the DEL option to keep the directory up to date.

Example: To delete a member named OLDMEM:

```
//DELDIR   EXEC EDCLIB,
//         LIBRARY='LUCKY13.CXX.OBJMATH',
//         OPARM='DEL OLDMEM'
```

Creating an object library under TSO

The object library utility has the following syntax:

```
►►──C370LIB──┬─ADD──LIB──(libname(membername))──┬───────────────────┬─┬──────────►◄
             │                                  └─OBJ──(objname)────┘ │
             ├─DEL──LIB──(libname(membername))────────────────────────┤
             ├─MAP──LIB──(libname)──┬────────────────┬────────────────┤
             │                      └─LIST──(map)────┘                │
             ├─MAP370──LIB──(libname)──┬────────────────┬─────────────┤
             │                         └─LIST──(map)────┘             │
             ├─MAP390──LIB──(libname)──┬────────────────┬─────────────┤
             │                         └─LIST──(map)────┘             │
             ├─DIR──LIB──(libname)────────────────────────────────────┤
             └─DIR390──LIB──(libname)─────────────────────────────────┘
```

where:

ADD
: Adds (or replaces) an object module to an object library.

	If you use ADD to insert an object module to a member of a library that already exists, the previous member is deleted prior to the insert. If the source data set is the same as the target data set, ADD does not delete the member, and only updates the Object Library Utility directory.
DEL	Deletes an object module from an object library.
MAP	Lists the names (entry points) of object library members in the Enhanced Directory Member if it is available; otherwise in the Basic Directory Member. You will only see object library members that were compiled with the options IPA(NOOBJECT), XPLINK or LP64 in the listing if the Enhanced Directory Member is available.
MAP370	Lists the names (entry points) of all object library members in the Basic Directory Member.
MAP390	Lists the names (entry points) of all object library members in the Enhanced Directory Member.
DIR	Builds the Object Library Utility directory member. The object library utility directory contains the names (entry points) of library members. The DIR function is only necessary if object modules were previously added or deleted from the library without using the object library utility.
DIR390	As of z/OS V1R2, the DIR and DIR390 commands are aliases of each other, and can be used interchangeably.
LIB(*libname(membername)***)**	Specifies the target data set for the ADD and DEL functions. The data set name must contain a member specification to indicate which member Object Library Utility should create, replace, or delete.
OBJ(*objname***)**	Specifies the source data set that contains the object module that is to be added to the library. If you do not specify a data set name, the object library utility uses the target data set that you specified in LIB(*libname*(*membername*)) as the source.
LIB(*libname***)**	Specifies the object library for which a map is to be produced or for which an object library utility directory is to be built.
LIST(*map***)**	Specifies the data set that is to contain the object library utility listing. If you specified an asterisk (*), the listing is directed to your terminal. If you do not specify a data set name, a name is generated using the library name and the qualifier MAP. If `TEST.OBJ` is the input library data set, and your user prefix is `FRANK`, the data set name for the listing is `FRANK.TEST.OBJ.MAP`.

Under TSO, for z/OS XL C you can use either the C370LIB REXX EXEC or the CC REXX EXEC with the parameter C370LIB. The C370LIB parameter of the CC REXX EXEC specifies that, if the object module from the compile is directed to a PDS member, the object library utility directory is to be updated. This step is the equivalent to a compile and C370LIB ADD step. If the C370LIB parameter is specified, and the object module is not directed to a member of a PDS, the C370LIB parameter is ignored.

Object library utility map

The object library utility produces a listing for a given library when you specify the MAP, MAP370, or MAP390 command. MAP370 displays the listing using only the information in the Basic Directory Member. It assumes that all the extended attributes are set as zero, which provides backward compatibility with earlier versions of the object library utility. MAP390 displays the listing using only the information in the Enhanced Directory Member. MAP is the preferred way of getting a listing. It generates a listing based on the Enhanced Directory Member if it's available, otherwise it generates a listing based on the Basic Directory Member. It provides additional attribute information on symbols when the information is available.

Example: The example that follows is produced by the object library utility for a given library when you specify the MAP or MAP390 command. The listing contains information on each member of the library.

```
========================================================================
|1                       Object Library Utility Map                    |
|                                                                      |
|C370LIB:5694A01 V1 R11 M0 IBM LANGUAGE ENVIRONMENT 2009/04/29 04:55:17|
========================================================================

  Library Name: USERID1.LIB

*----------------------------------------------------------------------*
* 2  Member Name:    CGOFF                  (P) 2009/04/29 04:55:16 *
*                                               5694A01    V1 R11   *
*----------------------------------------------------------------------*
3
   User Comment:
AGGRCOPY(NOOVERLAP) NOALIAS ANSIALIAS ARCH(5) ARGPARSE NOASCII
ASSERT(RESTRICT) BITFIELD(UNSIGNED) CHARS(UNSIGNED) NOCOMPACT
NOCOMPRESS NOCONVLIT CSECT() NODEBUG NODFP NODLL(NOCALLBACKANY)
ENUMSIZE(SMALL) EXECOPS NOEXPORTALL FLOAT(HEX, FOLD, NOMAF, NORRM,
AFP(NOVOLATILE)) GOFF NOGONUMBER NOHGPR(NOPRESERVE) NOHOT NOIGNERRNO
ILP32 NOINITAUTO NOINLINE NOIPA LANGLVL(EXTENDED) NOLIBANSI NOLOCALE
LONGNAME MAXMEM(2097152) NOOPTIMIZE PLIST(HOST) PREFETCH REDIR NORENT
NOROCONST ROUND(Z) ROSTRING NORTCHECK NOSERVICE SPILL(128) START STRICT
NOSTRICT_INDUCTION TARGET(LE, zOSV1R11) TUNE(5) UNROLL(AUTO) NOUPCONV
NOWSIZEOF NOXPLINK COMPILED_ON_MVS

4 (    L) Function Name: @InStream@#C
   (    L) Function Name: foo
   (   WL) External Name: @InStream@#S
   (   WL) External Name: @InStream@#T
   (   WL) External Name: this_int_is_in_writable_static_and_will_wrap_b
                          ecause_it_is_too_long

*----------------------------------------------------------------------*
*  Member Name: CPPIPANO                    (P) 2009/04/29 04:55:18 *
*                                               5694A01    V1 R02   *
*----------------------------------------------------------------------*

   User Comment:
AGGRCOPY(NOOVERLAP) ANSIALIAS ARCH(5) ARGPARSE NOASCII ASSERT(RESTRICT)
BITFIELD(UNSIGNED) CHARS(UNSIGNED) NOCOMPACT NOCOMPRESS NOCONVLIT
NOCSECT CVFT NODEBUG NODFP DLL(NOCALLBACKANY) ENUMSIZE(SMALL) EXECOPS
EXH NOEXPORTALL FLOAT(HEX, FOLD, NOMAF, NORRM, AFP(NOVOLATILE)) NOGOFF
NOGONUMBER NOHGPR(NOPRESERVE) NOHOT NOIGNERRNO NOINITAUTO
NOINLINE(NOAUTO, NOREPORT, 100, 1000) IPA(NOLINK, NOOBJECT, OPTIMIZE,
COMPRESS, NOGONUM, NOPDF1, NOPDF2, NOATTRIBUTE, NOXREF)
LANGLVL(ANONSTRUCT, ANONUNION, ANSIFOR, ANSISINIT, C99VLA, C99__FUNC__,
NODBCS, DEPENDENTBASELOOKUP, NODOLLARINNAMES, EMPTYSTRUCT, ILLPTOM,
IMPLICITINT, LIBEXT, LONGLONG, NONEWEXCP, OFFSETNONPOD, NOOLDDIGRAPH,
OLDFRIEND, NOOLDMATH, NOOLDSTR, OLDTEMPACC, NOOLDTMPLALIGN, OLDTMPLSPEC
 TRAILENUM, TYPEDEFCLASS, NOUCS, GNU_INCLUDE_NEXT, ZEROEXTARRAY)
NOLIBANSI NOLOCALE LONGNAME ILP32 MAXMEM(2097152) NAMEMANGLING(zOSV1R2)
OBJECTMODEL(COMPAT) NOOPTIMIZE PLIST(HOST) PREFETCH REDIR ROCONST
ROSTRING ROUND(Z)  NORTCHECK NORTTI NOSERVICE SPILL(128) START STRICT
NOSTRICT_INDUCTION TARGET(LE, zOSV1R11) TEMPLATERECOMPILE
NOTEMPLATEREGISTRY TMPLPARSE(NO) TUNE(5) UNROLL(AUTO) NOWSIZEOF
NOXPLINK(NOBACKCHAIN, NOCALLBACK, NOGUARD, OSCALL(UPSTACK),
NOSTOREARGS) COMPILED_ON_MVS

   ( I  L) Function Name: myclass::myclass()
   ( I  L) External Name: another_global
   ( I  L) Function Name: myclass::foo(float,double)
   ( I  L) Function Name: some_function(char)
   ( I  L) External Name: some_global
```

```
*-------------------------------------------------------------------*
*  Member Name: CPPNOIPA                    (P) 2009/04/29 04:55:20 *
*                                               5694A01    V1 R11   *
*-------------------------------------------------------------------*

   User Comment:
AGGRCOPY(NOOVERLAP) ANSIALIAS ARCH(5) ARGPARSE NOASCII ASSERT(RESTRICT)
BITFIELD(UNSIGNED) CHARS(UNSIGNED) NOCOMPACT NOCOMPRESS NOCONVLIT
NOCSECT CVFT NODEBUG NODFP DLL(NOCALLBACKANY) ENUMSIZE(SMALL) EXECOPS
EXH NOEXPORTALL FLOAT(HEX, FOLD, NOMAF, NORRM, AFP(NOVOLATILE)) NOGOFF
NOGONUMBER NOHGPR(NOPRESERVE) NOHOT NOIGNERRNO NOINITAUTO
NOINLINE(NOAUTO, NOREPORT, 100, 1000) NOIPA LANGLVL(ANONSTRUCT,
ANONUNION, ANSIFOR, ANSISINIT, C99VLA, C99__FUNC__, NODBCS,
DEPENDENTBASELOOKUP, NODOLLARINNAMES, EMPTYSTRUCT, ILLPTOM, IMPLICITINT
 LIBEXT, LONGLONG, NONEWEXCP, OFFSETNONPOD, NOOLDDIGRAPH, OLDFRIEND,
NOOLDMATH, NOOLDSTR, OLDTEMPACC, NOOLDTMPLALIGN, OLDTMPLSPEC, TRAILENUM
 TYPEDEFCLASS, NOUCS, GNU_INCLUDE_NEXT, ZEROEXTARRAY) NOLIBANSI
NOLOCALE LONGNAME ILP32 MAXMEM(2097152) NAMEMANGLING(zOSV1R2)
OBJECTMODEL(COMPAT) NOOPTIMIZE PLIST(HOST) PREFETCH REDIR ROCONST
ROSTRING ROUND(Z)  NORTCHECK NORTTI NOSERVICE SPILL(128) START STRICT
NOSTRICT_INDUCTION TARGET(LE, zOSV1R11) TEMPLATERECOMPILE
NOTEMPLATEREGISTRY TMPLPARSE(NO) TUNE(5) UNROLL(AUTO) NOWSIZEOF
NOXPLINK(NOBACKCHAIN, NOCALLBACK, NOGUARD, OSCALL(UPSTACK),
NOSTOREARGS) COMPILED_ON_MVS

   (    L) Function Name: myclass::myclass()
   (    L) Function Name: myclass::foo(float,double)
   (    L) Function Name: some_function(char)
   (   WL) External Name: another_global
   (   WL) External Name: some_global

*-------------------------------------------------------------------*
*  Member Name:  CPPLP64                    (P) 2009/04/29 04:55:20 *
*                                               5694A01    V1 R11   *
*-------------------------------------------------------------------*

   User Comment:
AGGRCOPY(NOOVERLAP) ANSIALIAS ARCH(5) ARGPARSE NOASCII ASSERT(RESTRICT)
BITFIELD(UNSIGNED) CHARS(UNSIGNED) NOCOMPACT NOCOMPRESS NOCONVLIT
CSECT(CODE, @InStream@#C) CSECT(STATIC, @InStream@#S) CSECT(TEST,
@InStream@#T) CVFT NODEBUG NODFP DLL(NOCALLBACKANY) ENUMSIZE(SMALL)
EXECOPS EXH NOEXPORTALL FLOAT(HEX, FOLD, NOMAF, NORRM, AFP(NOVOLATILE))
GOFF NOGONUMBER NOHGPR(NOPRESERVE) NOHOT NOIGNERRNO NOINITAUTO
NOINLINE(NOAUTO, NOREPORT, 100, 1000) NOIPA LANGLVL(ANONSTRUCT,
ANONUNION, ANSIFOR, ANSISINIT, C99VLA, C99__FUNC__, NODBCS,
DEPENDENTBASELOOKUP, NODOLLARINNAMES, EMPTYSTRUCT, ILLPTOM, IMPLICITINT
 LIBEXT, LONGLONG, NONEWEXCP, OFFSETNONPOD, NOOLDDIGRAPH, OLDFRIEND,
NOOLDMATH, NOOLDSTR, OLDTEMPACC, NOOLDTMPLALIGN, OLDTMPLSPEC, TRAILENUM
 TYPEDEFCLASS, NOUCS, GNU_INCLUDE_NEXT, ZEROEXTARRAY) NOLIBANSI
NOLOCALE LONGNAME LP64 MAXMEM(2097152) NAMEMANGLING(ANSI)
OBJECTMODEL(IBM) NOOPTIMIZE PLIST(HOST) PREFETCH REDIR ROCONST ROSTRING
ROUND(Z)  NORTCHECK NORTTI NOSERVICE SPILL(256) START STRICT
NOSTRICT_INDUCTION TARGET(LE, zOSV1R11) TEMPLATERECOMPILE
NOTEMPLATEREGISTRY TMPLPARSE(NO) TUNE(5) UNROLL(AUTO) NOWSIZEOF
XPLINK(NOBACKCHAIN, NOCALLBACK, GUARD, OSCALL(UPSTACK), NOSTOREARGS)
COMPILED_ON_MVS

   (6 X L) Function Name: myclass::myclass()
   (6 X L) Function Name: @InStream@#C
   (6 X L) Function Name: myclass::foo(float,double)
   (6 X L) Function Name: some_function(char)
   (6 XWL) External Name: @InStream@#S
   (6 XWL) External Name: @InStream@#T
   (6 XWL) External Name: another_global
   (6 XWL) External Name: some_global
```

```
*------------------------------------------------------------------------*
*  Member Name:   CIPA64                       (P) 2009/04/29 04:55:22 *
*                                                   5694A01    V1 R02   *
*------------------------------------------------------------------------*

   User Comment:
AGGRCOPY(NOOVERLAP) NOALIAS ANSIALIAS ARCH(5) ARGPARSE NOASCII
ASSERT(RESTRICT) BITFIELD(UNSIGNED) CHARS(UNSIGNED) NOCOMPACT
NOCOMPRESS NOCONVLIT CSECT() NODEBUG NODFP NODLL(NOCALLBACKANY)
ENUMSIZE(SMALL) EXECOPS NOEXPORTALL FLOAT(HEX, FOLD, NOMAF, NORRM,
AFP(NOVOLATILE)) GOFF NOGONUMBER NOHGPR(NOPRESERVE) NOHOT NOIGNERRNO
NOINITAUTO NOINLINE IPA(NOLINK, NOOBJ, COM, OPT, NOGONUM)
LANGLVL(EXTENDED) NOLIBANSI NOLOCALE LONGNAME LP64 MAXMEM(2097152)
NOOPTIMIZE PLIST(HOST) PREFETCH REDIR RENT NOROCONST ROUND(Z) ROSTRING
NORTCHECK NOSERVICE SPILL(256) START STRICT NOSTRICT_INDUCTION
TARGET(LE, zOSV1R11) TUNE(5) UNROLL(AUTO) NOUPCONV NOWSIZEOF
XPLINK(NOBACKCHAIN, NOSTOREARGS, NOCALLBACK, GUARD, OSCALL(NOSTACK))
COMPILED_ON_MVS

   (6IX L) Function Name: foo
   (6IX L) External Name: this_int_is_in_writable_static_and_will_wrap_b
                          ecause_it_is_too_long

========================================================================
|5                        Symbol Definition Map                         |
========================================================================


*------------------------------------------------------------------------*
|6   Symbol Name: @InStream@#C                                          |
*------------------------------------------------------------------------*

  From member:    CGOFF Type: Function (    L)
  From member:  CPPLP64 Type: Function (6 X L)

*------------------------------------------------------------------------*
|  Symbol Name: this_int_is_in_writable_static_and_will_wrap_because_i |
|               t_is_too_long                                           |
*------------------------------------------------------------------------*

  From member:    CGOFF Type: External (   WL)
  From member:   CIPA64 Type: External (6IX L)

*------------------------------------------------------------------------*
|  Symbol Name: foo                                                     |
*------------------------------------------------------------------------*

  From member:    CGOFF Type: Function (    L)
  From member:   CIPA64 Type: Function (6IX L)

*------------------------------------------------------------------------*
|  Symbol Name: @InStream@#T                                            |
*------------------------------------------------------------------------*

  From member:    CGOFF Type: External (   WL)
  From member:  CPPLP64 Type: External (6 XWL)

*------------------------------------------------------------------------*
|  Symbol Name: @InStream@#S                                            |
*------------------------------------------------------------------------*

  From member:    CGOFF Type: External (   WL)
  From member:  CPPLP64 Type: External (6 XWL)
```

```
*----------------------------------------------------------------------*
|  Symbol Name: some_function(char)                                    |
*----------------------------------------------------------------------*

  From member: CPPNOIPA Type: Function (    L)
  From member:  CPPLP64 Type: Function (6 X L)
  From member: CPPIPANO Type: Function ( I  L)

*----------------------------------------------------------------------*
|  Symbol Name: myclass::myclass()                                     |
*----------------------------------------------------------------------*

  From member: CPPNOIPA Type: Function (    L)
  From member: CPPIPANO Type: Function ( I  L)

*----------------------------------------------------------------------*
|  Symbol Name: myclass::foo(float,double)                             |
*----------------------------------------------------------------------*

  From member: CPPNOIPA Type: Function (    L)
  From member: CPPIPANO Type: Function ( I  L)

*----------------------------------------------------------------------*
|  Symbol Name: some_global                                            |
*----------------------------------------------------------------------*

  From member: CPPNOIPA Type: External (   WL)
  From member:  CPPLP64 Type: External (6 XWL)
  From member: CPPIPANO Type: External ( I  L)

*----------------------------------------------------------------------*
|  Symbol Name: another_global                                         |
*----------------------------------------------------------------------*

  From member: CPPNOIPA Type: External (   WL)
  From member:  CPPLP64 Type: External (6 XWL)
  From member: CPPIPANO Type: External ( I  L)

*----------------------------------------------------------------------*
|  Symbol Name: myclass::myclass()                                     |
*----------------------------------------------------------------------*

  From member:  CPPLP64 Type: Function (6 X L)

*----------------------------------------------------------------------*
|  Symbol Name: myclass::foo(float,double)                             |
*----------------------------------------------------------------------*

  From member:  CPPLP64 Type: Function (6 X L)

=========  E N D   O F   O B J E C T   L I B R A R Y   M A P  ==========
```

|

The object library utility produces a listing for a given library when the MAP370 command is specified. The listing produced by MAP370 will only contain information from the object library utility directory members that are in the XOBJ object file format. In other words, files compiled with the GOFF compiler option (which includes all XPLINK and LP64 compiled object files) will not appear in the MAP370 listing. Also, IPA(NOOBJECT) compiled files will not appear in the MAP370 listing either.

```
=========================================================================
|1                        Object Library Utility Map                     |
|                                                                        |
|C370LIB:5694A01 V1 R11 M0 IBM LANGUAGE ENVIRONMENT 2009/04/29 04:55:17|
=========================================================================

  Library Name: USERID1.LIB

*-----------------------------------------------------------------------*
* 2   Member Name:    CGOFF                   (P) 2009/04/29 04:55:16 *
*-----------------------------------------------------------------------*

*-----------------------------------------------------------------------*
*  Member Name: CPPIPANO                      (P) 2009/04/29 04:55:18 *
*                                                 5694A01    V1 R02   *
*-----------------------------------------------------------------------*
 3
   User Comment:
AGGRCOPY(NOOVERLAP) ANSIALIAS ARCH(5) ARGPARSE NOASCII ASSERT(RESTRICT)
BITFIELD(UNSIGNED) CHARS(UNSIGNED) NOCOMPACT NOCOMPRESS NOCONVLIT
NOCSECT CVFT NODEBUG NODFP DLL(NOCALLBACKANY) ENUMSIZE(SMALL) EXECOPS
EXH NOEXPORTALL FLOAT(HEX, FOLD, NOMAF, NORRM, AFP(NOVOLATILE)) NOGOFF
NOGONUMBER NOHGPR(NOPRESERVE) NOHOT NOIGNERRNO NOINITAUTO
NOINLINE(NOAUTO, NOREPORT, 100, 1000) IPA(NOLINK, NOOBJECT, OPTIMIZE,
COMPRESS, NOGONUM, NOPDF1, NOPDF2, NOATTRIBUTE, NOXREF)
LANGLVL(ANONSTRUCT, ANONUNION, ANSIFOR, ANSISINIT, C99VLA, C99__FUNC__,
NODBCS, DEPENDENTBASELOOKUP, NODOLLARINNAMES, EMPTYSTRUCT, ILLPTOM,
IMPLICITINT, LIBEXT, LONGLONG, NONEWEXCP, OFFSETNONPOD, NOOLDDIGRAPH,
OLDFRIEND, NOOLDMATH, NOOLDSTR, OLDTEMPACC, NOOLDTMPLALIGN, OLDTMPLSPEC
 TRAILENUM, TYPEDEFCLASS, NOUCS, GNU_INCLUDE_NEXT, ZEROEXTARRAY)
NOLIBANSI NOLOCALE LONGNAME ILP32 MAXMEM(2097152) NAMEMANGLING(zOSV1R2)
OBJECTMODEL(COMPAT) NOOPTIMIZE PLIST(HOST) PREFETCH REDIR ROCONST
ROSTRING ROUND(Z)  NORTCHECK NORTTI NOSERVICE SPILL(128) START STRICT
NOSTRICT_INDUCTION TARGET(LE, zOSV1R11) TEMPLATERECOMPILE
NOTEMPLATEREGISTRY TMPLPARSE(NO) TUNE(5) UNROLL(AUTO) NOWSIZEOF
NOXPLINK(NOBACKCHAIN, NOCALLBACK, NOGUARD, OSCALL(UPSTACK),
NOSTOREARGS) COMPILED_ON_MVS

 4 (    L) External Name: another_global
   (    L) External Name: some_global
   (    L) Function Name: myclass::myclass()
   (    L) Function Name: myclass::foo(float,double)
   (    L) Function Name: some_function(char)
```

```
*----------------------------------------------------------------------*
*  Member Name: CPPNOIPA                       (P) 2009/04/29 04:55:20 *
*                                                  5694A01    V1 R11   *
*----------------------------------------------------------------------*

   User Comment:
AGGRCOPY(NOOVERLAP) ANSIALIAS ARCH(5) ARGPARSE NOASCII ASSERT(RESTRICT)
BITFIELD(UNSIGNED) CHARS(UNSIGNED) NOCOMPACT NOCOMPRESS NOCONVLIT
NOCSECT CVFT NODEBUG NODFP DLL(NOCALLBACKANY) ENUMSIZE(SMALL) EXECOPS
EXH NOEXPORTALL FLOAT(HEX, FOLD, NOMAF, NORRM, AFP(NOVOLATILE)) NOGOFF
NOGONUMBER NOHGPR(NOPRESERVE) NOHOT NOIGNERRNO NOINITAUTO
NOINLINE(NOAUTO, NOREPORT, 100, 1000) NOIPA LANGLVL(ANONSTRUCT,
ANONUNION, ANSIFOR, ANSISINIT, C99VLA, C99__FUNC__, NODBCS,
DEPENDENTBASELOOKUP, NODOLLARINNAMES, EMPTYSTRUCT, ILLPTOM, IMPLICITINT
 LIBEXT, LONGLONG, NONEWEXCP, OFFSETNONPOD, NOOLDDIGRAPH, OLDFRIEND,
NOOLDMATH, NOOLDSTR, OLDTEMPACC, NOOLDTMPLALIGN, OLDTMPLSPEC, TRAILENUM
 TYPEDEFCLASS, NOUCS, GNU_INCLUDE_NEXT, ZEROEXTARRAY) NOLIBANSI
NOLOCALE LONGNAME ILP32 MAXMEM(2097152) NAMEMANGLING(zOSV1R2)
OBJECTMODEL(COMPAT) NOOPTIMIZE PLIST(HOST) PREFETCH REDIR ROCONST
ROSTRING ROUND(Z)  NORTCHECK NORTTI NOSERVICE SPILL(128) START STRICT
NOSTRICT_INDUCTION TARGET(LE, zOSV1R11) TEMPLATERECOMPILE
NOTEMPLATEREGISTRY TMPLPARSE(NO) TUNE(5) UNROLL(AUTO) NOWSIZEOF
NOXPLINK(NOBACKCHAIN, NOCALLBACK, NOGUARD, OSCALL(UPSTACK),
NOSTOREARGS) COMPILED_ON_MVS

   (    L) Function Name: myclass::myclass()
   (    L) Function Name: myclass::foo(float,double)
   (    L) Function Name: some_function(char)
   (   WL) External Name: another_global
   (   WL) External Name: some_global

*----------------------------------------------------------------------*
*  Member Name:  CPPLP64                       (P) 2009/04/29 04:55:20 *
*----------------------------------------------------------------------*

*----------------------------------------------------------------------*
*  Member Name:   CIPA64                       (P) 2009/04/29 04:55:22 *
*----------------------------------------------------------------------*

========================================================================
| 5                         Symbol Definition Map                      |
========================================================================


*----------------------------------------------------------------------*
| 6  Symbol Name: some_function(char)                                  |
*----------------------------------------------------------------------*

  From member: CPPIPANO Type: Function (    L)
  From member: CPPNOIPA Type: Function (    L)
```

```
*---------------------------------------------------------------------*
|  Symbol Name: myclass::myclass()                                    |
*---------------------------------------------------------------------*

  From member: CPPIPANO Type: Function (    L)
  From member: CPPNOIPA Type: Function (    L)

*---------------------------------------------------------------------*
|  Symbol Name: myclass::foo(float,double)                            |
*---------------------------------------------------------------------*

  From member: CPPIPANO Type: Function (    L)
  From member: CPPNOIPA Type: Function (    L)

*---------------------------------------------------------------------*
|  Symbol Name: some_global                                           |
*---------------------------------------------------------------------*

  From member: CPPIPANO Type: External (    L)
  From member: CPPNOIPA Type: External (   WL)

*---------------------------------------------------------------------*
|  Symbol Name: another_global                                        |
*---------------------------------------------------------------------*

  From member: CPPIPANO Type: External (    L)
  From member: CPPNOIPA Type: External (   WL)

=========  E N D   O F   O B J E C T   L I B R A R Y   M A P  =========
```

1 Map Heading
The heading contains the product number, the library version and release number, and the date and the time the Object Library Utility step began. The name of the library immediately follows the heading. To the right of the library name is the start time of the last object library utility step that updated the Object Library Utility directory.

2 Member Heading
The name of the object module member is immediately followed by the Timestamp field presented in *yyyy/mm/dd* format. The meaning of the timestamp is enclosed in parentheses. The object library utility retains a timestamp for each member and selects the time according to the following hierarchy:

(P) indicates that the compile timestamp is extracted from the object module.

(D) indicates that the timestamp is based on the time that the object library utility DIR command was last issued.

(T) indicates that the timestamp is the time that the ADD command was issued for the member.

The next line contains the ID of the processor that produced the object module. If the processor ID is not present, the `Processor ID` field is not listed.

3 User Comments
Displays any comments that were specified in the object module with the **#pragma comment** directive. It is possible to manually add such comments to the END records of an object member and have them displayed in the listing. These comments are extracted from the END record. The compile time options are stored in the same area as user comments and are displayed here as well.

4 Symbol Information

Immediately following Member Heading and user comments is a list of the defined objects that the member contains. Each symbol is prefixed by type information that is enclosed in parentheses and either External Name or Function Name. Function Name will appear, provided the object module was compiled with the LONGNAME option and the symbol is the name of a defined external function. In all other cases, External Name is displayed. The `Type` field gives the following additional information on each symbol:

6 indicates that the object was compiled with LP64

I indicates that the name is compiled IPA(NOOBJECT).

L indicates that the name is a long name. A long name is an external C++ name in an object module or an external non-C++ name in an object module produced by compiling with the LONGNAME option.

S indicates that the name is a short name. A short name is an external non-C++ name in an object module produced by compiling with the NOLONGNAME option. Such a name is up to 8 characters long and single case.

W indicates that this is a writable static object. If it is not present, then this is not a writable static object.

X indicates that the name was compiled with the XPLINK option.

5 Symbol Definition Map

This section of the listing has an entry for each unique symbol name that appeared in the previous half of the listing. Any duplicate symbol names that appear in the entire object library utility directory are grouped here for cross-reference purposes. This allows you to quickly determine which attributes a particular symbol name possesses within this object library utility directory.

6 Symbol Source List

Displays the object module(s) found by the given symbol. Symbol attributes (described under “Symbol Information” in this topic) immediately follow the names of the source objects.

Chapter 14. Filter utility

This information describes how to use the CXXFILT utility to convert C++ mangled names to demangled names, which are human-readable.

When z/OS XL C++ compiles one of your source files, it does not place the function and class names appearing in that file verbatim in the object file, as would occur if you were compiling a z/OS XL C program. Instead, it ″mangles″ them, which means it encodes your function names with their type and scoping information. This process is required to support the more advanced features of C++ such as inheritance and function overloading. Mangled names are used for type-safe linking.

Use the CXXFILT utility to convert these mangled names to demangled names. The utility copies the characters from either a given file or from standard input, to standard output. It replaces all mangled names with their corresponding demangled names.

The CXXFILT utility demangles any of the following classes of mangled names when the appropriate options are specified.

regular names
: Names that appear within the context of a function name or a member variable.

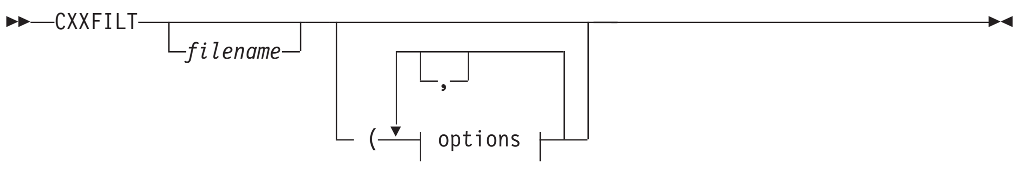
  Example: The mangled name `__ls__7ostreamFPCc` is demangled as `ostream::operator<<(const char*)`.

class names
: Includes stand-alone class names that do not appear within the context of a function name or a member variable.

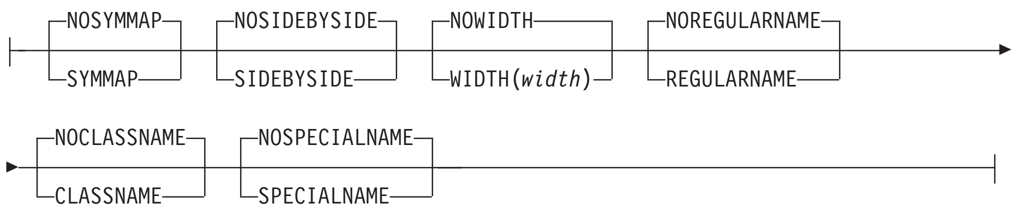
  Example: For example, the stand-alone class name `Q2_1X1Y` is demangled as `X::Y`

special names
: Special compiler-generated class objects.

  Example: For example, the compiler-generated symbol name `__vft1X` is demangled as `X::virtual-fn-table-ptr`.



options:



The *filename* refers to the files that contain the mangled names to be demangled. You may specify more than one file name, which can be a sequential file or a PDS member. If you do not specify a file name, CXXFILT reads its input from `stdin`.

The following topic describes the options that you can use with the CXXFILT utility.

CXXFILT options

You can use the following options with CXXFILT.

SYMMAP | NOSYMMAP

Default: NOSYMMAP

Produces a symbol map on standard output. This map contains a list of the mangled names and their corresponding demangled names. The map only displays the first 40 bytes of each demangled name; it truncates the rest. Mangled names are not truncated.

If an input mangled name does not have a demangled version, the symbol mapping does not display it.

The symbol mapping is displayed after the end of the input stream is encountered, and after CXXFILT terminates.

SIDEBYSIDE | NOSIDEBYSIDE

Default: NOSIDEBYSIDE

Each mangled name that is encountered in the input stream is displayed beside its corresponding demangled name. If you do not specify this option, then only the demangled names are printed. In either case, trailing characters in the input name that are not part of a mangled name appear next to the demangled name. For example, if an extraneous `xxxx` is input with the mangled name `pr__3FOOF`, then the SIDEBYSIDE option would produce this result:

```
FOO::pr()       pr__3FOOFvxxxx
```

WIDTH(width) | NOWIDTH

Default: NOWIDTH

Prints demangled names in fields, *width* characters wide. If the name is shorter than *width*, it is padded on the right with blanks; if longer, it is truncated to *width*. The value of *width* must be greater than 0. If *width* is greater than the record width, then the output is wrapped.

REGULARNAME | NOREGULARNAME

Default: REGULARNAME

This option demangles regular names such as `pr__3FOOFv` to `FOO:pr()`.

The mangled name that is supplied to CXXFILT is treated as a regular name by default. Specifying the NOREGULARNAME option will turn the default off. For example, specifying the CLASSNAME option without the NOREGULARNAME option will cause CXXFILT to treat the mangled name as either a regular name or stand-alone class name.

CLASSNAME | NOCLASSNAME

Default: NOCLASSNAME

This option demangles stand-alone class names such as Q2_1X1Y to `X::Y`.

To request that the mangled names be treated as stand-alone class names only, and never as a regular name, use both CLASSNAME and NOREGULARNAME.

SPECIALNAME | NOSPECIALNAME

Default: NOSPECIALNAME

This option demangles special names, such as compiler-generated symbol names; for example, `__vft1X` is demangled to `X::virtual-fn-table-ptr`.

To request that the mangled names be treated as special names only, and never as regular names, use CXXFILT (SPECIALNAME NOREGULARNAME.

Unknown type of name

If you cannot specify the type of name, use CXXFILT (SPECIALNAME CLASSNAME. This causes CXXFILT to attempt to demangle the name in the following order:
1. Regular name
2. Stand-alone class name
3. Special name

Under z/OS batch

The CXXFILT utility accepts input by two methods: from `stdin` or from a file.

Example: The following example uses the CXXFILT cataloged procedure, from data set CBC.SCBCPRC. CXXFILT reads from `stdin` (`sysin`), treats mangled names as regular names, produces a symbol mapping, and uses a field width 15 characters. The JCL follows:

```
//RUN EXEC CXXFILT,CXXPARM='(SYMMAP WIDTH(15)'
⋮
//SYSIN DD *
pr__3FOOFvxxxx
__ls__7ostreamFPCc
__vft1X
/*
```

The output is:

```
FOO::pr()      xxxx
ostream::operator<<(const char*)
__vft1X


C++ Symbol Mapping

demangled                                mangled
---------                                -------
FOO::pr()                                pr__3FOOFv
ostream::operator<<(const char*)         __ls__7ostreamFPCs
```

Notes:

1. Because the trailing characters `xxxx` in the input name `pr__3FOOFvxxxx` are not part of a valid mangled name, and the SIDEBYSIDE option is not on, the trailing characters are not demangled.

   Note: In the symbol mappings, the trailing characters `xxxx` are *not* displayed.
2. The `_vft1X` input is not demangled and does not appear in the symbol mapping because it is a special name, and the SPECIALNAME option was not specified.

The second method of giving input to CXXFILT is to supply it in one or more files. Fixed and variable file record formats are supported. Each line of a file can have one or more names separated by space. In the example below, mangled names are treated either as regular names or as special names (the special names are compiler-generated symbol names). Demangled names are printed in fields 35 characters wide, and output is in side-by-side format.

The output contains the following two mangled names:

```
pr__3FOOFv
__vft1X
```

You can use the following JCL:

```
//RUN EXEC CXXFILT,CXXPARM='FILE1 (SPECIALNAME WIDTH(35) SIDEBYSIDE'
```

The CXXFILT utility terminates when it reads the end-of-file terminator.

Under TSO

The CXXFILT utility accepts input by two methods: from `stdin` or from a file.

With the first method, enter names after invoking CXXFILT. You can specify one or more names on one or more lines. The output is displayed after you press Enter. Names that are successfully demangled, as well as those which are not demangled, are displayed in the same order as they were entered. To indicate end of input, enter `/*`.

Example: In the following example, CXXFILT treats mangled names as regular names, produces a symbol mapping, and uses a field width 15 characters wide.

```
user>  CXXFILT (SYMMAP WIDTH(15)
user>  pr__3FOOFvxxxx
reply< FOO::pr()      xxxx
user>  __ls__7ostreamFPCc
reply> ostream::operator<<(const char*)
user>  __vft1X
reply> __vft1X
user>  /*

reply> C++ Symbol Mapping
reply>
reply> demangled                                     mangled
reply> ---------                                     -------
reply> FOO::pr()                                     pr__3FOOFv
reply> ostream::operator<<(const char*)              __ls__7ostreamFPCs
```

Notes:

1. Because the trailing characters `xxxx` in the input name `pr__3FOOFvxxxx` are not part of a valid mangled name, and the SIDEBYSIDE option is not on, the trailing characters are not demangled.

In the symbol mappings, the trailing characters xxxx are *not* displayed.

2. The __vft1X input is not demangled and does not appear in the symbol mapping because it is a special name, and the SPECIALNAME option was not specified.
3. The symbol mapping is displayed only after /* requests CXXFILT termination

The second method of giving input to CXXFILT is to supply it in one or more files. CXXFILT supports fixed and variable file record formats. Each line of a file can have one or more names separated by space. In the example below, mangled names are treated either as regular names or as special names (the special names are compiler-generated symbol names). Demangled names are printed in fields 35 characters wide, and output is in side-by-side format.

The output contains the following two mangled names:

```
pr__3FOOFv
__vft1X
```

Example: Enter the following command:

```
cxxfilt FILE1 (SPECIALNAME WIDTH(35) SIDEBYSIDE
```

This will produce the output:

```
FOO::pr()                          pr__3FOOFv
X::virtual-fn-table-ptr            __vft1X
```

CXXFILT terminates when it reads the end-of-file terminator.

Chapter 15. DSECT conversion utility

This information describes how to use the DSECT conversion utility, which generates a structure to map an assembler DSECT. This utility is used when a C or C++ program calls or is called by an assembler program, and a structure is required to map the area passed.

You assemble the source for the assembler DSECT by using the High Level Assembler, and specifying the ADATA option. (See *HLASM Programmer's Guide*, for a description of the ADATA option.) The DSECT utility then reads the SYSADATA file that is produced by the High Level Assembler and produces a file that contains the equivalent C structure according to the options specified.

| **Note:** The SYSADATA file can be either a PDS member, PDSE member or a
| sequential file, but cannot be a z/OS UNIX System Services file.

DSECT Utility options

The options that you can use to control the generation of the C or C++ structure are as follows. You can specify them in uppercase or lowercase, separating them by spaces or commas.

Table 46. DSECT Utility options, abbreviations, and IBM-supplied defaults

DSECT Utility Option	Abbreviated Name	IBM-supplied Default
SECT[(*name*,...)]	None	SECT(ALL)
BITF0XL | NOBITF0XL	BITF | NOBITF	NOBITF0XL
COMMENT[(*delim*,...)] | NOCOMMENT	COM | NOCOM	COMMENT
DECIMAL | NODECIMAL	None	NODECIMAL
DEFSUB | NODEFSUB	DEF | NODEF	DEFSUB
EQUATE[(*suboptions*,...)] | NOEQUATE	EQU | NOEQU	NOEQUATE
HDRSKIP[(*length*)] | NOHDRSKIP	HDR(*length*) | NOHDR	NOHDRSKIP
LOCALE(*name*) | NOLOCALE	LOC | NOLOC	NOLOCALE
INDENT[(*count*)] | NOINDENT	IN(*count*) | NOIN	INDENT(2)
LOWERCASE> | NOLOWERCASE	LC | NOLC	LOWERCASE>
OPTFILE(*filename*) | NOOPTFILE	OPTF | NOOPTF	NOOPTFILE
PPCOND[(*switch*)] | NOPPCOND	PP(*switch*) | NOPP	NOPPCOND
SEQUENCE | NOSEQUENCE	SEQ | NOSEQ	NOSEQUENCE
UNIQUE|NOUNIQUE	None	NOUNIQUE
UNNAME | NOUNNAMED	UNN | NOUNN	NOUNNAMED
OUTPUT[(*filename*)]	OUT[(*filename*)]	OUTPUT (DD:EDCDSECT)
RECFM[(*recfm*)]	None	C/C++ Library defaults
LRECL[(*lrecl*)]	None	C/C++ Library defaults
BLKSIZE[(*blksize*)]	None	C/C++ Library defaults
LP64	None	NOLP64

SECT

DEFAULT: SECT(ALL)

The SECT option specifies the section names for which structures are produced. The section names can be either CSECT or DSECT names. They must exist in the SYSADATA file that is produced by the assembler. If you do not specify the SECT option or if you specify SECT(ALL), structures are produced for all CSECTs and DSECTs defined in the SYSADATA file, except for private code and unnamed DSECTs.

If the High Level Assembler is run with the BATCH option, only the section names defined within the first program can be specified on the SECT option. If you specify SECT(ALL) (or select it by default), only the sections from the first program are selected.

BITF0XL | NOBITF0XL

DEFAULT: NOBITF0XL

Specify the BITF0XL option when the bit fields are mapped into a flag byte as in the following example:

```
FLAGFLD   DS   F
          ORG  FLAGFLD+0
B1FLG1    DC   0XL(B'10000000')'00'    Definition for bit 0 of 1st byte
B1FLG2    DC   0XL(B'01000000')'00'    Definition for bit 1 of 1st byte
B1FLG3    DC   0XL(B'00100000')'00'    Definition for bit 2 of 1st byte
B1FLG4    DC   0XL(B'00010000')'00'    Definition for bit 3 of 1st byte
B1FLG5    DC   0XL(B'00001000')'00'    Definition for bit 4 of 1st byte
B1FLG6    DC   0XL(B'00000100')'00'    Definition for bit 5 of 1st byte
B1FLG7    DC   0XL(B'00000010')'00'    Definition for bit 6 of 1st byte
B1FLG8    DC   0XL(B'00000001')'00'    Definition for bit 7 of 1st byte
          ORG  FLAGFLD+1
B2FLG1    DC   0XL(B'10000000')'00'    Definition for bit 0 of 2nd byte
B2FLG2    DC   0XL(B'01000000')'00'    Definition for bit 1 of 2nd byte
B2FLG3    DC   0XL(B'00100000')'00'    Definition for bit 2 of 2nd byte
B2FLG4    DC   0XL(B'00010000')'00'    Definition for bit 3 of 2nd byte
```

When the bit fields are mapped as shown in this example, you can use the following code to test the bit fields:

```
TM   FLAGFLD,L'B1FLG1         Test bit 0 of byte 1
Bx   label                    Branch if set/not set
```

When you specify the BITF0XL option, the length attribute of the following fields provides the mapping for the bits within the flag bytes.

The length attribute of the following fields is used to map the bit fields if a field conforms to the following rules:

- The field does not have a duplication factor of zero.
- The field has a length between 1 and 4 bytes and does not have a bit length.
- The field does not have more than one nominal value.

and the following fields conform to the following rules:

- Has a Type attribute of `B`, `C`, or `X`.
- Has the same offset as the field (or consecutive fields have overlapping offsets).
- Has a duplication factor of zero.
- Does not have more than one nominal value.
- Has a length attribute between 1 and 255 and does not have a bit length.
- The length attribute maps one bit or consecutive bits; for example, B'10000000' or B'11000000', but not B'10100000'.

The fields must be on consecutive lines and must overlap a named field. If these fields are used to define the bits for a field, EQU statements that follow the field are not used to define the bit fields.

COMMENT | NOCOMMENT

DEFAULT: COMMENT

The COMMENT option specifies whether the comments on the line where the field is defined will be placed in the structure produced.

If you specify the COMMENT option without a delimiter, the entire comment is placed in the structure.

If you specify a delimiter, any comments that follow the delimiter are skipped and are not placed in the structure. You can remove changes that are flagged with a particular delimiter. The delimiter cannot contain imbedded spaces or commas. The case of the delimiter and the comment text is not significant. You can specify up to 10 delimiters, and they can contain up to 10 characters each.

DECIMAL | NODECIMAL

DEFAULT: NODECIMAL

The DECIMAL option will instruct the DSECT utility to convert all SYSATADA DC/DS records of type P to the function type macro: **_dec__var(w,0)**. **w** is the number of digits and it is computed by taking the byte size of the P-type data, multiplying it by two, and subtracting one from the result [in other words, (byte_size * 2)-1]. The byte size of the P type data is found in the SYSADATA DC/DS record. If a SYSADATA DC/DS record of type P is interpreted to be part of a union then the DSECT utility will map it to the function type macro: **_dec__uvar(w,0)**. **w** still represents the number of digits. The **_dec__uvar** macro will expand to a decimal datatype for C and a unsigned character array for C++. This is necessary because decimal support in C++ is implemented by a decimal class. C++ does not allow a class with constructors, or deconstructors, to be part of a union, hence in the case of C++ such decimal data must be mapped to a character array of the same byte size.

The precision will always be left as zero since there is no way to figure out its value from the DC/DS SYSADATA record. The zero will be output, rather than just the digit size (that is, **_dec__var(w,0)** rather than just **_dec__var(w,))**), to allow you to easily edit the DSECT utility output and adjust for the desired precision. Do not remove the zero as it will cause compilation errors because the function type macros can no longer be expanded.

If the DECIMAL option is enabled and P type records are found, then the utility will also include the following code at the beginning of the output file:

```
#ifndef __decimal_found
     #define __decimal_found
     #ifdef __cplusplus
         #define _dec__var(w,p)  decimal<n>
             #define _dec_uvar(w,p) _decchar##w
         #include <idecimal.hpp>
             typedef char _decchar1[1];
             typedef char _decchar2[2];
             typedef char _decchar3[2];
             typedef char _decchar4[3];
             typedef char _decchar5[3];
             typedef char _decchar6[4];
```

```
            typedef char _decchar7[4];
            typedef char _decchar8[5];
            typedef char _decchar9[5];
            typedef char _decchar10[6];
            typedef char _decchar11[6];
            typedef char _decchar12[7];
            typedef char _decchar13[7];
            typedef char _decchar14[8];
            typedef char _decchar15[8];
            typedef char _decchar16[9];
            typedef char _decchar17[9];
            typedef char _decchar18[10];
            typedef char _decchar19[10];
            typedef char _decchar20[[11];
            typedef char _decchar21[11];
            typedef char _decchar22[12];
            typedef char _decchar23[12];
            typedef char _decchar24[13];
            typedef char _decchar25[13];
            typedef char _decchar26[14];
            typedef char _decchar27[14];
            typedef char _decchar28[15];
            typedef char _decchar29[15];
            typedef char _decchar30[16];
            typedef char _decchar31[16];
    #else
        #define _dec__var(w,p)  decimal(n,p)
            #define _dec_uvar(w,p) decimal(w,p)
        #include <decimal.h>
    #endif
#endif
```

This code will force the inclusion of the necessary header files, depending on whether the C or C++ compiler is used. It will also force the **_dec__var** and **_dec_uvar** types, which are outputted by the DSECT utility, to be mapped to the appropriate C or C++ decimal type. The definition of the macro **__decimal_found** is used to guard against the redefinition of macros if several DSECT utility output files are compiled together.

If the default NODECIMAL option is used then the DSECT utility will convert all P type DC/DS SYSATADA records to character arrays of the same byte size as the P type data, as is the existing behavior; for example, 171 (a value of PL3) will map to an `unsigned char[3]`.

DEFSUB | NODEFSUB

DEFAULT: DEFSUB

The DEFSUB option specifies whether `#define` directives will be built for fields that are part of a union or substructure.

Example: If the DEFSUB option is in effect, fields within a substructure or union have the field names prefixed by an underscore. A `#define` directive is written at the end of the structure to allow the field name to be specified directly as in the following example.

```
struct dsect_name {
  int           field1;
  struct {
    int           _subfld1;
    short int     _subfld2;
    unsigned char _subfld3[4];
    } field2;
  }
#define subfld1  field2._subfld1
#define subfld2  field2._subfld2
#define subfld3  field2._subfld3
```

If the DEFSUB option is in effect, the fields that are prefixed by an underscore may match the name of another field within the structure. No warning is issued.

EQUATE | NOEQUATE

DEFAULT: NOEQUATE

The EQUATE option specifies whether the EQU statements following a field are to be used to define bit fields, to generate `#define` directives, or are to be ignored.

The suboptions specify how the EQU statement is used. You can specify one or more of the suboptions, separating them by spaces or commas. If you specify more than one suboption, the EQU statements that follow a field are checked to see if they are valid for the first suboption. If so, they are formatted according to that option. Otherwise, the subsequent suboptions are checked to see if they are applicable.

If you specify the EQUATE option without suboptions, EQUATE(BIT) is used. If you specify NOEQUATE (or select it by default), the EQU statements that follow a field are ignored.

You can specify the following suboptions for the EQUATE option:

BIT Indicates that the value for an EQU statement is used to define the bits for a field where the field conforms to the following rules:
- The field does not have a duplication factor of zero.
- The field has a length between 1 and 4 bytes and has a bit length that is a multiple of 8.
- The field does not have more than one nominal value.

and the EQU statements that follow the field conform to the following rules:
- The value for the EQU statements that follow the field mask consecutive bits (for example, X’80’ followed by X’40’).
- The value for an EQU statement masks one bit or consecutive bits for example, B’10000000’ or B’11000000’, but not B’10100000’.
- Where the length of the field is greater than 1 byte, the bits for the remaining bytes can be defined by providing the EQU statements for the second byte after the EQU statement for the first byte.
- The value for the EQU statement is not a relocatable value.

Example: When you specify EQUATE(BIT), the EQU statements are converted as in the following example:

```
FLAGFLD  DS   H
FLAG21   EQU  X'80'
FLAG22   EQU  X'40'
FLAG23   EQU  X'20'
FLAG24   EQU  X'10'
FLAG25   EQU  X'08'
FLAG26   EQU  X'04'
FLAG27   EQU  X'02'
FLAG28   EQU  X'01'
FLAG2A   EQU  X'80'
FLAG2B   EQU  X'40'
struct dsect_name {
  unsigned int flag21   : 1,
               flag22   : 1,
               flag23   : 1,
               flag24   : 1,
               flag25   : 1,
               flag26   : 1,
               flag27   : 1,
               flag28   : 1,
               flag2a   : 1,
               flag2b   : 1,
                        : 6;
  }
```

BITL Indicates that the length attribute for an EQU statement is used to define the bits for a field where the field conforms to the following rules:

- The field does not have a duplication factor of zero.
- The field has a length between 1 and 4 bytes and has a bit length that is a multiple of 8.
- The field does not have more than one nominal value.

and the EQU statements that follow the field conform to the following rules:

- The value that is specified for the EQU statement has the same or overlapping offset as the field.
- The length attribute for the EQU statement is between 1 and 255.
- The length attribute for the EQU statement masks one bit or consecutive bits; for example, B'10000000' or B'11000000', but not B'10100000'.
- The value for the EQU statement is a relocatable value.

Example: When you specify EQUATE(BITL), the EQU statements are converted as in the following example:

```
BYTEFLD   DS   F
B1FLG1    EQU  BYTEFLD+0,B'10000000'
B1FLG2    EQU  BYTEFLD+0,B'01000000'
B1FLG3    EQU  BYTEFLD+0,B'00100000'
B1FLG4    EQU  BYTEFLD+0,B'00010000'
B1FLG5    EQU  BYTEFLD+0,B'00001000'
B1FLG6    EQU  BYTEFLD+0,B'00000100'
B1FLG7    EQU  BYTEFLD+0,B'00000010'
B1FLG8    EQU  BYTEFLD+0,B'00000001'
B2FLG1    EQU  BYTEFLD+1,B'10000000'
B2FLG2    EQU  BYTEFLD+1,B'01000000'
B2FLG3    EQU  BYTEFLD+1,B'00100000'
B2FLG4    EQU  BYTEFLD+1,B'00010000'
struct dsect_name {
  unsigned int b1flg1  : 1,
               b1flg2  : 1,
               b1flg3  : 1,
               b1flg4  : 1,
               b1flg5  : 1,
               b1flg6  : 1,
               b1flg7  : 1,
               b1flg8  : 1,
               b2flg1  : 1,
               b2flg2  : 1,
               b2flg3  : 1,
               b2flg4  : 1,
                       : 20;
  }
```

DEF Indicates that the EQU statements following a field are used to build `#define` directives to define the possible values for a field. The `#define` directives are placed after the end of the structure. The EQU statements should not specify a relocatable value.

Example: When you specify EQUATE(DEF), the EQU statements are converted as in the following example:

```
FLAGBYTE DS   X
FLAG1    EQU  X'80'
FLAG2    EQU  X'20'
FLAG3    EQU  X'10'
FLAG4    EQU  X'08'
FLAG5    EQU  X'06'
FLAG6    EQU  X'01'
struct dsect_name {
  unsigned char flagbyte;
  }
/* Values for flagbyte field */
#define flag1 0x80
#define flag2 0x20
#define flag3 0x10
#define flag4 0x08
#define flag5 0x06
#define flag6 0x01
```

HDRSKIP | NOHDRSKIP

DEFAULT: NOHDRSKIP

The HDRSKIP option specifies that the fields within the specified number of bytes from the start of the section are to be skipped. Use this option where a section has a header that is not required in the structure produced.

The value that is specified on the HDRSKIP option indicates the number of bytes at the start of the section that are to be skipped. HDRSKIP(0) is equivalent to NOHDRSKIP.

Example: In the following example, if you specify HDRSKIP(8), the first two fields are skipped and only the remaining two fields are built into the structure.

```
SECTNAME DSECT
PREFIX1  DS   CL4
PREFIX2  DS   CL4
FIELD1   DS   CL4
FIELD2   DS   CL4
struct sectname {
  unsigned char field1[4];
  unsigned char field2[4];
  }
```

If the value specified for the HDRSKIP option is greater than the length of the section, the structure is not be produced for that section.

INDENT | NOINDENT

DEFAULT: INDENT(2)

The INDENT option specifies the number of character positions that the fields, unions, and substructures are indented. Turn off indentation by specifying INDENT(0) or NOINDENT. The maximum value that you can specify for the INDENT option is 32767.

LOCALE | NOLOCALE

The LOCALE(*name*) specifies the name of a locale to be passed to the `setlocale()` function. Specifying LOCALE without the *name* parameter is equivalent to passing the NULL string to the `setlocale()` function.

The structure produced contains the left and right brace, and left and right square bracket, backslash, and number sign which have different code point values for the different code pages. When the LOCALE option is specified, and these characters are written to the output file, the code point from the LC_SYNTAX category for the specified locale is used.

The default is NOLOCALE.

You can abbreviate the option to LOC(*name*) or NOLOC.

LOWERCASE | NOLOWERCASE

DEFAULT: LOWERCASE>

The LOWERCASE> option specifies whether the field names within the C structure are to be converted to lowercase or left as entered. If you specify LOWERCASE>, all the field names are converted to lowercase. If you specify NOLOWERCASE, the field names are built into the structure in the case in which they were entered in the assembler section.

LP64 | NOLP64

DEFAULT: NOLP64

The equivalent of NOLP64 for the compiler is the option ILP32, which means 32-bit integer, `long`, and pointer type. This is the default in the compiler as well. LP64 means 64-bit `long` and pointer type. LP64 and ILP32 specify the data model for the programming language.

The LP64 option instructs the DSECT utility to generate structures for use by the programs compiled with the LP64 option. When this option is enabled, address fields are mapped to C pointer types (64 bits), and 64-bit integer fields are mapped to `long` data types. C/C++ also supports a `__ptr32` qualifier for declaring pointers that are 32-bit in size, which means that if a field is explicitly specified with a 31-bit address, it is mapped to a `__ptr32` qualified pointer.

OPTFILE | NOOPTFILE

The OPTFILE(*filename*) option specifies the *filename* that contains the records that specify the options to be used for processing the sections. The records must be as follows:

- The lines must begin with the SECT option, and only one section name must be specified. The options following determine how the structure is produced for the specified section. The section name must only be specified once.
- The lines may contain the BITF0XL, COMMENT, DEFSUB, EQUATE, HDRSKIP, INDENT, LOWERCASE, PPCOND, and UNNAMED options, separated by spaces or commas. These override the options that are specified on the command line for the section.

The OPTFILE option is ignored if the SECT option is also specified on the command line.

The default is NOOPTFILE.

You can abbreviate the option to OPTF(*filename*) or NOOPTF.

PPCOND | NOPPCOND

DEFAULT: NOPPCOND

The PPCOND option specifies whether preprocessor directives will be built around the structure definition to prevent duplicate definitions.

If you specify PPCOND, the following are built around the structure definition.

```
#ifndef switch
#define switch
   :
  structure definition for section
   :
#endif
```

where *switch* is the switch specified on the PPCOND option or the section name prefixed and suffixed by two underscores; for example, _ _name_ _.

If you specify a switch, the `#ifndef` and `#endif` directives are placed around all structures that are produced. If you do not specify a switch, the `#ifndef` and `#endif` directives are placed around each structure produced.

SEQUENCE | NOSEQUENCE

DEFAULT: NOSEQUENCE

The SEQUENCE option specifies whether sequence numbers will be placed in columns 73 to 80 of the output record. If you specify the SEQUENCE option, the structure is built into columns 1 to 72 of the output record, and sequence numbers are placed in columns 73 to 80. If you specify NOSEQUENCE (or select it by default), sequence numbers are not generated, and the structure is built within all available columns in the output record.

If the record length for the output file is less than 80 characters, the SEQUENCE option is ignored.

UNIQUE | NOUNIQUE

DEFAULT: NOUNIQUE

The UNIQUE option tells the DSECT utility to consider the given unique string as not occurring in any field names in the input SYSADATA. This is necessary because it is a guarantee from the user that if the DSECT utility were to use the unique string to map national characters, no conflict would occur with any other field name. Given this guarantee the DSECT utility maps national characters as follows:

```
# = unique string + 'n' + unique string
@ = unique string + 'a' + unique string
$ = unique string + 'd' + unique string
```

Example: If the default ″_″ unique string was used then the national characters would be mapped as:

```
# = _n_
@ = _a_
$ = _d_
```

If the default NOUNIQUE option is enabled, the DSECT utility converts all national characters to a single underscore, even if the resulting label names conflict (as is the existing behavior).

Note: If the DSECT utility detects a field name that has a length that exceeds the maximum allowed, a message is displayed and the name is truncated in the output. This can happen due to the substitution characters in the UNIQUE option. That is, the field name as specified by you is within the maximum limit, but due to the presence of national characters and the mapping done by UNIQUE, the resulting field name can exceed the limit. The DSECT utility then ends the output field name with ″...″ to make it easy to find. You should check and fix the field name either by changing the UNIQUE option, or by shortening the original field name, or both.

UNNAMED | NOUNNAMED

DEFAULT: NOUNNAMED

The UNNAME option specifies that names are not generated for the unions and substructures within the main structure.

OUTPUT

DEFAULT: OUTPUT(`DD:EDCDSECT`)

The structures that are produced are, by default, written to the EDCDSECT DD statement. You can use the OUTPUT option to specify an alternative DD statement or data set name to write the structure. You can specify any valid file name up to 60 characters in length. The file name specified will be passed to fopen() as entered.

RECFM

DEFAULT: C/C++ Library default

The RECFM option specifies the record format for the file to be produced. You can specify up to 10 characters. If it is not specified, the C or C++ library defaults are used.

LRECL

DEFAULT: C/C++ Library default

The LRECL option specifies the logical record length for the file to be produced. The logical record length that is specified must not be greater than 32767. If it is not specified, the C or C++ library defaults will be used.

BLKSIZE

DEFAULT: C/C++ Library default

The BLKSIZE option specifies the block size for the file to be produced. The block size that is specified must not be greater than 32767. If it is not specified, the C or C++ library defaults will be used.

Generation of structures

The structure is produced as follows according to the options in effect.

- The section name is used as the structure name. A **#pragma pack(packed)** is generated at the top of the file, and a **#pragma pack(reset)** is generated at the end so that the structure matches the assembler section. For example:

```
#pragma pack(packed)
struct dsect_name {
   ⋮
  };
#pragma pack(reset)
```

- Any nonalphanumeric characters in the section or field names are converted to underscores. Duplicate names may be generated when the field names are identical except for the national character. No warning is issued.
- Where fields overlap, a substructure or union is built within the main structure. A substructure is produced where possible. When substructures and unions are built, the DSECT utility generates the structure and union names.
- The substructures and unions within the main structure are indented according to the INDENT option unless the record length is too small to permit any further indentation.
- Fillers are added within the structure when required. The DSECT utility generates a filler name.
- Where there is no direct equivalent for an assembler definition within the C or C++ language, the field is defined as a character field.
- If a field has a duplication factor of zero, but cannot be used as a structure name, the field is defined as though the duplication factor of zero was eliminated.
- Where a line within the assembler input consists of an operand with a duplication factor of zero (for alignment), followed by the field definition, the first operand is skipped. For example:

  ```
  FIELDA    DS  OF,CLB
  ```

  is treated as though the following was specified.

  ```
  FIELDA    DS  CLB
  ```

- When the COMMENT option is in effect, the comment on the line that follows the definition of the field is placed in the structure. The comment is placed on the same line as the field definition where possible, or on the following line.

  /* is removed from the beginning of comments, and */ is removed from the end of comments. Any remaining instances of */ in the comment are converted to **.

Each field within the section is converted to a field within the structure, as the following examples show:

- Bit length fields

  If the field has a bit length that is not a multiple of 8, it is converted as follows. Otherwise, it is converted according to the field type.

  DS CL.n `unsigned int   name : n;` where `n` is from 1 to 31.

  DS CL.n `unsigned char  name[x];` where `n` is greater than 32. `x` will be the number of bytes that are required (that is, the bit length / 8 + 1).

  DS 5CL.n `unsigned char  name[x];` where `x` will be the number of bytes required (that is, the duplication factor * bit length / 8 + 1).

- Characters

DS C	`unsigned char name;`
DS CL2	`unsigned char name[2];`
DS 4CL2	`unsigned char name[4][2];`

- Graphic Characters

DS G	`wchar_t       name;`
DS GL1	`unsigned char name;`
DS GL2	`wchar_t       name;`
DS GL3	`unsigned char name[3];`

DS 4GL1	`unsigned char name[4];`
DS 4GL2	`wchar_t       name[4];`
DS 4GL3	`unsigned char name[4][3];`

- Hexadecimal Characters

DS X	`unsigned char name;`
DS XL2	`unsigned char name[2];`
DS 4XL2	`unsigned char name[4][2];`

- Binary fields

DS B	`unsigned char  name;`
DS BL2	`unsigned char  name[2];`
DS 4BL2	`unsigned char  name[4][2];`

- Half and Fullword Fixed-point

DS F	`int`	`name;`
DS H	`short int`	`name;`
DS FL1 or HL1	`char`	`name;`
DS FL2 or HL2	`short int`	`name;`
DS FL3 or HL3	`int`	`name : 24;`
DS FLn or HLn	`unsigned char`	`name[n];` where `n` is greater than 4.
DS 4F	`int`	`name[4];`
DS 4H	`short int`	`name[4];`
DS 4FL1 or 4HL1	`char`	`name[4];`
DS 4FL2 or 4HL2	`short int`	`name[4];`
DS 4FL3 or 4HL3	`unsigned char`	`name[4][3];`
DS 4FLn or 4HLn	`unsigned char`	`name[4][n];` where `n` is greater than 4.

- Floating Point

DS E	`float`	`name;`
DS D	`double`	`name;`
DS L	`long double`	`name;`
DS 4E	`float`	`name[4];`
DS 4D	`double`	`name[4];`
DS 4L	`long double`	`name[4];`
DS EL4 or DL4 or LL4	`float`	`name;`
DS EL8 or DL8 or LL8	`double`	`name;`
DS LL16	`long double`	`name;`
DS E, D or L	`unsigned char`	`name[n];` where `n` is other than 4, 8, or 16.

- Packed Decimal

DS P	`unsigned char  name;`
DS PL2	`unsigned char  name[2];`
DS 4PL2	`unsigned char  name[4][2];`

- Zoned Decimal

DS Z	`unsigned char  name;`
DS ZL2	`unsigned char  name[2];`
DS 4ZL2	`unsigned char  name[4][2];`

- Address

DS A	`void          *name;`
DS AL1	`unsigned char  name;`
DS AL2	`unsigned short name;`
DS AL3	`unsigned int   name : 24;`
DS 4A	`void          *name[4];`
DS 4AL1	`unsigned char  name[4];`

DS 4AL2	`unsigned short name[4];`
DS 4AL3	`unsigned char  name[4][3];`

- Y-type Address

DS Y	`unsigned short name;`
DS YL1	`unsigned char  name;`
DS 4Y	`unsigned short name[4];`
DS 4YL1	`unsigned char  name[4];`

- S-type Address (Base and displacement)

DS S	`unsigned short name;`
DS SL1	`unsigned char  name;`
DS 4S	`unsigned short name[4];`
DS 4SL1	`unsigned char  name[4];`

- External Symbol Address

DS V	`void          *name;`
DS VL3	`unsigned int   name : 24;`
DS 4V	`void          *name[4];`
DS 4VL3	`unsigned char  name[4][3];`

- External Dummy Section Offset

DS Q	`unsigned int   name;`
DS QL1	`unsigned char  name;`
DS QL2	`unsigned short name;`
DS QL3	`unsigned int   name : 24;`
DS 4Q	`unsigned int   name[4];`
DS 4QL1	`unsigned char  name[4];`
DS 4QL2	`unsigned short name[4];`
DS 4QL3	`unsigned char  name[4][3];`

- Channel Command Words

  When a CCW, CCW0, or CCW1 assembler instruction is present within the section, a typedef `ccw0_t` or `ccw1_t` is defined to map the format of the CCW.

  The CCW, CCW0, or CCW1 is built into the structure as follows:

CCW cc,addr,flags,count	`ccw0_t`	`name;`
CCW0 cc,addr,flags,count	`ccw0_t`	`name;`
CCW1 cc,addr,flags,count	`ccw1_t`	`name;`

Under z/OS batch

Example: You can use the IBM-supplied cataloged procedure EDCDSECT to execute the DSECT utility as in the following example.

```
KNOWN:   - The assembler source name is FRED.SOURCE(TESTASM).
         - The structure is to be written to FRED.INCLUDE(TESTASM).
         - The required DSECT Utility options are EQU(BIT).

USE THE FOLLOWING JCL:
         //DSECT    EXEC PROC=EDCDSECT,
         //         INFILE='FRED.SOURCE(TESTASM)',
         //         OUTFILE='FRED.INCLUDE(TESTASM)',
         //         DPARM='EQU(BIT)'
```

Figure 45. Running the DSECT Utility under z/OS batch

EDCDSECT invokes the High Level Assembler to assemble the source that is provided with the ADATA option. It then executes the DSECT utility to produce the structure. It writes the structure to the data set that is specified by the OUTFILE parameter, unless the OUTPUT option is also specified. A report that indicates the options in effect and any error messages is written to SYSOUT.

If the assembler source requires macros or copy members from a macro library, include them on the SYSLIB DD for the ASSEMBLY step.

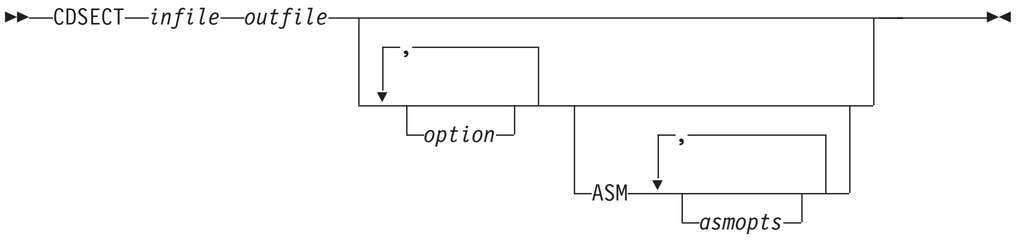
The parameters to the EDCDSECT procedure are:

Table 47. EDCDSECT procedure parameters

Parameter	Description
`INFILE`	Input assembler source data set name. This option must be provided.
`OUTFILE`	The data set name for the file into which the structure is written. If you do not specify an OUTFILE name, a temporary data set is generated.
`APARM`	High Level Assembler options.
`DPARM`	DSECT Utility options.

Under TSO

If you have REXX installed, you can run the DSECT utility under TSO by using the CDSECT EXEC. The format of the parameters for the CDSECT EXEC is:



where *infile* specifies the file name of the assembler source program containing the required section. *outfile* specifies the file that the structure produced is written to, and *option* is any valid DSECT utility option. If you specify ASM, any following options must be High-Level Assembler options. The ADATA is specified by default.

```
  KNOWN:   - The assembler source name is FRED.SOURCE(TESTASM).
           - The structure is to be written to FRED.INCLUDE(TESTASM).
           - The required DSECT Utility options are EQU(BIT).

  USE THE FOLLOWING COMMAND:
           CDSECT 'FRED.SOURCE(TESTASM)' 'FRED.INCLUDE(TESTASM)' EQU(BIT)
```

Figure 46. Running the DSECT Utility under TSO

When the CDSECT command is executed, the High Level Assembler is executed with the required options. The DSECT utility is then executed with the specified options. A report of the options and any error messages will be displayed on the terminal.

If the assembler source requires macros or copy members from a macro library, issue the ALLOCATE command to allocate the required macro libraries to the SYSLIB DD statement before issuing the CDSECT command.

Chapter 16. Coded character set and locale utilities

This information describes the coded character set conversion utilities and the localedef utility. The coded character set conversion utilities help you to convert a file from one coded character set to another. The localedef utility allows you to define the language and cultural conventions that your environment uses.

Coded character set conversion utilities

These are the coded character set conversion utilities that you may find useful:

iconv Converts a file from one coded character set encoding to another. You can use iconv to convert C source code before compilation or to convert input files. For more information, refer to *z/OS UNIX System Services Command Reference*.

uconvdef
Reads the input source file and creates a binary conversion table. The input source file defines a mapping between UCS-2 and multibyte code sets. For more information, refer to *z/OS UNIX System Services Command Reference*.

genxlt Generates a translate table that the iconv utility and the iconv family of functions can use to convert coded character sets. It can be used to build code set converters for code pages that are not supplied with z/OS XL C/C++, or to build code set conversions for existing code pages.

| Custom conversion tables generated by the genxlt or uconvdef utilities, like
| those shipped in the National Language Resources component of z/OS
| Language Environment, are intended for use with the C/C++ iconv
| interfaces or the iconv utility. Direct programming to these tables is not
| supported and will produce unpredictable results. IBM makes no guarantee
| that converter binaries shipped with z/OS Language Environment will
| continue to be shipped in future releases. For more information, refer to
| *z/OS XL C/C++ Programming Guide*.

The genxlt utility runs under z/OS batch and TSO. The iconv utility runs under z/OS batch, TSO, and the z/OS shell. The `iconv_open()`, `iconv()`, and `iconv_close()` functions can be called from applications running under these environments and IBM CICS/ESA®.

iconv utility

The iconv utility converts the characters from the input file from one coded character set (code set) definition to another code set definition, and writes the characters to the output file.

The iconv utility creates one character in the output file for each character in the input file, and does not perform padding or truncation.

When conversions are performed between single-byte code pages, the output files are the same length as the input files. When conversions are performed between double-byte code pages, the output files may be longer or shorter than the input files because the shift-out and shift-in characters may be added or removed. If you are using the iconv utility under the z/OS shell, see *z/OS UNIX System Services Command Reference* for details on syntax and uses.

© Copyright IBM Corp. 1998, 2009

There are three standard library functions that can be used by any application to change the character set of data. These functions are `iconv_open()`, `iconv()`, and `iconv_close()`. For more information on the iconv utility, see *z/OS XL C/C++ Programming Guide*.

Under z/OS batch

JCL procedure EDCICONV invokes the iconv utility to copy the input data set to the output data set and convert the characters from the input code page to the output code page.

The EDCICONV procedure has the following parameters:

INFILE	The data set name for the input data set
OUTFILE	The data set name for the output data set
FROMC	The name of the code set in which the input data is encoded
TOC	The name of the code set to which the output data is to be converted

Example:

```
//ICONV    EXEC PROC=EDCICONV,
//         INFILE='FRED.INFILE',
//         OUTFILE='FRED.OUTFILE',
//         FROMC='IBM-037',
//         TOC='IBM-1047'
```

| Records from the input data set are processed by the iconv utility sequentially, one record at a time. If DBCS or multibyte character codes span a record boundary, the iconv utility will not recognize the partial sequence as a valid character and will fail. Therefore, input records must terminate with a complete character sequence. Be sure that the record length of the input data set is large enough to contain the longest input record.

The output data set must be preallocated. If the data set does not exist, iconv will fail. An output data set with a fixed record format may only be used if all the records created by the iconv utility will have the same record length as the output data set. No padding or truncation is performed. If the output data set has variable length records, the record length must be large enough for the longest record created. Because of these restrictions, when converting to or from a DBCS, the output data set must have variable length records. Otherwise, the iconv utility will fail.

For more information on the iconv utility, refer to *z/OS XL C/C++ Programming Guide*.

Under TSO

TSO CLIST ICONV invokes the iconv utility to copy the input data set to the output data set and convert the characters from the input code page to the output code page.

The parameters of the ICONV CLIST are as follows:

```
►►──ICONV──infile──outfile──FROMCODE(──fromcode──)──TOCODE(──tocode──)──────────►◄
```

Where:

infile	The input data set name.
outfile	The output data set name.

fromcode	The name of the code set in which the input data is encoded.
tocode	The name of the code set to which the output data is to be converted.

Example:

```
ICONV INPUT.FILE OUTPUT.FILE FROMCODE(IBM-037) TOCODE(IBM-1047)
```

Records from the input data set are processed by the iconv utility sequentially, one record at a time. If DBCS or multibyte character codes span a record boundary, the iconv utility will not recognize the partial sequence as a valid character and will fail. Therefore, input records must terminate with a complete character sequence. Be sure that the record length of the input data set is large enough to contain the longest input record.

The output data set must be pre-allocated. If the data set does not exist, iconv will fail. An output data set with a fixed record format may only be used if all the records created by the iconv utility will have the same record length as the output data set. No padding or truncation is performed. If the output data set has variable length records, the record length must be large enough for the longest record created. Because of these restrictions, when converting to or from a DBCS, the output data set must have variable length records. Otherwise the iconv utility will fail.

For more information on iconv, refer to *z/OS XL C/C++ Programming Guide*.

Under the z/OS Shell

Under z/OS UNIX System Services, use the **iconv** command to invoke the iconv utility. The following is the invocation syntax for the **iconv** command:

iconv [–sc] –f *oldset* **–t** *newset [file ...]*

or

iconv –l [–v]

The iconv utility converts characters in *file* (or from `stdin` if you do not specify a *file*) from one code page set to another. It writes the converted text to `stdout`. See *z/OS XL C/C++ Programming Guide* for more information about the code sets that are supported for this command.

If the input contains a character that is not valid in the source code set, iconv replaces it with the byte `0xff` and continues, unless the **–c** option is specified.

If the input contains a character that is not valid in the destination code set, the behavior depends on the `iconv()` function of the system. See *z/OS XL C/C++ Run-Time Library Reference* for more information about the character that is used for converting incorrect characters.

You can use iconv to convert single-byte data or double-byte data.

Options:

–c	Characters that contain conversion errors are not written to the output. By default, characters not in the source character set are converted to the value `0xff` and written to the output.

–f *oldset*	*oldset* can be either the code set name or a pathname to a file that contains an external code set. Specifies the current code set of the input.
–l	Lists code sets in the internal table. This option is not supported.
–s	Suppresses all error messages about faulty encodings.
–t *newset*	Specifies the destination code set for the output. *newset* can be either the code set name or a pathname to a file that contains an external code set.
–v	Specifies verbose output.

genxlt utility

The genxlt utility creates translation tables, which are used by the `iconv_open()`, `iconv()`, and `iconv_close()` services of the run-time library. These services can be called from both non-XPLINK and XPLINK applications. The non-XPLINK and XPLINK versions have different names. The non-XPLINK and XPLINK versions of the GENXLT table should always be generated. If any XPLINK applications will require one of these translation tables, then the XPLINK version should also be generated.

Under TSO, you specify the options on the command line. Under z/OS batch, the options are specified on the EXEC PARM, and may be separated by spaces or commas. If you specify the same option more than once, genxlt uses the last specification.

DBCS|NODBCS
Specifies whether genxlt will convert the DBCS characters within shift-out and shift-in characters. You should only specify the DBCS option when you are converting an EBCDIC code page to a different EBCDIC code page.

If the DBCS option is specified, when a shift-out character is encountered in the input, the characters up to the shift-in character are copied to the output, and not converted. There must be an even number of characters between the shift-out and shift-in characters, and the characters must be valid DBCS characters.

If you specify the NODBCS option, genxlt treats all the characters as a single SBCS character, and does not perform a check of DBCS characters.

For more information on the genxlt utility, refer to *z/OS XL C/C++ Programming Guide*.

Under z/OS batch

JCL procedure EDCGNXLT invokes the genxlt utility to read the character conversion information and produce the conversion table. It invokes the system Linkage Editor to build the load module.

The EDCGNXLT procedure has the following parameters:

`INFILE`	The data set name for the file that contains the character conversion information.
`OUTFILE`	The data set name for the output file that is to contain the link-edited conversion table. The non-XPLINK version of this table should have EDCU as the first four characters. The XPLINK version of this table should have CEHU as the first four characters.

GOPT Options for the genxlt utility.

Example:

```
//GENXLT    EXEC PROC=GENXLT,
//          INFILE='FRED.GENXLT.SOURCE(EDCUEAEY)',
//          OUTFILE='FRED.GENXLT.LOADLIB(EDCUEAEY)',
//          GOPT='DBCS'
```

Under TSO

TSO CLIST GENXLT invokes the genxlt utility to read the character conversion information and produce the conversion table. It then invokes the system Linkage Editor to build the load module.

The general parameters for GENXLT CLIST are as follows:

```
►►──GENXLT──infile──outfile──┬──────────┬──────────────────────────────────────►◄
                             ├─DBCS─────┤
                             └─NODBCS───┘
```

Where:

infile The file name for the file that contains the character conversion information.

outfile The file name for the output file that is to contain the link-edited conversion table. The non-XPLINK version of the table should have EDCU as the first four characters. The XPLINK version of this table should have CEHU as the first four characters.

For example:

```
GENXLT GENXLT.SOURCE(EDCUEAEY) GENXLT.LOADLIB(EDCUEAEY) DBCS
```

localedef utility

A *locale* is a collection of data that defines language and cultural conventions. Locales consist of various categories, that are identified by name, that characterize specific aspects of your cultural environment.

The localedef utility generates locales according to the rules that are defined in the locale definition file. You can create your own customized locale definition file.

The localedef utility creates locale objects, which are used by the setlocale() service of the run-time library. This service can be called from both non-XPLINK and XPLINK applications. The non-XPLINK, XPLINK, and 64-bit locale object versions have different names. Also, localedef can generate the locale objects into a PDS or PDSE under BATCH or TSO, or into the z/OS UNIX System Services file system under the z/OS shell. The non-XPLINK, XPLINK, and 64-bit versions of the locale object should always be generated.

The utility reads the locale definition file and produces a locale object that the locale-specific library functions can use. You invoke localedef using either a JCL procedure or a TSO CLIST, or by specifying the **localedef** command under z/OS UNIX System Services. To activate a locale during your application's execution, you call the run-time function `setlocale()`.

Note: TSO and z/OS batch are not supported for building 64-bit locales. You must use the **localedef** command under z/OS UNIX System Services to build 64-bit locales.

The options for the localedef utility in TSO or z/OS batch are as follows. Spaces or commas can separate the options. If you specify the same option more than once, localedef uses the last option that you specified.

CHARMAP(*name*)	Specifies the member name of the file that contains the definition of the encoded character set. If you do not specify this option, the localedef utility assumes the encoded character set IBM-1047. The name that is specified for the CHARMAP is the member name within a partitioned data set, with the – (dash) sign converted to an @ (at) sign.
FLAG(W|E)	The FLAG option controls whether localedef issues warning messages. If you specify FLAG(W), localedef issues warning and error messages. If you specify FLAG(E), localedef issues only the error messages.
BLDERR|NOBLDERR	If you specify the BLDERR option, localedef generates the locale even if it detects errors. If you specify the NOBLDERR option, localedef does not generate the locale if it detects an error.

The following topics describe how you can invoke the localedef utility. For more information on locale source definition files, codeset mapping files (CHARMAPs), method files, and locale object names, refer to *z/OS XL C/C++ Programming Guide*. For information on using the localedef utility under z/OS UNIX System Services, refer to *z/OS UNIX System Services Command Reference*.

Under z/OS batch

Note: To build XPLINK optimized locales, use EDCXLDEF.

Under z/OS batch, JCL procedure EDCLDEF invokes the localedef utility. It does the following:

1. Invokes the EDCLDEF module to read the locale definition data set and produces the C code to build the locale
2. Invokes the z/OS XL C compiler to compile the C source generated
3. Invokes the Linkage Editor to build the locale into a loadable module

The EDCLDEF JCL procedure has the following parameters:

INFILE	The data set name for the file that contains the locale definition information.
OUTFILE	For non-XPLINK, it is the data set name for the output partitioned data set and member that is to contain the link-edited locale object. For XPLINK, it is the data set name for the output PDSE and member that is to contain the bound locale object. The non-XPLINK version of the locale object should have EDC$ or EDC@ as the first four characters of the member name. The name that is chosen determines the locale that is built (for further information, see *z/OS XL C/C++ Programming Guide*). The XPLINK version should have CEH$ or CEH@ as the first four characters of the member name.
LOPT	The options for the localedef utility

Example:

```
//LOCALDEF    EXEC PROC=EDCLDEF,
//            INFILE='FRED.LOCALE.SOURCE(EDC$EUEY)',
//            OUTFILE='FRED.LOCALE.LOADLIB(EDC$EUEM)',
//            LOPT='CHARMAP(IBM-297)'
```

Under z/OS batch, you specify the options on the EXEC PARM and separate them by spaces or commas.

Under TSO

Under TSO, LOCALDEF invokes the localedef utility. The name is shortened to 8 characters from LOCALEDEF because of the file naming restrictions. It does the following:

1. Invokes the EDCLDEF module to read the locale definition data set and produce the C code to build the locale
2. Invokes the z/OS XL C compiler to compile the C source
3. Invokes the Linkage Editor to build the locale into a loadable module

The invocation syntax for the LOCALDEF REXX EXEC is as follows:

```
►►──LOCALDEF──infile──outfile──┬──────────────────┬──┬────────┬──────────────►◄
                               └─LOPT(──loptions)─┘  └─XPLINK─┘
```

where:

infile	The data set name for the data set that contains the locale definition information
outfile	For non-XPLINK, it is the data set name for the output partitioned data set and member that is to contain the link-edited locale object. For XPLINK, it is the data set name for the output PDSE and member that is to contain the bound locale object. The non-XPLINK version of the locale object should have EDC$ or EDC@ as the first four characters of the member name. The XPLINK version should have CEH$ or CEH@ as the first four characters of the member name.
loptions	The options for the localedef utility.
XPLINK	Indicates that the locale to be built is an XPLINK locale.

Example: In the following example, the input source is `LOCALE.SOURCE(EDC$EUEY)`, the output library is `LOCALE.LOADLIB(EDC$EUEM)` for `en_us.IBM-297`, and the options are `CHARMAP(IBM-297)`:

```
LOCALEDEF LOCALE.SOURCE(EDC$EUEY) LOCALE.LOADLIB(EDC$EUEM) LOPT(CHARMAP(IBM-297))
```

Under TSO, you specify the options on the command line.

Under the z/OS Shell

Under z/OS UNIX System Services, use the **localedef** command to invoke the localedef utility. The following is the invocation syntax for the **localedef** command:

localedef [**–c**] [**–w**] [**–X**] [**–A**][**–f** *charmap*] [**–i** *sourcefile*] [**–m**] [**–L** *binderoptions*] *name*

Options:

–A	Causes localedef to generate an ASCII locale object. ASCII locales invoke ASCII methods, so they must be generated using ASCII

charmaps. An ASCII *charmap* maps symbolic character names into ASCII code points, but even ASCII *charmap* specifications are written in EBCDIC code page IBM-1047. Users must ensure that the *charmap* specified, when they invoke the localedef utility, is an ASCII *charmap*. Note: When –A is specified, –X is assumed because ASCII locales are only supported as XPLINK locales.

–c
: Creates permanent output even if there were warning messages. Normally, localedef does not create permanent output when it has issued warning messages.

–f *charmap*
: Specifies a *charmap* file that contains a mapping of character symbols and collating element symbols to actual character encodings.

–i *sourcefile*
: Specifies the file that contains the source definitions. If there is no **–i**, localedef reads the source definitions from the standard input.

–m *MethodFile*
: Specifies the names of a method file that identifies the methods to be overridden when constructing a locale object. The localedef utility reads a method file and uses indicated entry points when constructing a locale object. Method files are used to replace IBM-supplied method functions with user-written method functions. For each replaced method, the *method file* supplies the user-written method function name and optionally indicates where the method function code is to be found (.o file, archive library or DLL). Method files typically replace the *charmap* related methods. When this is done, the end result is the creation of a locale, which supports a blended code page. The user-written method functions are used both by the locale-sensitive APIs they represent, and also by localedef itself while generating the method-file based ASCII locale object. This second use by localedef itself causes a temporary DLL to be created, while processing the *charmap* file supplied on the **–f** parameter. The name of the file containing method objects or side deck information is passed by localedef as a parameter on the c89 command line, so the standard archive/object/side deck suffix naming conventions apply (in other words, `.a`, `.o`, `.x`).

Note: Method files may only be used when constructing ASCII locale objects (that is, when the –A option is also specified). If the –A option is not specified along with the **–m** option, then a severe error message will be issued and processing will be terminated.

–w
: Instructs localedef to issue a warning message when a duplicate character definition is found. This is mainly intended for debugging character map specifications. It can help to ensure that a code point value is not accidentally assigned to the wrong symbolic character name.

–X
: Causes localedef to generate an XPLINK AMODE 31 locale object (DLL).

–L *binderoptions*
: Instructs localedef to pass additional binder options (mostly for diagnostic purposes).

-6
: Instructs localedef to generate an XPLINK AMODE 64 locale object (DLL). The -X option is implied when this option is specified.

name	Is the name applied to the target locale object generated by localedef. Locale naming conventions are described in detail in *z/OS XL C/C++ Programming Guide*. If the naming conventions are not followed, applications are required to supply the full path name on each `setlocale()` invocation of locales that reside in the z/OS UNIX file system.

z/OS ships two versions of the localedef utility:

- One can be invoked under z/OS batch and TSO, and is shipped in the CEE.SCEERUN2 data set.
- The other can be invoked under z/OS UNIX System Services, and is shipped with z/OS UNIX.

The TSO REXX Exec localedef, included in the z/OS XL C/C++ compiler, is not supported in the z/OS shell environment. In that environment, use the z/OS UNIX System Services **localedef** command instead. For more information on the **localedef** command, refer to *z/OS UNIX System Services Command Reference*.

Chapter 17. CDAHLASM — Use the HLASM assembler to create DWARF debug information (C only)

Description

The CDAHLASM utility is the MVS batch equivalent of the as utility. This utility is shipped as part of the Run-Time Library Extensions and is installed in CEE.SCEERUN2.

When the METAL compiler option is specified, the compiler generates output in the form of assembler source. The XL C compiler cannot generate DWARF information directly because it cannot create symbolic debugging information. The symbolic debugging information can be obtained only during object code generation, in this case, during the assembly stage.

Debuggers can use the DWARF-formatted output from the CDAHLASM utility to debug Metal C applications. To enable the generation of complete DWARF information, the compiler embeds the type information, created during the compilation stage, into the generated assembler source output. The assembly stage takes the embedded information, and combines it with the symbolic debugging information obtained during assembling, and produces the final DWARF information side file.

The CDAHLASM utility also produces debug information in ADATA format, which is required for the generation of DWARF information. The ADATA assembler option will be passed to the assembler unless the NODEBUG option is passed to CDAHLASM. For further information on the NODEBUG option, see the CDAHLASM "Options" topic.

For information on the CDAASMC cataloged procedure, which executes the CDAHLASM utility, see Chapter 12, “Cataloged procedures and REXX EXECs,” on page 465.

Options directed to the CDAHLASM utility can be specified only through the DD:CDAHOPT.

Options

PHASEID
: Displays the version of CDAHLASM as well as the Common Debug Architecture run-time `phaseid` information.

NODEBUG
: Suppresses the generation of DWARF debug information.

VERBOSE
: Specifies verbose mode, which writes additional informational messages to DD:SYSOUT.

Chapter 18. Archive and make utilities

This information describes the z/OS UNIX System Services archive (ar) and make utilities. There are several other useful z/OS UNIX System Services utilities such as gencat and mkcatdefs. For information on their syntax and use, refer to *z/OS UNIX System Services Command Reference*.

The z/OS Shell and Utilities provide two utilities that you can use to simplify the task of creating and managing z/OS UNIX System Services XL C/C++ application programs: ar and make. Use these utilities with the c89 and c++ utilities to build application programs into easily updated and maintained executable files.

Archive libraries

The ar utility allows you to create and maintain a library of z/OS XL C/C++ application object files. You can specify the **c89** and **c++** command strings so that archive libraries are processed during the IPA Link step or binding.

The archive library file, when created for application program object files, has a special symbol table for members that are object files. The symbol table is read to determine which object files should be bound into the application program executable file. The binder processes archive libraries during the binding process. It includes any object file in the specified archive library that it can use to resolve external symbols. Use of this autocall library mechanism is analogous to the use of Object Libraries with object files in data sets. For more information, see Chapter 13, “Object library utility,” on page 481.

By default, the c89 and c++ utilities require that archive libraries end in the suffix `.a`, as in `file.a`. For example; source file `dirsum.c` is in your `src` subdirectory in your working directory, and the archive library `symb.a` is in your working directory. To compile `dirsum.c` and resolve external symbols from `symb.a`, and create the executable in `exfils/dirsum` enter:

```
c89 -o exfils/dirsum src/dirsum.c symb.a
```

Creating archive libraries

To create the archive library, use the **ar -r** option.

Example: To create an archive library that is named `bin/libbrobompgm.a` from your working directory, and add the member `jkeyadd.o` to it, specify:

```
ar -rc ./bin/libbrobompgm.a jkeyadd.o
```

ar creates the archive library file `libbrobompgm.a` in the `bin` subdirectory of your z/OS UNIX System services working directory. The **-c** option tells **ar** to suppress the message that it normally sends when it creates an archive library file.

Example: For control purposes, and when working interactively, you can use the **-v** option to generate a message as each member is added to the archive:

```
ar -rv ./bin/libbrobompgm.a jkeyadd.o
```

Example: To display the object files that are archived in the `bin/libbrobompgm.a` library from your working directory, specify:

```
ar -t ./bin/libbrobompgm.a
```

For a detailed discussion of the ar utility, see *z/OS UNIX System Services Command Reference*.

Creating makefiles

The make utility maintains all the parts of and dependencies for your application program. It uses a makefile, to keep your application parts (listed in it) up to date with one another. If one part changes, make updates all the other files that depend on the changed part.

A *makefile* is a z/OS UNIX System Services text file. You can use any text editor to create and edit the file. It describes the application program files, their locations, dependencies on other files, and rules for building the files into an executable file. When creating a makefile, remember that tabbing of information in the file is important and not all editors support tab characters the same way.

The make utility uses c89 or c++ to call the z/OS XL C/C++ compiler, and the binder, to recompile and rebind an updated application program.

See *z/OS UNIX System Services Programming Tools* and *z/OS UNIX System Services Command Reference* for a detailed discussion of the shell make utility.

Makedepend utility

The makedepend utility can also be used to create a makefile. The makedepend utility is used to analyze each source file to determine what dependency it has on other files. This information is then placed into a usable makefile. See *z/OS UNIX System Services Command Reference* for a detailed discussion of the makedepend utility.

Note: The stand-alone makedepend utility is no longer being enhanced in future release. Instead, this utility is superseded in favour of the xlc option **-qmakedep**. For more information of the new **-qmakedep** compiler option, see Chapter 4, “Compiler Options,” on page 41 and Chapter 24, “xlc — Compiler invocation using a customizable configuration file,” on page 577.

Chapter 19. BPXBATCH utility

This information provides a quick reference for the IBM-supplied BPXBATCH program. BPXBATCH makes it easy for you to run shell scripts and z/OS XL C/C++ executable files that reside in z/OS UNIX System Services files through the z/OS batch environment. If you do most of your work from TSO/E, use BPXBATCH to avoid going into the shell to run your scripts and applications.

In addition to using BPXBATCH, if you want to perform a local spawn without being concerned about environment set-up (that is, without having to set specific environment variables, which could be overwritten if they are also set in your profile) you can use BPXBATSL. BPXBATSL, which provide you with an alternate entry point into BPXBATCH, and force a program to run using a local spawn instead of fork or exec as BPXBATCH does. This ultimately allows a program to run faster.

BPXBATSL is also useful when you want to perform a local spawn of your program, but also need subsequent child processes to be forked or executed. Formerly, with BPXBATCH, this could not be done since BPXBATCH and the requested program shared the same environment variables. BPXBATSL is provided as an alternative to BPXBATCH. It will force the running of the target program into the same address space as the job itself is initiated in, so that all resources for the job can be used by the target program; for example, DD allocations. In all other respects, it is identical to BPXBATCH.

For information on c89 commands, see Chapter 21, “c89 — Compiler invocation using host environment variables,” on page 535.

BPXBATCH usage

The BPXBATCH program allows you to submit z/OS batch jobs that run shell commands, scripts, or z/OS XL C/C++ executable files in z/OS UNIX System Services files from a shell session. You can invoke BPXBATCH from a JCL job, from TSO/E (as a command, through a CALL command, from a REXX EXEC).

JCL: Use one of the following:

- `EXEC PGM=BPXBATCH,PARM='SH program-name'`
- `EXEC PGM=BPXBATCH,PARM='PGM program-name'`

TSO/E: Use one of the following:

- `BPXBATCH SH program-name`
- `BPXBATCH PGM program-name`

BPXBATCH allows you to allocate the z/OS standard files `stdin`, `stdout`, and `stderr` as z/OS UNIX files for passing input, for shell command processing, and writing output and error messages. If you do allocate standard files, they must be z/OS UNIX files. If you do not allocate them, `stdin`, `stdout`, and `stderr` default to `/dev/null`. You allocate the standard files by using the options of the data definition keyword `PATH`.

Note: The BPXBATCH utility also uses the STDENV file to allow you to pass environment variables to the program that is being invoked. This can be useful when not using the shell, such as when using the PGM parameter.

Example: For JCL jobs, specify PATH keyword options on DD statements; for example:

```
//jobname JOB ...

//stepname EXEC PGM=BPXBATCH,PARM='PGM program-name parm1 parm2'

//STDIN   DD  PATH='/stdin-file-pathname',PATHOPTS=(ORDONLY)
//STDOUT  DD  PATH='/stdout-file-pathname',PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//            PATHMODE=SIRWXU
//STDERR  DD  PATH='/stderr-file-pathname',PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//            PATHMODE=SIRWXU
⋮
```

You can also allocate the standard files dynamically through use of SVC 99.

For TSO/E, you specify PATH keyword options on the ALLOCATE command. For example:

```
ALLOCATE FILE(STDIN) PATH('/stdin-file-pathname') PATHOPTS(ORDONLY)
ALLOCATE FILE(STDOUT) PATH('/stdout-file-pathname')
         PATHOPTS(OWRONLY,OCREAT,OTRUNC) PATHMODE(SIRWXU)
ALLOCATE FILE(STDERR) PATH('/stderr-file-pathname')
         PATHOPTS(OWRONLY,OCREAT,OTRUNC) PATHMODE(SIRWXU)

BPXBATCH SH program-name
```

You must always allocate `stdin` as read. You must always allocate `stdout` and `stderr` as write.

Parameter

BPXBATCH accepts one parameter string as input. At least one blank character must separate the parts of the parameter string. When BPXBATCH is run from a batch job, the total length of the parameter string must not exceed 100 characters. When BPXBATCH is run from TSO, the parameter string can be up to 500 characters. If neither SH nor PGM is specified as part of the parameter string, BPXBATCH assumes that it must start the shell to run the shell script allocated by `stdin`.

SH | PGM
Specifies whether BPXBATCH is to run a shell script or command or a z/OS XL C/C++ executable file that is located in a z/OS UNIX System Services file.

SH
Instructs BPXBATCH to start the shell, and to run shell commands or scripts that are provided from `stdin` or the specified *program-name*.

Note: If you specify SH with no program-name information, BPXBATCH attempts to run anything read in from `stdin`.

PGM
Instructs BPXBATCH to run the specified *program-name* as a called program.

If you specify PGM, you must also specify *program-name*. BPXBATCH creates a process for the program to run in and then calls the program. The HOME and LOGNAME environment variables are set automatically when the program is run, only if they do not exist in the file that is referenced by STDENV. You can use STDENV to set these environment variables, and others.

program-name
Specifies the shell command or the z/OS UNIX path name for the shell script or z/OS XL C/C++ executable file to be run. In addition, *program-name* can contain option information.

BPXBATCH interprets the program name as case-sensitive.

Note: When PGM and *program-name* are specified and the specified program name does not begin with a slash character (/), BPXBATCH prefixes your *initial* working directory information to the program path name.

Usage notes

You should be aware of the following:

1. BPXBATCH is an alias for the program BPXMBATC, which resides in the SYS1.LINKLIB data set.
2. BPXBATCH must be invoked from a user address space running with a program status word (PSW) key of 8.
3. BPXBATCH does not perform any character translation on the supplied parameter information. You should supply parameter information, including z/OS UNIX path names, using only the POSIX portable character set.
4. A program that is run by BPXBATCH cannot use allocations for any files other than `stdin`, `stdout`, or `stderr`.
5. BPXBATCH does not close file descriptors except for `0`, `1`, and `2`. Other file descriptors that are open and not defined as “marked to be closed” remain open when you call BPXBATCH. BPXBATCH runs the specified script or executable file.
6. BPXBATCH uses write-to-operator (WTO) routing code 11 to write error messages to either the JCL job log or your TSO/E terminal. Your TSO/E user profile must specify WTPMSG so that BPXBATCH can display messages to the terminal.

Files

The following list describes the files:

- SYS1.LINKLIB(BPXMBATC) is the BPXBATCH program location.
- The `stdin` default is `/dev/null`.
- The `stdout` default is `/dev/null`.
- The `stdenv` default is `/dev/null`.
- The `stderr` default is the value of `stdout`. If all defaults are accepted, `stderr` is `/dev/null`.

Chapter 20. as — Use the HLASM assembler to produce object files

Format

as
[**--***option*[, *option*] ...] ...
[**-a**[**egimrsx**][=*file*]] ...
[**-g**]
[**--[no]gadata**[=*file*]]
[**--[no]gdwarf3**[=*file*]]
[**-m***option*]
[**-I** *name*]
[**-o** *objectfile*]
[**-v**]
[**--[no]help**]
[**--[no]verbose**]
file

Description

The **as** command processes assembler source files and invokes the HLASM assembler to produce object files.

Options

--
Accepts all options that are accepted by HLASM. Multiple options can be specified by separating them with a comma. This style of option specification is designed to provide smooth migration for users accustomed to specifying options in JCL. For example:

```
--"FLAG(ALIGN),RENT"
```

-a[egimrsx][=*file*]
Instructs the assembler to produce a listing.

-ae
Instructs the assembler to produce the External Symbol Dictionary section of the assembler listing. This is equivalent to specifying: --ESD.

-ag
Instructs the assembler to produce the General Purpose Register Cross Reference section of the assembler listing. This is equivalent to specifying: --RXREF.

-ai
Instructs the assembler to copy all product information to the list data set. This is equivalent to specifying: --INFO.

-am
Instructs the assembler to produce the Macro and Copy Code Source Summary section of the assembler listing. This is equivalent to specifying: --MXREF.

-ar
Instructs the assembler to produce the Relocation Dictionary (RLD) section of the assembler listing. This is equivalent to specifying: --RLD.

-as
Instructs the assembler to produce the Ordinary Symbol and Literal Cross Reference section of the assembler listing. It also instructs the assembler to produce the un-referenced symbols defined in the

CSECTs section of the assembler listing. This is equivalent to specifying: --XREF(SHORT,UNREFS).

-ax Instructs the assembler to produce the DSECT Cross Reference section of the assembler listing. This is equivalent to specifying: --DXREF.

=*file* Specifies the file name of the listing output. If you do not specify a file name, the output goes to `stdout`.

You may combine these options; for example, use **-ams** for an assembly listing with expanded macro and symbol output. The **=***file* option, if used, must be specified last.

-g Instructs the assembler to collect debug information. By default, the debug information is produced in DWARF Version 3 format (or **--gdwarf3**).

--[no]gadata[=*file***]**
Instructs the assembler to collect associated data and write it to the associated data file. You can optionally specify the name of the output debug file. The specified name cannot be a PDS or z/OS UNIX file system directory name. If you do not specify a file name, the default name is created as follows:

- If you are compiling a data set, the **as** command uses the source file name to form the name of the output data set. The high-level qualifier is replaced with the user ID under which the **as** command is running, and .ADATA is appended as the low-level qualifier. For example, if `TS12345` is compiling `TSMYID.MYSOURCE(src)` with this option, the produced debug file name will be `TS12345.MYSOURCE.ADATA(src)`.
- If you are compiling a z/OS UNIX file, the **as** command stores the debug information in a file that has the name of the source file with an .ad extension. For example, if you are compiling `src.a` with this option, the compiler will create a debug file named `src.ad`.

--[no]gdwarf3[=*file***]**
Instructs the assembler to generate debug information conforming to the DWARF Version 3 format. Debugging tools (for example, dbx) can take advantage of this debug information. You can optionally specify the name of the output debug file. The file name of the output debug file must be a PDS member, a sequential data set or z/OS UNIX System Services file; it cannot be a PDS directory or z/OS UNIX System Services file system directory name. If you do not specify a file name, the default name is created as follows:

- If you are compiling a data set, the **as** command uses the source file name to form the name of the output data set. The high-level qualifier is replaced with the userid under which the **as** command is running, and .DBG is appended as the low-level qualifier. For example, if `TS12345` is compiling `TSMYID.MYSOURCE(src)` with the **-g** option, the produced debug file name will be `TS12345.MYSOURCE.DBG(src)`. If `TS12345` is compiling `TSMYID.SEQSRC` with the **-g** option, the produced debug file name will be `TS12345.SEQSRC.DBG`.
- If you are compiling a z/OS UNIX System Services file, the **as** command stores the debug information in a file that has the name of the source file with a .dbg extension. For example, if you are compiling `src.a` with the **-g** option, the produced debug file name will be `src.dbg`.

-m*option*
HLASM keyword options are specified using the following syntax:

```
-m<option>[=<parm>[=<value>][:<parm>[=<value>]]...]
```

where <option> is an option name, <parm> is a suboption name, and <value> is the suboption value.

Keyword options with no parameters represent switches that may be either on or off. The keyword by itself turns the switch on, and the keyword preceded by the letters NO turns the switch off. For example, **-mLIST** tells the HLASM assembler to produce a listing and **-mNOLIST** tells the HLASM assembler not to produce a listing. If an option that represents a switch is set more than once, the HLASM assembler uses the last setting.

Keyword option and parameter names may appear in mixed case letters in the invocation command.

-I *name*
: Instructs HLASM to look for assembler macro invocation in the specified location. The *name* can be either a PDS name or z/OS UNIX file system directory name. If a PDS data set is specified, it must be fully qualified. The specified locations are then prepended to a default set of macro libraries. The **as** command assumes a default set of macro libraries that is compatible with the defaults for the C/C++ compilers. The default data sets used are: -I CEE.SCEEMAC, -I SYS1.MACLIB, and -I SYS1.MODGEN. The default data sets can be changed via the environment variable _AS_MACLIB, for example:

```
export _AS_MACLIB="FIRST.PDS:SECOND.PDS"
```

-o *objectfile*
: Specifies the name of the object file. If the name specified is a PDS or z/OS UNIX System Services directory name, a default file name is created in the PDS or z/OS UNIX directory specified as follows:

- If the source file is a sequential data set, the second last part of the data set name will be used. If the data set name only contains one part after the high-level qualifier, then the last part will be used.
- If the source file is a PDS member, the member name will be used.
- If the source file is a z/OS UNIX file, the suffix will be removed if applicable.
- If the object file is going into a PDS, the first eight characters of the name will be used. If there is a dot, anything after the first dot will be removed.
- If the object file is going into a z/OS UNIX directory, .o will be appended to the name.

For example:

```
Source file: //'abc.hello.source'
Ouput file in PDS: HELLO
Output file in UNIX directory: hello.o

Source file: //'ABC.HELLO'
Ouput file in PDS: HELLO
Output file in UNIX directory: HELLO.o

Source file: //SOURCE(hello)
Ouput file in PDS: HELLO
Output file in UNIX directory: hello.o

Source file: /abc/hello.s
Ouput file in PDS: HELLO
Output file in UNIX directory: hello.o
```

```
Source file: /abc/hellothere.s
Ouput file in PDS: HELLOTHE
Output file in UNIX directory: hellothere.o
```

-v Writes the version of the **as** command to `stderr`.

--[no]help
Help menu. Displays the syntax of the **as** command.

--[no]verbose
Specifies verbose mode, which writes additional information messages to `stdout`.

file may be:
- An MVS data set (for example, //somename)
- An absolute z/OS UNIX file (for example, /somename)
- A relative z/OS UNIX file (for example, ./somename or somename)

The output of the **as** command is an object file. If you do not specify a file name via the **-o** option, the default name is created as follows:
- If you are compiling a data set, the **as** command uses the source file name to form the name of the output data set. The high-level qualifier is replaced with the user ID under which the **as** command is running, and .OBJ is appended as the low-level qualifier. For example, if `TS12345` is compiling `TSMYID.MYSOURCE(src)`, the compiler will create an object file named `TS12345.MYSOURCE.OBJ(src)`.
- If you are compiling a z/OS UNIX file, the **as** command names the object file with the name of the source file with an .o extension. For example, if you are compiling `src.a`, the object file name will be `src.o`.

Notes:
1. The **as** command does not accept standard input as a file.
2. The **as** command invokes the HLASM assembler to produce the object file. The HLASM assembler is invoked with the default options ASA and TERM. The ASA option instructs HLASM to use American National Standard printer control characters in records written to the listing file, thus making the listing file more readable in the z/OS UNIX System Services environment. The TERM option instructs HLASM to write error messages to `stderr`. These defaults can be changed by using the **-m** option or **--** option.
3. HLASM messages and **as** error messages are directed to `stderr`. Verbose option output is directed to `stdout`.
4. When invoking **as** from the shell, any option arguments or operands specified that contain characters with special meaning to the shell must be escaped. For example, source files specified as PDS member names contain parentheses; if they are specified as fully qualified names, they contain single quotation marks. To escape these special characters, either enclose the option argument or operand in double quotation marks, or precede each character with a backslash.

Chapter 21. c89 — Compiler invocation using host environment variables

Format

c89 | **cc** | **c++** | **cxx** [**–+CcEFfgOpqrsVv0123**]
[**-D** *name*[=*value*]]... [**-U** *name*]...
[**-e** *function*] [**-u** *function*]...
[**-W** *phase,option*[*,option*]...]...
[**-o** *outfile*]
[**-I** *directory*]... [**-L** *directory*]...
[*file.C*]... [*file.i*]... [*file.c*]... [*file.s*]...
[*file.o*]... [*file.x*]... [*file.p*]... [*file.I*]... [*file.a*]... [**-l** *libname*]...

Notes:

1. The **c99** command is only supported by the xlc utility. See Chapter 24, "xlc — Compiler invocation using a customizable configuration file," on page 577 for more information.
2. In this information, **-l** signifies `-l` (a lowercase L) and not an uppercase `I`.

Description

c89 and **cc** compile, assemble, and link-edit C programs; **cxx**/**c++** does the same for C++ programs.

- **c89** should be used when compiling C programs that are written according to *Standard C*.
- **cc** should be used when compiling C programs that are written according to *Common Usage C*.
- **cxx/c++** must be used when compiling C++ programs. Prior to z/OS V1R2, the C++ compiler supported the *Draft Proposal International Standard for Information Systems — Programming Language C++ (X3J16)*. As of z/OS V1R7, the C++ compiler supports the *Programming languages - C++ (ISO/IEC 14882:2003(E))* standard, as well as the *Programming languages - C++ (ISO/IEC 14882:1998)* standard. **c++** can compile both C++ and C programs, and can also be invoked by the name **cxx** (all references to **c++** throughout this document apply to both names).

c89, **cc**, and **c++** call other programs for each step of the compilation, assemble and link-editing phases. The list below contains the following: the step name, the name of the document that describes the program you use for that step and the document that describes any messages issued by that program, and prefixes to those messages:

Table 48. Reference documentation for programs invoked by c89, cc, and c++ commands

Step Name	Document Describing Options and How to Call Program	Document Containing Messages Issued by Program	Prefix of Messages Issued by Program
ASSEMBLE	*HLASM Programmer's Guide*	*HLASM Programmer's Guide*	ASMA

Table 48. Reference documentation for programs invoked by c89, cc, and c++ commands (continued)

Step Name	**Document Describing Options and How to Call Program**	**Document Containing Messages Issued by Program**	**Prefix of Messages Issued by Program**
COMPILE, IPACOMP, TEMPINC, IPATEMP, IPALINK	*z/OS C/C++ User's Guide* for releases prior to z/OS V1R7 and *z/OS XL C/C++ User's Guide* for z/OS V1R7 and later releases	*z/OS C/C++ Messages* for z/OS V1R5 and z/OS V1R6 releases and *z/OS XL C/C++ Messages* for z/OS V1R7 and later releases	CCN for z/OS V1R2 and later releases
PRELINK	*z/OS Language Environment Programming Guide* and *z/OS XL C/C++ User's Guide*	*z/OS Language Environment Debugging Guide*	EDC
LINKEDIT (Program Management Binder)	*z/OS MVS Program Management: User's Guide and Reference*	*z/OS MVS System Messages, Vol 8 (IEF-IGD)*	IEW

Execution of any Language Environment program (including c89 and the z/OS XL C/C++ compiler) can result in run-time messages. These messages are described in *z/OS Language Environment Run-Time Messages* and have an EDC prefix. In some cases, c89 issues messages with Language Environment messages appended to them. Messages issued by c89 have an FSUM3 prefix.

In order for **c89**, **cc**, and **c++** to perform C and C++ compiles, the z/OS C/C++ Optional Feature must be installed on the system. The z/OS C/C++ Optional Feature provides a C compiler, a C++ compiler, C++ Class Libraries, and some utilities. See *z/OS Introduction and Release Guide* for further details. Also see *prefix*_CLIB_PREFIX and *prefix*_PLIB_PREFIX in “Environment variables” on page 550 for information about the names of the z/OS XL C/C++ Optional Feature data sets that must be made available to **c89/cc/c++**.

Note: The term *prefix* is defined in “Environment variables” on page 550.

First, **c89**, **cc**, and **c++** perform the compilation phase (including preprocessing) by compiling all source file operands (*file.C*, *file.i*, and *file.c*, as appropriate). For **c++**, if automatic template generation is being used (which is the default), then z/OS XL C++ source files may be created or updated in the `tempinc` subdirectory of the working directory during the compilation phase (the `tempinc` subdirectory will be created if it does not already exist). Then, **c89**, **cc**, and **c++** perform the assemble phase by assembling all operands of the *file.s* form. The result of each compile step and each assemble step is a *file.o* file. If all compilations and assemblies are successful, or if only *file.o* and/or *file.a* files are specified, **c89**, **cc**, and **c++** proceed to the link-editing phase. For **c++**, the link-editing phase begins with an automatic template generation step when applicable. For IPA (Interprocedural Analysis) optimization an additional IPA Link step comes next. The link-edit step is last. See the environment variable *prefix*_STEPS under “Environment variables” on page 550 for more information about the link-editing phase steps.

In the link-editing phase, **c89**, **cc**, and **c++** combine all *file.o* files from the compilation phase along with any *file.o* files that were specified on the command line. For **c++**, this is preceded by compiling all C++ source files in the `tempinc` subdirectory of the working directory (possibly creating and updating additional C++ source files during the automatic template generation step). After compiling all the C++ source files, the resulting object files are combined along with the *file.o* files from the compilation phase and the command line. Any *file.a* files, *file.x* files and **-l** *libname* operands that were specified are also used.

The usual output of the link-editing phase is an executable file. For **c89**, **cc**, and **c++** to produce an executable file, you must specify at least one operand which is of other than **-l** *libname* form. If **-r** is used, the output file is not executable.

For more information about automatic template generation, see the information on ″Using TEMPINC or NOTEMPINC″ in *z/OS XL C/C++ Programming Guide*. Note that the **c++** command only supports using the `tempinc` subdirectory of the working directory for automatic template generation.

For more information on IPA, see “Options.”

Options

–+
Specifies that all source files are to be recognized as C++ source files. All *file.s*, *file.o*, and *file.a* files will continue to be recognized as assembler source, object, and archive files respectively. However, any C *file.c* or *file.i* files will be processed as corresponding C++ *file.C* or *file.i* files, and any other file suffix which would otherwise be unrecognized will be processed as a *file.C* file.

This option effectively overrides the environment variable *prefix*_EXTRA_ARGS. This option is only supported by the **c++** command.

-C
Specifies that C and C++ source comments should be retained by the preprocessor. By default, all comments are removed by the preprocessor. This option is ignored except when used with the -E option.

-c
Specifies that only compilations and assemblies be done. Link-edit is not done.

-D *name***[**=*value***]**
Defines a C or C++ macro for use in compilation. If only *name* is provided, a value of `1` is used for the macro it specifies. For information about macros that **c89/cc/c++** automatically define, see Usage Note 5 on page 565. Also see Usage Note 13 on page 567.

Note: The xlc utility has slightly different semantics for processing -D options.

-E
Specifies that output of the compiler preprocessor phase be copied to `stdout`. Object files are not created, and no link-editing is performed.

-e *function*
Specifies the name of the function to be used as the entry point of the program. This can be useful when creating a fetchable program, or a non–C or non–C++ main, such as a COBOL program. Non–C++ linkage symbols of up to 1024 characters in length may be specified. You can specify an

S-name by preceding the function name with double slash (`//`). (For more information about S-names, see Usage Note 23 on page 569.)

Specify a null S-name ("-e //") so that no function name is identified by **c89/cc/c++** as the entry point of the program. In that case, the Program Management Binder (link editor) default rules will determine the entry point of the program. For more information about the Program Management Binder and the ENTRY control statement, see *z/OS MVS Program Management: User's Guide and Reference*.

The function `//ceestart` is the default. When the default function entry point is used, a binder ORDER control statement is generated by **c89/cc/c++** to cause the CEESTART code section to be ordered to the beginning of the program. Specify the name with a trailing blank to disable this behavior, as in `"//ceestart "`. For more information about the Program Management Binder and the ORDER control statement, see *z/OS MVS Program Management: User's Guide and Reference*.

This option may be required when building products which are intended to be installed using the IBM SMP/E product. When installing ++MOD elements with SMP/E, binder control statements should be provided in the JCLIN created to install the product instead of being embedded in the elements themselves.

-F Ignored by **cc**. Provided for compatibility with historical implementations of **cc**. Flagged as an error by **c89** and **c++**.

-f Ignored by **cc**. Provided for compatibility with historical implementations of **cc**. Flagged as an error by **c89** and **c++**.

Historical implementations of C/C++ used this option to enable floating-point support. Floating-point is automatically included in z/OS XL C/C++. However, in z/OS XL C/C++, two types of floating-point support are available:

HEXADECIMAL
Base 16 IBM System z9 hexadecimal format. The IBM System z9 hexadecimal format is referred to as the hexadecimal floating-point format, and is unique to IBM System z9 hardware. This is the default.

IEEE754
Base 2 IEEE-754 binary format. The IEEE-754 binary format is referred to as binary floating-point format. The IEEE-754 binary format is the more common floating point format used on other platforms.

If you are porting an application from another platform, transmitting floating-point numbers between other platforms or workstations, or your application requires the larger exponent range provided by IEEE-754 binary format, then you should consider using IEEE floating-point.

Example: The following is an example of compiling with *IEEE*-754 binary floating point format:

```
c89 -o outfile -Wc,'float(ieee)' file.c
```

-g Specifies that a side file that contains symbolic information is emitted and the executable is to be loaded into read/write storage, which is required for source-level debugging with dbx, and other debuggers.

For 32-bit compiles, if the _DEBUG_FORMAT=ISD environment variable is exported, then **-g** specifies that the output file (executable) is to contain symbolic information and is to be loaded into read/write storage, which is required for source-level debugging with dbx, and other debuggers.

When specified for the compilation phase, the compiler produces symbolic information for source-level debugging.

When specified for the link-editing phase, the executable file is marked as being serially reusable and will always be loaded into read/write storage.

dbx requires that all the executables comprising the process be loaded into read/write storage so that it can set break points in these executables. When dbx is attached to a running process, this cannot be guaranteed because the process was already running and some executables were already loaded. There are two techniques that will cause all the executables comprising the process to be loaded into read-write storage:

1. Specify the **-g** option for the link-editing phase of each executable. After this is done, the executable is always loaded into read/write storage.

   Because the executable is marked as being serially reusable, this technique works except in cases where the executable must be marked as being reentrant. For example:

   - If the executable is to be used by multiple processes in the same user space.
   - If the executable is a DLL that is used on more than one thread in a multithreaded program.

   In these cases, use the following technique instead:
2. Do not specify the **-g** option during the link-editing phase so that the executable will be marked as being reentrant. Before invoking the program, export the environment variable _BPX_PTRACE_ATTACH with a value of `YES`. After you do this, then executables will be loaded into read/write storage regardless of their reusability attribute.

If you compile an MVS data set source using the **-g** option, you can use dbx to perform source-level debugging for the executable file. You must first issue the **dbx use** subcommand to specify a path of double slash (//), causing dbx to recognize that the symbolic name of the primary source file is an MVS data set. For information on the **dbx** command and its use subcommand, see *z/OS UNIX System Services Command Reference*.

For more information on using dbx, see *z/OS UNIX System Services Programming Tools*.

The z/OS UNIX System Services web page also has more information about dbx. Go to http://www.ibm.com/servers/eserver/zseries/zos/unix/.

For more information on the _BPX_PTRACE_ATTACH environment variable, see *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

The GONUMBER option is automatically turned on by the **-g** option, but can also be turned on independently. There is no execution path overhead incurred for turning on this option, only some additional space for the saved line number tables.

For 31-bit compiles and In Storage Debug (ISD) information, the GONUMBER option generates tables that correspond to the input source file line numbers. These tables make it possible for Debug Tools and for error trace back information in CEE dumps to display the source line numbers. Having source line numbers in CEE dumps improves serviceability costs of applications in production.

Example: The following is an example of compiling with the GONUMBER compiler option:

```
c89 -o outfile -Wc,'GONUM' file.c
```

Note: **-g** forces the NOOPTIMIZE compiler option regardless of its position in the command line.

-I *directory*

Note: The **I** option signifies an uppercase `i`, not a lowercase `L`.
-I specifies the directories to be used during compilation in searching for include files (also called header files).

Absolute pathnames specified on `#include` directives are searched exactly as specified. The directories specified using the **-I** option or from the usual places are not searched.

If absolute pathnames are not specified on `#include` directives, then the search order is as follows:

1. Include files enclosed in double quotation marks (") are first searched for in the directory of the file containing the `#include` directive. Include files enclosed in angle-brackets (< >) skip this initial search.
2. The include files are then searched for in all directories specified by the **-I** option, in the order specified.
3. Finally, the include files are searched for in the usual places. (See Usage Note 4 on page 564 for a description of the usual places.)

You can specify an MVS data set name as an include file search directory. Also, MVS data set names can explicitly be specified on `#include` directives. You can indicate both by specifying a leading double slash (`//`).

Example: To include the include file DEF that is a member of the MVS PDS ABC.HDRS, code your C or C++ source as follows:

```
#include <//'abc.hdrs(def)'>
```

MVS data set include files are handled according to z/OS XL C/C++ compiler conversion rules (see Usage Note 4 on page 564). When specifying an `#include` directive with a leading double slash (in a format other than `#include<//'`*dsname*`'>` and `#include<//dd:`*ddname*`>`), the specified name is paired only with MVS data set names specified on the **-I** option. That is, when you explicitly specify an MVS data set name, any z/OS UNIX file system directory names specified on the **-I** option are ignored.

-L *directory*
Specifies the directories to be used to search for archive libraries specified by the **-l** operand. The directories are searched in the order specified, followed by the usual places. You cannot specify an MVS data set as an archive library directory.

For information on specifying C370LIB libraries, see the description of the **-l** *libname* operand. Also see Usage Note 7 on page 565 for a description of the usual places.

-0, -O (-1), -2, -3
Specifies the level of compiler optimization (including inlining) to be used. The level -1 (number one) is equivalent to -O (capital letter O). The level -3 gives the highest level of optimization. The default is -0 (level zero), no optimization and no inlining, when not using IPA (Interprocedural Analysis).

When optimization is specified, the default is ANSIALIAS. The ANSIALIAS default specifies whether type-based aliasing is to be used during optimization. That is, the optimizer assumes that pointers can only be used to access objects of the same type. Type-based aliasing improves optimization. Applications that use pointers that point to objects of a different type will need to specify NOANSIALIAS when the optimization compiler option is specified. If your application works when compiled with no optimization and fails when compiled with optimization, then try compiling your application with both optimization and NOANSIALIAS compiler options.

Notes:

1. Options can also be specified as -O1 (using capital letter O), -O2, and -O3. For further information, see Usage Note 12 on page 566.
2. These options cannot be overridden by specifying optimization options using the **-Wc** syntax. This behavior differs from the behavior of the xlc utility, which allows the use of **-q** and **-Wc** syntax to override the flag optimization options.

Example: The following is an example of a compile with the highest level of optimization and no type-based aliasing:

```
c89 -o outfile -O3 -Wc,NOANSIALIAS file.c
```

When optimization is specified, you may want to obtain a report on the amount of inlining performed and increase or decrease the level of inlining. More inlining will improve application performance and increase application memory usage.

Example: The following is an example of a compile with optimization with no report generated, a threshold of 500 abstract code units, and a limit of 2500 abstract code units:

```
c89 -o outfile -O2 -Wc,'inline(auto,noreport,500,2500)' file.c
```

When using IPA, the default is -O (level 1) optimization and inlining. IPA optimization is independent from and can be specified in addition to this optimization level. IPA is further described in this topic.

If compiling with PDF, the same optimization level must be used in the PDF1 and PDF2 steps.

If you compile your program to take advantage of **dbx** source-level debugging and specify **-g** (see the **-g** option description in this topic), you will always get -0 (level zero) optimization regardless of which of these compiler optimization levels you specify.

In addition to using optimization techniques, you may want to control writable strings by using the **#pragma strings(readonly)** directive or the ROSTRING compiler option. As of z/OS Version 1 Release 2, ROSTRING is the default.

For more information on this topic, refer to reentrancy in z/OS XL C/C++ in *z/OS XL C/C++ Programming Guide* or the description of “ROSTRING | NOROSTRING” on page 214.

-o *outfile*

Specifies the name of the **c89/cc/c++** output file.

If the **-o** option is specified in addition to the **–c** option, and only one source file is specified, then this option specifies the name of the output file associated with the one source file. See *file.o* under “Operands” on page 548 for information on the default name of the output file.

Otherwise the **-o** option specifies the name of the executable file produced during the link-editing phase. The default output file is **a.out**.

-p Ignored by **cc**. Provided for compatibility with historical implementations of **cc**. Flagged as an error by **c89** and **c++**.

-q Ignored by **cc**. Provided for compatibility with historical implementations of **cc**. Flagged as an error by **c89** and **c++**.

-r Specifies that **c89/cc/c++** is to save relocation information about the object files which are processed. When the output file (as specified on **–o**) is created, it is not made an executable file. Instead, this output file can later be used as input to **c89/cc/c++**. This can be used as an alternative to an archive library.

IPA Usage Note:

When using **-r** and link-editing IPA compiled object files, you must link-edit with IPA (see the description of IPA under the -W option). However, the **-r** option is typically not useful when creating an IPA optimized program. This is because link-editing with IPA requires that all of the program information is available to the link editor (that is, all of the object files). It is not acceptable to have unresolved symbols, especially the program entry point symbol (which is usually `main`). The **-r** option is normally used when you wish to combine object files incrementally. You would specify some object files during the initial link-edit that uses **-r**. Later, you would specify the output of the initial link-edit, along with the remaining object files in a final link-edit that is done without using **-r**. In such situations where you wish to combine IPA compiled object files, there is an alternative which does not involve the link editor. That alternative is to concatenate the object files into one larger file. This larger file can later be used in a final link-edit, when the remainder of the object files are also made available. (This concatenation can easily be done using the **cp** or **cat** utilities.)

-s Specifies that the compilation phase is to produce a *file.o* file that does *not* include symbolic information, and that the link-editing phase produce an executable that is marked reentrant. This is the default behavior for **c89/cc/c++**.

-U *name*
Undefines a C or C++ macro specified with *name*. This option affects only macros defined by the -D option, including those automatically specified by **c89/cc/c++**. For information about macros that **c89/cc/c++** automatically define, see Usage Note 5 on page 565. Also see Usage Note 13 on page 567.

Note: The xlc utility uses different semantics for handling the -U option. See Chapter 24, “xlc — Compiler invocation using a customizable configuration file,” on page 577 for more information.

-u *function*
Specifies the name of the function to be added to the list of symbols which are not yet defined. This can be useful if the only input to **c89/cc/c++** is archive libraries. Non–C++ linkage symbols of up to 255 characters in length may be specified. You can specify an S-name by preceding the function name with double slash (//). (For more information about S-names, see Usage Note 23 on page 569.) The function `//ceemain` is the

default for non-IPA Link-editing, and the function `main` is the default for IPA Link-editing. However, if this **-u** option is used, or the DLL link editor option is used, then the default function is not added to the list.

-V This verbose option produces and directs output to `stdout` as compiler, assembler, IPA linker, prelinker, and link editor listings. If the -O, -2, or -3 options are specified and cause **c89/cc/c++** to use the compiler INLINE option, then the inline report is also produced with the compiler listing. Error output continues to be directed to `stderr`. Because this option causes **c89/cc/c++** to change the options passed to the steps producing these listings so that they produce more information, it may also result in additional messages being directed to `stderr`. In the case of the compile step, it may also result in the return code of the compiler changing from 0 to 4.

Note: This option has a different meaning when using the xlc utility. See Chapter 24, "xlc — Compiler invocation using a customizable configuration file," on page 577 for more information.

-v This verbose option causes pseudo-JCL to be written to `stdout` before the compiler, assembler, IPA linker, prelinker, and link editor programs are run.

Example: It also causes phaseid information to be emitted in `stderr`:

```
FSUM0000I  Utility(c89)                    Level(UQ99999)
```

It provides information about exactly which compiler, prelinker, and link editor options are being passed, and also which data sets are being used. If you want to obtain this information without actually invoking the underlying programs, specify the **-v** option more than once on the **c89/cc/c++** command string. For more information about the programs which are executed, see Usage Note 14 on page 567.

-W *phase, option[,option]...*

Specifies options to be passed to the steps associated with the compile, assemble, or link-editing phases of **c89/cc/c++**. The valid phase codes are:

0 Specifies the compile phase (used for both non-IPA and IPA compilation).

a Specifies the assemble phase.

c Same as phase code 0.

I Enables IPA (Interprocedural Analysis) optimization.

Unlike other phase codes, the IPA phase code I does not require that any additional options be specified, but it does allow them. In order to pass IPA suboptions, specify those suboptions using the IPA phase code.

Example: To specify that an IPA Compile should save source line number information, without writing a listing file, specify:

```
c89 -c -W I,list file.c
```

Example: To specify that an IPA Link-edit should write the map file to `stdout`, specify:

```
c89 -W I,map file.o
```

l Specifies the link-editing phase.

- To pass options to the prelinker, the first link-editing phase option must be **p** or P. All the following options are then prelink options.

  Example: To write the prelink map to `stdout`, specify:

  ```
  c89 –W l,p,map file.c
  ```

Note: The prelinker is no longer used in the link-editing phase in most circumstances. If it is not used, any options passed are accepted but ignored. See the environment variable *prefix*_STEPS under "Environment variables" on page 550 for more information about the link-editing phase prelink step.

- To pass options to the IPA linker, the first link-editing phase option must be **i** or **I**. All the following options are then IPA Link options.

  Example: To specify the size of the SPILL area to be used during an IPA Link-edit, you could specify:

  ```
  c89 –W l,I,"spill(256)" file.o
  ```
- To link-edit a DLL (Dynamic Link Library) and produce a side deck, the link-editing phase option DLL must be specified.

  Example: To accomplish this task, you could specify:

  ```
  c89 –o outdll –W l,dll file.o
  ```

Most z/OS XL C/C++ extensions can be enabled by using this option. Those which do not directly pass options through to the underlying steps, or involve files which are extensions to the compile and link-edit model, are described here:

DLL (Dynamic Link Library)

A DLL is a part of a program that is not statically bound to the program. Instead, linkage to symbols (variables and functions) is completed dynamically at execution time. DLLs can improve storage utilization, because the program can be broken into smaller parts, and some parts may not always need to be loaded. DLLs can improve maintainability, because the individual parts can be managed and serviced separately.

In order to create a DLL, some symbols must be identified as being exported for use by other parts of the program. This can be done with the z/OS XL C/C++ **#pragma export** compiler directive, or by using the z/OS XL C/C++ EXPORTALL compiler option. If during the link-editing phase some of the parts have exported symbols, the executable which is created is a DLL. In addition to the DLL, a definition side-deck is created, containing link-editing phase IMPORT control statements which name those symbols which were exported by the DLL. In order for the definition side-deck to be created, the DLL link editor option must be specified. This definition side-deck is subsequently used during the link-editing phase of a program which is to use the DLL. See the *file.x* operand description in the "Operands" on page 548 topic for information on where the definition side-deck is written. In order for the program to refer to symbols exported by the DLL, it must be compiled with the DLL compiler option.

Example: To compile and link a program into a DLL, you could specify:

```
c89 -o outdll -W c,exportall -W l,dll file.c
```

To subsequently use *file.x* definition side-decks, specify them along with any other *file.o* object files specified for **c89/cc/c++** link-editing phase.

Example: To accomplish this task, you could specify:

```
c89 –o myappl –W c,dll myappl.c outdll.x
```

In order to run an application which is link-edited with a definition side-deck, the DLL must be made available (the definition side-deck created along with the DLL is not needed at execution time). When the DLL resides in the z/OS UNIX file system, it must be in either the working directory or in a directory named on the LIBPATH environment variable. Otherwise it must be a member of a data set in the search order used for MVS programs.

Note: For non-DLL C++ compiles, a dummy definition side file will be allocated to prevent the binder from issuing a warning message. If you do want the binder to issue a warning message when an exported symbol is encountered, specify the DLL=NO option for the link-editing phase; for example:

```
c++ -o outfile -W l,dll=no file.C
```

IPA (interprocedural analysis)

IPA optimization is independent from and can be used in addition to the **c89/cc/c++** optimization level options (such as –O). IPA optimization can also improve the execution time of your application. IPA is a mechanism for performing optimizations across function boundaries, even across compilation units. It also performs optimizations not otherwise available with the z/OS XL C/C++ compiler.

When phase code I (capital letter `I`) is specified for the compilation phase, then IPA compilation steps are performed. When phase code I is specified for the link-editing phase, or when the first link-editing phase (code **l**) option is **i** or I, then an additional IPA Link step is performed prior to the prelink and link-edit steps.

With conventional compilation and link-editing, the object code generation takes place during the compilation phase. With IPA compilation and link-editing, the object code generation takes place during the link-editing phase. Therefore, you might need to request listing information about the program (such as with the -V option) during the link-editing phase.

Unlike the other phase codes, phase code I does not require that any additional options be specified. If they are, they should be specified for both the compilation and link-editing phases.

No additional preparation needs to be done in order to use IPA.

Example: To create the executable `myIPApgm` using `c89` with some existing source program `mypgm.c`, you could specify:

```
c89 –W I –o myIPApgm mypgm.c
```

When IPA is used with **c++**, and automatic template generation is being used, phase code I will control whether the automatic template generation compiles are done using IPA. If you do not specify phase code I, then regular compiles will be done. Specifying I as the first option of the link-editing phase option (that is, **`-W l,I`**), will cause the IPA linker to be used, but will not cause the IPA compiler to be used for automatic template generation unless phase code I (that is, -W I) is also specified.

The IPA Profile-Directed Feedback (PDF) option tunes optimizations, where results from sample program execution are used to improve optimization near conditional branches and in frequently executed code sections. The profiling information is placed in the file specified by the PDFNAME(filename) suboption. If PDFNAME(filename) is not specified, the default name of the file containing profile information is PDF.

Note: When using the c89 utility to invoke the compiler for IPA Compile and IPA Link processing using a single command line, some compiler options are not passed to both the IPA Compile and IPA Link steps; for example, the LIST compiler option is not passed. If you want to pass it to both steps, you must use `-Wl,I,list` syntax so that it is also passed to the IPA Link step. The xlc utility passes all compiler options to both the IPA Compile and IPA Link step.

LP64 The LP64 option instructs the compiler to generate AMODE 64 code utilizing the z/Architecture 64-bit instructions.

To compile 64-bit code, specify the z/OS XL C/C++ LP64 compiler option.

Example: The following example shows how to compile and bind using the LP64 option:

```
c89 -o -W c,LP64 -Wl,LP64 file.c
```

XPLINK (Extra Performance Linkage)

z/OS XPLINK provides improved performance for many C/C++ programs. The XPLINK compiler option instructs the z/OS XL C/C++ compiler to generate high performance linkage for subroutine calls. It does so primarily by making subroutine calls as fast and efficient as possible, by reducing linkage overhead, and by passing function call parameters in registers. Furthermore, it reduces the data size by eliminating unused information from function control blocks.

An XPLINK-compiled program is implicitly a DLL-compiled program (the C/C++ DLL compiler option need not be specified along with the XPLINK option). XPLINK improves performance when crossing function boundaries, even across compilation units, since XPLINK uses a more efficient linkage mechanism.

For more information about Extra Performance Linkage, refer to *z/OS Language Environment Programming Guide*.

To use XPLINK, you must both compile and link-edit the program for XPLINK. All C and C++ source files must be compiled XPLINK, as you cannot statically link together XPLINK and non-XPLINK C and C++ object files (with the exception of non-XPLINK ″OS″ linkage). You can however mix XPLINK and non-XPLINK executables across DLL and fetch() boundaries.

To compile a program as XPLINK, specify the z/OS XL C/C++ XPLINK compiler option. If there are any exported symbols in the executable and you want to produce a definition side-deck, specify the DLL link editor option. When XPLINK is specified in the link-editing step, different link-edit libraries will be used.

Example: Here is an example of compiling and link-editing an XPLINK application in one command:

```
c89 -o outxpl -W c,XPLINK -W l,XPLINK,dll file.c
```

In order to execute an XPLINK program, the SCEERUN2 as well as the SCEERUN data set must be in the MVS program search order (see the *prefix*_PLIB_PREFIX environment variable).

You cannot use -W to override the compiler options that correspond to c89 utility flag options, with the following exceptions:

- Listing options (corresponding to -V)
- Inlining options (corresponding to -O, -2, and -3)
- Symbolic options (corresponding to **-s** and **-g**); symbolic options can be overridden only when neither **-s** nor **-g** is specified.

Notes:

1. Most compiler, prelinker, and IPA linker options have a positive and negative form. The negative form is the positive with a prepended NO (as in XREF and NOXREF).
2. The compiler **#pragma options** directives as well as any other `#pragma` directives which are overridden by compiler options, will have no effect in source code compiled by **c89/cc/c++**.
3. Link editor options must be specified in the *name*=*value* format. Both the option *name* and *value* must be spelled out in full. If you do not specify a value, a default value of YES is used, except for the following options, which if specified without a value, have the default values shown here:

ALIASES	ALIASES=ALL
DYNAM	DYNAM=DLL
LET	LET=8
LIST	LIST=NOIMPORT

   Notes:

   a. The binder default is COMPAT=MIN. For downward compatibility (when - `Wc,'target(release)'` is used), COMPAT should also be used (for example, `-Wl,compat=min`, or the specific program object format level supported by the target deployment system, if it is known). For more information, see “Language Environment downward compatibility” on page 8.
   b. As of z/OS V1R8, the default for the COMPAT option is no longer emitted by the c89 utility. In prior releases, the default was COMPAT=CURRENT.
   c. References throughout this document to the link editor are generic references. **c89/cc/c++** specifically uses the Program Management binder for this function.
4. Related information about the z/OS XL C/C++ run-time library, including information about DLL and IPA support, is described in *z/OS XL C/C++ Programming Guide*. Related information about the C and C++ languages, including information about compiler directives, is described in *z/OS XL C/C++ Language Reference*.
5. Since some compiler options are only used by z/OS XL C and some compiler options are only used by z/OS XL C++, you may get warning messages and a compiler return code of 4, if you use this option and compile both C and C++ source programs in the same **c++** command invocation.

6. For more information on the prelinker, see Appendix A, “Prelinking and linking z/OS XL C/C++ programs,” on page 601.
7. Any messages produced by it (CCN messages) are documented in *z/OS XL C/C++ Messages*.
8. You may see run-time messages (CEE or EDC) in executing your applications. These messages are described in *z/OS Language Environment Debugging Guide*.
9. The link editor (the Program Management binder) is described in *z/OS MVS Program Management: User’s Guide and Reference*. The Program Management binder messages are described in *z/OS MVS System Messages, Vol 8 (IEF-IGD)*.

Operands

c89/cc/c++ generally recognize their file operand types by file suffixes. The suffixes shown here represent the default values used by **c89/cc/c++**. See “Environment variables” on page 550 for information on changing the suffixes to be used.

Unlike **c89** and **c++**, which report an error if given an operand with an unrecognized suffix, **cc** determines that it is either an object file or a library based on the file itself. This behavior is in accordance with the environment variable *prefix*_EXTRA_ARGS.

file.a Specifies the name of an archive file, as produced by the **ar** command, to be used during the link-editing phase. You can specify an MVS data set name, by preceding the file name with double slash (//), in which case the last qualifier of the data set name must be *LIB*. The data set specified must be a C370LIB object library or a load library. See the description of the **-l** *libname* operand for more information about using data sets as libraries.

file.C Specifies the name of a C++ source file to be compiled. You can specify an MVS data set name by preceding the file name with double slash (//), in which case the last qualifier of the data set name must be *CXX*. This operand is only supported by the **c++** command.

file.c Specifies the name of a C source file to be compiled. You can specify an MVS data set name by preceding the file name with double slash (//), in which case the last qualifier of the data set name must be *C*. (The conventions formerly used by c89 for specifying data set names are still supported. See the environment variables *prefix*_OSUFFIX_HOSTRULE and *prefix*_OSUFFIX_HOSTQUAL for more information.)

file.I Specifies the name of a IPA linker output file produced during the **c89/cc/c++** link-editing phase, when the -W option is specified with phase code `I`. IPA is further described in the “Options” on page 537 topic. By default the IPA linker output file is written to a temporary file. To have the IPA linker output file written to a permanent file, see the environment variable *prefix*_TMPS under Environment variables.

When an IPA linker output file is produced by **c89/cc/c++**, the default name is based upon the output file name. See the **-o** option description in the Options topic, for information on the name of the output file.

If the output file is named *a.out*, then the IPA linker output file is named *a.I*, and is always in the working directory. If the output file is named *//a.load*, then the IPA linker output file is named *//a.IPA*. If the output file specified already has a suffix, that suffix is replaced. Otherwise the suffix is appended. This file may also be specified on the command line, in which case it is used as a file to be link-edited.

file.i Specifies the name of a preprocessed C or C++ source file to be compiled. You can specify an MVS data set name, by preceding the file name with double slash (//), in which case the last qualifier of the data set name must be *CEX*.

When using the **c++** command, this source file is recognized as a C++ source file, otherwise it is recognized as a C source file. **c++** can be made to distinguish between the two. For more information see the environment variables *prefix*_IXXSUFFIX and *prefix*_IXXSUFFIX_HOST.

file.o Specifies the name of a C, C++, or assembler object file, produced by **c89/cc/c++**, to be link-edited.

When an object file is produced by **c89/cc/c++**, the default name is based upon the source file. If the source file is named *file.c*, then the object file is named *file.o*, and is always in the working directory. If the source file were a data set named *//file.C*, then the object file is named *//file.OBJ*.

If the data set specified as an object file has undefined (U) record format, then it is assumed to be a load module. Load modules are not processed by the prelinker.

You can specify an MVS data set name to be link-edited, by preceding the file name with double slash (//), in which case the last qualifier of the data set name must be *OBJ*.

Example: If a partitioned data set is specified, more than one member name may be specified by separating each with a comma (,):

```
c89 //file.OBJ(mem1,mem2,mem3)
```

file.p Specifies the name of a prelinker composite object file produced during the **c89/cc/c++** link-editing phase. By default, the composite object file is written to a temporary file. To have the composite object file written to a permanent file, see the environment variable *prefix*_TMPS under Environment variables.

When a composite object file is produced by **c89/cc/c++**, the default name is based upon the output file name. See the **-o** option description in the Options topic, for information on the name of the output file.

If the output file is named *a.out*, then the composite object file is named *a.p*, and is always in the working directory. If the output file is named *//a.load*, then the composite object file is named *//a.CPOBJ*. If the output file specified already has a suffix, that suffix is replaced. Otherwise the suffix is appended. This file may also be specified on the command line, in which case it is used as a file to be link-edited.

file.s Specifies the name of an assembler source file to be assembled. You can specify an MVS data set name, by preceding the file name with double slash (//), in which case the last qualifier of the data set name must be *ASM*.

file.x Specifies the name of a definition side-deck produced during the **c89/cc/c++** link-editing phase when creating a DLL (Dynamic Link Library), and used during the link-editing phase of an application using the DLL. DLLs are further described under the -W option.

When a definition side-deck is produced by **c89/cc/c++**, the default name is based upon the output file name. See the **-o** option description in the Options topic, for information on the name of the output file.

If the output file is named *a.dll*, then the definition side-deck is named *a.x*, and is always in the working directory. If the output file is named *//a.DLL*, then the definition side-deck is named *//a.EXP*. If the output file specified already has a suffix, that suffix is replaced. Otherwise the suffix is appended.

You can specify an MVS data set name to be link-edited, by preceding the file name with double slash (//), in which case the last qualifier of the data set name must be *EXP*.

Example: If a partitioned data set is specified, more than one member name may be specified by separating each with a comma (,):

```
c89 //file.EXP(mem1,mem2,mem3)
```

–l *libname*

Specifies the name of an archive library. **c89/cc/c++** searches for the file `lib`*libname*`.a` in the directories specified on the -L option and then in the usual places. The first occurrence of the archive library is used. For a description of the usual places, see Usage Note 7 on page 565.

You can also specify an MVS data set; you must specify the full data set name, because there are no rules for searching library directories.

The data set specified must be a C370LIB object library or a load library. If a data set specified as a library has undefined (U) record format, then it is assumed to be a load library. For more information about how load libraries are searched, see Usage Note 7 on page 565.

Environment variables

You can use environment variables to specify necessary system and operational information to **c89/cc/c++**. When a particular environment variable is not set, **c89/cc/c++** uses the default shown. For information about the JCL parameters used in these environment variables, see *z/OS MVS JCL User's Guide*.

Each environment variable has a *prefix* (shown in italics) that should be replaced by one of the following, depending on the command name used:
- _CC
- _CXX
- _C89

This means that to specify **cc** environment variables, the name shown must be prefixed with _CC (for example, _CC_ACCEPTABLE_RC). To specify **c89** environment variables, the name shown must be prefixed with _C89 (for example, _C89_ACCEPTABLE_RC). To specify **c++**/**cxx** environment variables, the name shown must be prefixed with _CXX (for example, _CXX_ACCEPTABLE_RC). The following examples show how to code one or more MVS data sets:
- export _CXX_LSYSLIB=CEE.SCEELKED
- export _CXX_LSYSLIB=CEE.SCEELKED:CEE.SCEELKEX

Notes:
1. For most environment variables, you can use all three prefixes (_CC, _CXX, _C89). In the list of environment variables that follows, you should assume that all three prefixes can be used unless otherwise indicated.
2. **c89/cc/c++** can accept parameters only in the syntax indicated here. A null value indicates that **c89/cc/c++** should omit the corresponding parameters during dynamic allocation. Numbers in parentheses following the environment

variable name correspond to usage notes, which indicate specific usage information for the environment variable.

*prefix*_**ACCEPTABLE_RC**
The maximum allowed return code (result) of any step (compile, assemble, IPA Link, prelink, or link-edit). If the result is between zero and this value (inclusive), then it is treated internally by **c89/cc/c++** exactly as if it were a zero result, except that message FSUM3065 is also issued. The default value is 4.

When used under **c89/cc/c++**, the prelinker by default returns at least a 4 when there are duplicate symbols or unresolved writable static symbols (but not for other unresolved references). The link editor returns at least a 4 when there are duplicate symbols, and at least an 8 when there are unresolved references and automatic library call was used.

*prefix*_**ASUFFIX** (15)
The suffix by which **c89/cc/c++** recognizes an archive file. This environment variable does not affect the treatment of archive libraries specified as **-l** operands, which are always prefixed with *lib* and suffixed with *.a*. The default value is `a`.

*prefix*_**ASUFFIX_HOST** (15)
The suffix by which **c89/cc/c++** recognizes a library data set. This environment variable does not affect the treatment of data set libraries specified as **-l** operands, which are always used exactly as specified. The default value is LIB.

*prefix*_**CCMODE**
Controls how **c89/cc/c++** does parsing. The default behavior of **c89/cc/c++** is to expect all options to precede all operands. Setting this variable allows compatibility with historical implementations (other **cc** commands). When set to 1, **c89/cc/c++** operates as follows:
- Options and operands can be interspersed.
- The double dash (—) is ignored.

Setting this variable to 0 results in the default behavior. The default value is 0.

*prefix*_**CLASSLIB_PREFIX** (17)
The prefix for the following named data sets used during the compilation phase and execution of your C++ application.

To be used, the following data sets must be cataloged:
- The data sets ${*prefix*_CLASSLIB_PREFIX}.SCLBH.+ contain the z/OS XL C++ Class Library include (header) files.
- The data set ${*prefix*_CLASSLIB_PREFIX}.SCLBSID contains the z/OS XL C++ Class Library definition side-decks.

The following data sets are also used:

The data sets ${*prefix*_CLASSLIB_PREFIX}.SCLBDLL and ${*prefix*_CLASSLIB_PREFIX}.SCLBDLL2 contain the z/OS XL C++ Class Library DLLs and messages.

The preceding data sets contain MVS programs that are invoked during the execution of a C++ application built by **c++**. To be executed correctly, these data sets must be made part of the MVS search order. Regardless of the setting of this or any other **c++** environment variable, **c++** does not affect the MVS search order. These data sets are listed here for information only, to assist in identifying the correct data sets to be added to the MVS program search order.

The default value is the value of the environment variable: _CXX_CLIB_PREFIXThe *prefix*_CLASSLIB_PREFIX environment variable applies only to **c++** and **cxx** command names. _CXX is the only valid prefix.

*prefix*_**CLASSVERSION**

The version of the C++ Class Library to be invoked by **c++**. The setting of this variable allows **c++** to control which C++ Class Library named data sets are used during the **c++** processing phases. It also sets default values for other environment variables.

The format of this variable is the same as the result of the Language Environment C/C++ run-time library function `_librel()`. See *z/OS XL C/C++ Run-Time Library Reference* for a description of the `_librel()` function. The default value is the same as the value for the _CVERSION environment variable. If _CVERSION is not set, then the default value will be the result of the C/C++ Run-Time library `_librel()` function.

The *prefix*_CLASSVERSION environment variable applies only to the **c++** and **cxx** command names. _CXX is the only valid prefix.

*prefix*_**CLIB_PREFIX** (17)

The prefix for the following named data set used during the compilation phase.

The data set ${*prefix*_CLIB_PREFIX}.SCCNCMP contains the compiler programs called by **c89/cc/c++**.

The preceding data set contains MVS programs that are invoked during the execution of **c89/cc/c++** and during the execution of a C/C++ application built by **c89/cc/c++**. To be executed correctly, the data set must be made part of the MVS search order. Regardless of the setting of this or any other **c89/cc/c++** environment variable, **c89/cc/c++** does not affect the MVS search order. The data set is listed here for information only, to assist in identifying the correct data set to be added to the MVS program search order.

The following data set is also used:

The data set ${*prefix*_CLIB_PREFIX}.SCCNOBJ contains object files required to instrument the code for profile-driven feedback optimization.

The default value is CBC.

*prefix*_**CMEMORY**

A suggestion as to the use of compiler C/C++ Runtime Library memory files. When set to `0`, **c89/cc/c++** will prefer to use the compiler NOMEMORY option. When set to `1`, **c89/cc/c++** will prefer to use the compiler MEMORY option. When set to 1, and if the compiler MEMORY option can be used, **c89/cc/c++** need not allocate data sets for the corresponding work files. In this case it is the responsibility of the user to not override the compiler options (using the -W option) with the NOMEMORY option or any other compiler option which implies the NOMEMORY option.

The default value is 1.

*prefix*_**CMSGS** (14)
: The Language Environment national language name used by the compiler program. A null value will cause the default Language Environment NATLANG run-time name to be used, and a non-null value must be a valid Language Environment NATLANG run-time option name (Language Environment run-time options are described in *z/OS Language Environment Programming Guide* . The default value is `"" (null)`.

*prefix*_**CNAME** (14)
: The name of the compiler program called by **c89/cc/c++**. It must be a member of a data set in the search order used for MVS programs. The default value is CCNDRVR. If **c89/cc/c++** is being used with *prefix*_CVERSION set to a release prior to z/OS V1R2, the default value will be CBCDRVR.

*prefix*_**CSUFFIX** (15)
: The suffix by which **c89/cc/c++** recognizes a C source file. The default value is `c`.

*prefix*_**CSUFFIX_HOST** (15)
: The suffix by which **c89/cc/c++** recognizes a C source data set. The default value is C.

*prefix*_**CSYSLIB** (4, 16)
: The system library data set concatenation to be used to resolve `#include` directives during compilation.

Normally `#include` directives are resolved using all the information specified including the directory name. When **c89/cc/c++** can determine that the directory information can be used, such as when the Language Environment include (header) files are installed in the default location (in accordance with *prefix*_INCDIRS), then the default concatenation is `"" (null)`.

When **c89/cc/c++** cannot determine that the directory information can be used, then the default concatenation is:

```
"${prefix_PLIB_PREFIX}.SCEEH.H"
"${prefix_PLIB_PREFIX}.SCEEH.SYS.H"
"${prefix_PLIB_PREFIX}.SCEEH.ARPA.H"
"${prefix_PLIB_PREFIX}.SCEEH.NET.H"
"${prefix_PLIB_PREFIX}.SCEEH.NETINET.H"
```

When this variable is a null value, then no allocation is done for compiler system library data sets. In this case, the use of //DD:SYSLIB on the **-I** option and the `#include` directive will be unsuccessful. Unless there is a dependency on the use of //DD:SYSLIB, it is recommended that for improved performance this variable be allowed to default to a null value.

*prefix*_**CVERSION**
: The version of the z/OS XL C/C++ compiler to be invoked by **c89/cc/c++**. The setting of this variable allows **c89/cc/c++** to control which z/OS XL C/C++ compiler program is invoked. It also sets default values for other environment variables.

The format of this variable is the same as the result of the Language Environment C/C++ run-time library function `_librel()`. See *z/OS XL C/C++ Run-Time Library Reference* for a description of the `_librel()` function. The default value is the result of the C/C++ Run-Time library `_librel()` function.

*prefix*_**CXXSUFFIX** (15)
: The suffix by which **c++** recognizes a C++ source file. The default value is C. This environment variable is only supported by the **c++** and **cxx** command names. _CXX is the only valid prefix.

*prefix*_**CXXSUFFIX_HOST** (15)
: The suffix by which **c++** recognizes a C++ source data set. The default value is CXX. This environment variable is only supported by the **c++** and **cxx** command names. _CXX is the only valid prefix.

*prefix*_**DAMPLEVEL**
: The minimum severity level of dynamic allocation messages returned by dynamic allocation message processing. Messages with severity greater than or equal to this number are written to `stderr`. However, if the number is out of the range shown here (that is, less than `0` or greater than `8`), then **c89/cc/c++** dynamic allocation message processing is disabled. The default value is 4. Following are the values:

0	Informational
1–4	Warning
5–8	Severe

*prefix*_**DAMPNAME** (14)
: The name of the dynamic allocation message processing program called by **c89/cc/c++**. It must be a member of a data set in the search order used for MVS programs. The default dynamic allocation message processing program is described in *z/OS MVS Programming: Authorized Assembler Services Guide*. The default value is IEFDB476.

*prefix*_**DCBF2008** (21)
: The DCB parameters used by **c89/cc/c++** for data sets with the attributes of record format fixed unblocked and minimum block size of 2008. The block size must be in multiples of 8, and the maximum depends on the phase in which it is used but can be at least 5100. The default value is (RECFM=F,LRECL=4088,BLKSIZE=4088).

*prefix*_**DCBU** (21)
: The DCB parameters used by **c89/cc/c++** for data sets with the attributes of record format undefined and data set organization partitioned. This DCB is used by **c89/cc/c++** for the output file when it is to be written to a data set. The default value is (RECFM=U,LRECL=0,BLKSIZE=6144,DSORG=PO).

*prefix*_**DCB121M** (21)
: The DCB parameters used by **c89/cc/c++** for data sets with the attributes of record format fixed blocked and logical record length 121, for data sets whose records may contain machine control characters. The default value is (RECFM=FBM,LRECL=121,BLKSIZE=3630).

*prefix*_**DCB133M** (21)
: The DCB parameters used by **c89/cc/c++** for data sets with the attributes of record format fixed blocked and logical record length 133, for data sets whose records may contain machine control characters. The default value is (RECFM=FBM,LRECL=133,BLKSIZE=3990).

*prefix*_**DCB137** (21)
: The DCB parameters used by **c89/cc/c++** for data sets with the attributes of record format variable blocked and logical record length 137. The default value is (RECFM=VB,LRECL=137,BLKSIZE=882).

*prefix*_**DCB137A** (21)
: The DCB parameters used by **c89/cc/c++** for data sets with the attributes of record format variable blocked and logical record length 137, for data sets whose records may contain ISO/ANSI control characters. The default value is (RECFM=VB,LRECL=137,BLKSIZE=882).

*prefix*_**DCB3200** (21)
: The DCB parameters used by **c89/cc/c++** for data sets with the attributes of record format fixed blocked and logical record length 3200. The default value is (RECFM=FB,LRECL=3200,BLKSIZE=12800).

*prefix*_**DCB80** (21)
: The DCB parameters used by **c89/cc/c++** for data sets with the attributes of record format fixed blocked and logical record length 80. This value is also used when **c89/cc/c++** allocates a new data set for an object file. The default value is (RECFM=FB,LRECL=80,BLKSIZE=3200).

*prefix*_**DEBUG_FORMAT** (21)
: This variable is used to determine to which debug format (DWARF or ISD) the **-g** flag is translated. If _DEBUG_FORMAT is set to DWARF, then **-g** is translated to DEBUG(FORMAT(DWARF)). If _DEBUG_FORMAT is set to ISD, then **-g** is translated to TEST. The default value is DWARF.

  Note: This environment variable only applies to 31-bit compiles.

*prefix*_**ELINES**
: This variable controls whether the output of the -E option will include `#line` directives. `#line` directives provide information about the source file names and line numbers from which the preprocessed source came. The preprocessor only inserts `#line` directives where it is necessary. When set to `1`, the output of the **c89/cc/c++** -E option will include `#line` directives where necessary. When set to `0`, the output will not include any `#line` directives. The default value is 0.

*prefix*_**EXTRA_ARGS**
: The setting of this variable controls whether **c89/cc/c++** treats a file operand with an unrecognized suffix as an error, or attempts to process it. When the **c++** command **-+** option is specified, all suffixes which otherwise would be unrecognized are instead recognized as C++ source, effectively disabling this environment variable. See “Options” on page 537 for information on the **-+** option.

  When set to `0`, **c89/cc/c++** treats such a file as an error and the command will be unsuccessful, because the suffix will not be recognized.

  When set to `1`, **c89/cc/c++** treats such a file as either an object file or a library, depending on the file itself. If it is neither an object file nor a library then the command will be unsuccessful, because the link-editing phase will be unable to process it. The default value for **c89** and **c++** is `0`. The default value for **cc** is 1.

*prefix*_**IL6SYSIX** (7, 16)
: The system definition side-deck list that is used to resolve symbols during the IPA Link step of the link-editing phase when using LP64 (see the description of LP64 in “Options” on page 537). The default value is whatever *prefix*_L6SYSIX is set to or defaults to.

*prefix*_**IL6SYSLIB** (7, 16)
: The system library data set list that is used to resolve symbols during the IPA Link step of the link-editing phase when using LP64 (see the

description of LP64 in "Options" on page 537). The default value is whatever *prefix*_L6SYSLIB is set to or defaults to.

*prefix*_**ILCTL** (14)
The name of the control file used by the IPA linker program. By default the control file is not used, so the -W option must be specified to enable its use, as in:

```
c89 -WI,control ...
```

The default value is `ipa.ctl`.

*prefix*_**ILMSGS** (14)
The name of the message data set member, or the Language Environment national language name, used by the IPA linker program. The default value is whatever *prefix*_CMSGS is. So if *prefix*_CMSGS is set or defaults to `"" (null)`, the default value is `"" (null)`.

*prefix*_**ILNAME** (14)
The name of the IPA linker program called by **c89/cc**. It must be a member of a data set in the search order used for MVS programs. The default value is whatever *prefix*_CNAME is. So if *prefix*_CNAME is set or defaults to CCNDRVR the default value is CCNDRVR.

*prefix*_**ILSUFFIX** (15)
The suffix **c89/cc** uses when creating an IPA linker output file. The default value is `I`.

*prefix*_**ILSUFFIX_HOST** (15)
The suffix **c89/cc** uses when creating an IPA linker output data set. The default value is IPA.

*prefix*_**ILSYSLIB** (7, 16)
The system library data set list to be used to resolve symbols during the IPA Link step of the link-editing phase of non-XPLINK programs. The default value is whatever *prefix*_PSYSLIB is set or defaults to, followed by whatever *prefix*_LSYSLIB is set or defaults to.

*prefix*_**ILSYSIX** (7, 16)
The system definition side-deck list to be used to resolve symbols during the IPA Link step of the link-editing phase in non-XPLINK programs. The default value is whatever *prefix*_PSYSIX is set or defaults to.

*prefix*_**ILXSYSLIB** (7, 16)
The system library data set list to be used to resolve symbols during the IPA Link step of the link-editing phase when using XPLINK (see XPLINK (Extra Performance Linkage) in "Options" on page 537). The default value is whatever *prefix*_LXSYSLIB is set or defaults to.

*prefix*_**ILXSYSIX** (7, 16)
The system definition side-deck list to be used to resolve symbols during the IPA Link step of the link-editing phase when using XPLINK (see XPLINK (Extra Performance Linkage) in "Options" on page 537). The default value is whatever *prefix*_LXSYSIX is set or defaults to.

*prefix*_**INCDIRS** (22)
The directories used by **c89/cc/c++** as a default place to search for include files during compilation (before searching *prefix*_INCLIBS and *prefix*_CSYSLIB). If **c++** is not being used the default value is /usr/include. If **c++** is being used the default value is /usr/include /usr/lpp/cbclib/include.

*prefix*_**INCLIBS** (22)
: The directories used by **c89/cc/c++** as a default place to search for include files during compilation (after searching *prefix*_INCDIRS and before searching *prefix*_CSYSLIB). The default value depends on whether or not **c++** is being used. If **c++** is not being used the default value is //'${*prefix*_PLIB_PREFIX}.SCEEH.+'

  If **c++** is being used, the default value is //'${*prefix*_PLIB_PREFIX}.SCEEH.+' //'${*prefix*_CLIB_PREFIX}.SCLBH.+'

*prefix*_**ISUFFIX** (15)
: The suffix by which **c89/cc/c++** recognizes a preprocessed C source file. The default value is `i`.

*prefix*_**ISUFFIX_HOST** (15)
: The suffix by which **c89/cc/c++** recognizes a preprocessed (expanded) C source data set. The default value is CEX.

*prefix*_**IXXSUFFIX** (15)
: The suffix by which **c++** recognizes a preprocessed C++ source file. The default value is i. This environment variable is only supported by the **c++** and **cxx** command names. _CXX is the only valid prefix.

*prefix*_**IXXSUFFIX_HOST** (15)
: The suffix by which **c++** recognizes a preprocessed (expanded) C++ source data set. The default value is CEX. This environment variable is only supported by the **c++** and **cxx** command names. _CXX is the valid prefix.

*prefix*_**L6SYSIX** (7, 16)
: The system definition side-deck list that resolves symbols during the link-editing phase when using `LP64` (see the description of LP64 in "Options" on page 537). A definition side-deck contains link-editing phase IMPORT control statements, which name symbols that are exported by a DLL. The default value depends on whether or not **c++** is used. If **c++** is not used, the default value is: ${*prefix*_PLIB_PREFIX}.SCEELLIB(CELQS003). If **c++** is used, the default value is the list concatenation:

```
"${prefix_PLIB_PREFIX}.SCEELIB(CELQS003,CELQSCPP,C64)"
"${prefix_CLASSLIB_PREFIX}.SCLBSID(IOSX64)"
```

*prefix*_**L6SYSLIB** (7, 16)
: The system library data set concatenation that is used to resolve symbols during the link-editing step when using LP64 (see the description of LP64 in "Options" on page 537). The default value is the concatenation:

```
"${prefix_PLIB_PREFIX}.SCEEBND2"
"${prefix_SLIB_PREFIX}.CSSLIB"
```

*prefix*_**LIBDIRS** (22)
: The directories used by **c89/cc/c++** as the default place to search for archive libraries which are specified using the **-l** operand. The default value is `/lib /usr/lib`.

*prefix*_**LSYSLIB** (7, 16)
: The system library data set concatenation to be used to resolve symbols during the IPA Link step and the link-edit step of the non-XPLINK link-editing phase. The *prefix*_PSYSLIB libraries always precede the *prefix*_LSYSLIB libraries when resolving symbols in the link-editing phase. The default value is the concatenation:

```
"${prefix_PLIB_PREFIX}.SCEELKEX"
"${prefix_PLIB_PREFIX}.SCEELKED"
"${prefix_SLIB_PREFIX}.CSSLIB"
```

*prefix*_**LXSYSLIB** (7, 16)
: The system library data set concatenation to be used to resolve symbols during the IPA Link step and the link-editing phase when using XPLINK (see XPLINK (Extra Performance Linkage) in "Options" on page 537). The default value is the concatenation:

```
"${prefix_PLIB_PREFIX}.SCEEBND2"
"${prefix_SLIB_PREFIX}.CSSLIB"
```

*prefix*_**LXSYSIX** (7, 16)
: The system definition side-deck list to be used to resolve symbols during the link-editing phase when using XPLINK (see XPLINK (Extra Performance Linkage) in "Options" on page 537). A definition side-deck contains link-editing phase IMPORT control statements naming symbols which are exported by a DLL. The default value depends on whether or not **c++** is being used. For 32-bit objects, if **c++** is not being used, the default value is the list ${*prefix*_PLIB_PREFIX}.SCEELIB(CELHS003,CELHS001). For 32-bit objects, if **c++** is being used with *prefix*_PVERSION and *prefix*_CLASSVERSION defaulted to the current z/OS release, the default value is the list concatenation:

```
"${prefix_PLIB_PREFIX}.SCEELIB(CELHS003,CELHS001,CELHSCPP,C128)"
"${prefix_CLASSLIB_PREFIX}.SCLBSID(IOSTREAM,COMPLEX)"
```

For 32-bit objects, if **c++** is being used with *prefix*_PVERSION and *prefix*_CLASSVERSION set to a release prior to z/OS V1R2 for a 32-bit program, the default value is the list concatenation:

```
"${prefix_PLIB_PREFIX}.SCEELIB(CELHS003,CELHS001,CELHSCPP)"
"${prefix_CLASSLIB_PREFIX}.SCLBSID(ASCCOLL,COMPLEX,IOSTREAM)"
```

Note: For 64–bit objects, see *prefix*_L6SYSIX.

*prefix*_**MEMORY**
: A suggestion as to the use of XL C/C++ run-time library memory files by **c89/cc/c++**. When set to 0, **c89/cc/c++** uses temporary data sets for all work files. When set to 1, **c89/cc/c++** uses memory files for all work files that it can. The default value is 1.

*prefix*_**NEW_DATACLAS** (18)
: The DATACLAS parameter used by **c89/cc/c++** for any new data sets it creates. The default value is `""` (`null`).

*prefix*_**NEW_DSNTYPE** (18, 20)
: The DSNTYPE parameter used by **c89/cc/c++** for any new data sets it creates. The default value is `""` (`null`).

*prefix*_**NEW_MGMTCLAS** (18)
: The MGMTCLAS parameter used by **c89/cc/c++** for any new data sets it creates. The default value is `""` (`null`).

*prefix*_**NEW_SPACE** (18, 19)
: The SPACE parameters used by **c89/cc/c++**for any new data sets it creates. A value for the number of directory blocks should always be specified. When allocating a sequential data set, **c89/cc/c++** automatically ignores the specification. The default value is `(,(10,10,10))`.

*prefix*_**NEW_STORCLAS** (18)
: The STORCLAS parameter used by **c89/cc/c++** for any new data sets it creates. The default value is `""` (`null`).

*prefix*_**NEW_UNIT** (18)
: The UNIT parameter used by **c89/cc/c++** for any new data sets it creates. The default value is `""` (`null`).

*prefix*_**NOCMDOPTS** (27)
: Controls how the compiler processes the default options set by **c89**. Setting this variable to 1, reverts the compiler to the behavior that was available prior to z/OS V1R5, when the compiler was unable to distinguish between the **c89** defaults and the user-specified options. Setting this variable to 0, results in the default behavior where the compiler is now able to recognize **c89** defaults. The default value is 0.

*prefix*_**OPERANDS** (22)
: These operands are parsed as if they were specified after all other operands on the **c89/cc/c++** command line. The default value is `""` (`null`).

*prefix*_**OPTIONS** (22)
: These options are parsed as if they were specified before all other options on the **c89/cc/c++** command line. The default value is `""` (`null`).

*prefix*_**OSUFFIX** (15)
: The suffix by which **c89/cc/c++** recognizes an object file. The default value is `o`.

*prefix*_**OSUFFIX_HOST** (15)
: The suffix by which **c89/cc/c++** recognizes an object data set. The default value is OBJ.

*prefix*_**OSUFFIX_HOSTQUAL**
: The data set name of an object data set is determined by the setting of this option. If it is set to `0`, then the suffix *prefix*_OSUFFIX_HOST is appended to the source data set name to produce the object data set name. If it is set to `1`, then the suffix *prefix*_OSUFFIX_HOST replaces the last qualifier of the source data set name to produce the object data set name (unless there is only a single qualifier, in which case the suffix is appended). The default value is 1.

  Note: Earlier versions of c89 always appended the suffix, which was inconsistent with the treatment of files in the hierarchical file system. It is recommended that any existing data sets be converted to use the new convention.

*prefix*_**OSUFFIX_HOSTRULE**
: The way in which suffixes are used for host data sets is determined by the setting of this option. If it is set to `0`, then data set types are determined by the rule described in the note which follows. If it is set to `1`, then the data set types are determined by last qualifier of the data set (just as a suffix is used to determine the type of hierarchical file system file). Each host file type has an environment variable by which the default suffix can be modified. The default value is 1.

  Notes:

  1. Earlier versions of c89 scanned the data set name to determine if it was an object data set. It searched for the string OBJ in the data set name, exclusive of the first qualifier and the member name. If it was found, the data set was determined to be an object data set, and otherwise it was determined to be a C source data set. It is recommended that any existing data sets be converted to use the new convention. Also, because the earlier convention only provided for recognition of C source files, assembler source cannot be processed if it is used.

2. The **c++** command does not support this environment variable, as the earlier convention would not provide for recognition of both C++ and C source files. Therefore regardless of its setting, **c++** always behaves as if it is set to `1`.

*prefix*_**PLIB_PREFIX** (17)
: The prefix for the following named data sets used during the compilation, assemble, and link-editing phases, and during the execution of your application.

  To be used, the following data sets must be cataloged:

  - The data sets ${*prefix*_PLIB_PREFIX}.SCEEH.+ contain the include (header) files for use with the run-time library functions (where + can be any of H, SYS.H, ARPA.H, NET.H, and NETINET.H).
  - The data set ${*prefix*_PLIB_PREFIX}.SCEEMAC contains COPY and MACRO files to be used during assembly.
  - The data sets ${*prefix*_PLIB_PREFIX}.SCEEOBJ and ${*prefix*_PLIB_PREFIX}.SCEECPP contain run-time library bindings which exploit constructed reentrancy, used during the link-editing phase of non-XPLINK programs.
  - The data set ${*prefix*_PLIB_PREFIX}.SCEELKEX contains C run-time library bindings which exploit L-names used during the link-editing phase of non-XPLINK programs. For more information about L-names, see usage note 23 on page 569.
  - The data set ${*prefix*_PLIB_PREFIX}.SCEELKED contains all other Language Environment run-time library bindings, used during the link-editing phase of non-XPLINK programs.
  - The data set ${*prefix*_PLIB_PREFIX}.SCEEBND2 contains all static Language Environment run-time library bindings, used during the link-editing phase of XPLINK programs.
  - The data set ${*prefix*_PLIB_PREFIX}.SCEELIB contains the definition side-decks for the run-time library bindings, used during the link-editing phase of XPLINK programs.

  The following data sets are also used:

  - The data sets ${*prefix*_PLIB_PREFIX}.SCEERUN and ${*prefix*_PLIB_PREFIX}.SCEERUN2 contains the run-time library programs.

  These data sets contain MVS programs that are invoked during the execution of **c89/cc/c++** and during the execution of a C/C++ application built by **c89/cc/c++**. To be executed correctly, these data sets must be made part of the MVS search order. Regardless of the setting of this or any other **c89/cc/c++** environment variable, **c89/cc/c++** does not affect the MVS program search order. These data sets are listed here for information only, to assist in identifying the correct data sets to be added to the MVS program search order. The default value is CEE.

*prefix*_**PMEMORY**
: A suggestion as to the use of prelinker C/C++ Runtime Library memory files. When set to `0`, **c89/cc/c++** uses the prelinker NOMEMORY option. When set to 1, **c89/cc/c++** uses the prelinker MEMORY option. The default value is 1.

*prefix*_**PMSGS** (14)
: The name of the message data set used by the prelinker program. It must

be a member of the cataloged data set ${*prefix*_PLIB_PREFIX}.SCEEMSGP. The default value is EDCPMSGE.

*prefix*_**PNAME** (14)
: The name of the prelinker program called by **c89/cc/c++**. It must be a member of a data set in the search order used for MVS programs. The prelinker program is shipped as a member of the ${*prefix*_PLIB_PREFIX}.SCEERUN data set. The default value is EDCPRLK.

*prefix*_**PSUFFIX** (15)
: The suffix **c89/cc/c++** uses when creating a prelinker (composite object) output file. The default value is `p`.

*prefix*_**PSUFFIX_HOST** (15)
: The suffix **c89/cc/c++** uses when creating a prelinker (composite object) output data set. The default value is CPOBJ.

*prefix*_**PSYSIX** (16)
: The system definition side-deck list to be used to resolve symbols during the non-XPLINK link-editing phase. A definition side-deck contains link-editing phase IMPORT control statements naming symbols which are exported by a DLL. The default value when **c++** is not being used is null. If **c++** is being used with *prefix*_PVERSION and *prefix*_CLASSVERSION set or defaulted to the current z/OS release, the default value is the list concatenation:

```
"${prefix_PLIB_PREFIX}.SCEELIB(C128)"
"${prefix_CLASSLIB_PREFIX}.SCLBSID(IOSTREAM,COMPLEX)"
```

If **c++** is being used with *prefix*_PVERSION and *prefix*_CLASSVERSION set to a release prior to z/OS V1R2, the default value is the list ${*prefix*_CLASSLIB_PREFIX}.SCLBSID(ASCCOLL,COMPLEX,IOSTREAM)

*prefix*_**PSYSLIB** (16)
: The system library data set list to be used to resolve symbols during the non-XPLINK link-editing phase. The *prefix*_PSYSLIB libraries always precede the *prefix*_LSYSLIB libraries when resolving symbols in the link-editing phase. The default value depends on whether or not **c++** is being used. If **c++** is not being used, the default value is the list containing the single entry:

```
"${prefix_PLIB_PREFIX}.SCEEOBJ"
```

If **c++** is being used, the default value is the list:

```
"${prefix_PLIB_PREFIX}.SCEEOBJ:${prefix_PLIB_PREFIX}.SCEECPP"
```

*prefix*_**PVERSION** (26)
: The version of the Language Environment run-time library to be used with **c89/cc/c++**. The setting of this variable allows **c89/cc/c++** to control which Language Environment named data sets are used during the **c89/cc/c++** processing phases. These named data sets include those required for use of the C/C++ run-time library as well as the ISO C++ Library. It also sets default values for other environment variables.

The format of this variable is the same as the result of the Language Environment C/C++ run-time library function `_librel()`. See *z/OS XL C/C++ Run-Time Library Reference* for a description of the `_librel()` function. The default value is the result of the C/C++ Run-Time library `_librel()` function.

*prefix*_**SLIB_PREFIX** (17)
: The prefix for the named data sets used by the link editor (CSSLIB) and the assembler system library data sets (MACLIB and MODGEN). The data set ${*prefix*_SLIB_PREFIX}.CSSLIB contains the z/OS UNIX assembler callable services bindings. The data sets ${*prefix*_SLIB_PREFIX}.MACLIB and ${*prefix*_SLIB_PREFIX}.MODGEN contain COPY and MACRO files to be used during assembly. These data sets must be cataloged to be used. The default value is SYS1.

*prefix*_**SNAME** (14)
: The name of the assembler program called by **c89/cc/c++**. It must be a member of a data set in the search order used for MVS programs. The default value is ASMA90.

*prefix*_**SSUFFIX** (15)
: The suffix by which **c89/cc/c++** recognizes an assembler source file. The default value is `s`.

*prefix*_**SSUFFIX_HOST** (15)
: The suffix by which **c89/cc/c++** recognizes an assembler source data set. The default value is ASM.

*prefix*_**SSYSLIB** (16)
: The system library data set concatenation to be used to find COPY and MACRO files during assembly. The default concatenation is:

```
"${prefix_PLIB_PREFIX}.SCEEMAC"
"${prefix_SLIB_PREFIX}.MACLIB"
"${prefix_SLIB_PREFIX}.MODGEN"
```

*prefix*_**STEPS**
: The steps that are executed for the link-editing phase can be controlled with this variable. For example, the prelinker step can be enabled, so that the inputs normally destined for the link editor instead go into the prelinker, and then the output of the prelinker becomes the input to the link editor.

This variable allows the prelinker to be used in order to produce output which is compatible with previous releases of **c89/cc/c++**. The prelinker is normally used by **c89/cc/c++** when the output file is a data set which is not a PDSE (partitioned data set extended).

Note: The prelinker and XPLINK are incompatible. When using the link editor XPLINK option, the prelinker cannot be used. Thus, specifying the prelinker on this variable will have no effect.

The format of this variable is a set of binary switches which either enable (when turned on) or disable (when turned off) the corresponding step. Turning a switch on will not cause a step to be enabled if it was not already determined by **c89/cc/c++** that any other conditions necessary for its use are satisfied. For example, the IPA Link step will not be executed unless the -W option is specified to enable the IPA linker. Enabling the IPA linker is described in "Options" on page 537.

Considering this variable to be a set of 32 switches, numbered left-to-right from 0 to 31, the steps corresponding to each of the switches are as follows:

0-27	Reserved
28	TEMPINC/IPATEMP
29	IPALINK
30	PRELINK
31	LINKEDIT

Example: To override the default behavior of **c89/cc/c++** and cause the prelinker step to be run (this is also the default when the output file is a data set which is not a PDSE), set this variable to: `0xffffffff` or the equivalent, `-1`. The default value when the output file is a z/OS UNIX file or a PDSE data set is `0xfffffffd` or the equivalent, `-3`.

Note: The IPATEMP step is the IPA equivalent of the TEMPINC (automatic template generation) step, just as the IPACOMP step is the IPA equivalent of the COMPILE step. See the description of IPA under the -W option for more information.

*prefix*_**SUSRLIB** (16)
The user library data set concatenation to be used to find COPY and MACRO files during assembly (before searching *prefix*_SSYSLIB). The default value is `"" (null)`.

*prefix*_**TMPS**
The use of temporary files by **c89/cc/c++** can be controlled with this variable.

The format of this variable is a set of binary switches which either cause a temporary file to be used (when turned on) or a permanent file to be used (when turned off) in the corresponding step.

The correspondence of these switches to steps is the same as for the variable *prefix*_STEPS. Only the prelinker and IPA linker output can be captured using this variable.

Example: To capture the prelinker output, set this variable to: `0xfffffffD` or the equivalent, `-3`. The default value is `0xffffffff` or the equivalent, -1.

*prefix*_**WORK_DATACLAS** (18)
The DATACLAS parameter used by **c89/cc/c++** for unnamed temporary (work) data sets. The default value is `"" (null)`.

*prefix*_**WORK_DSNTYPE** (18, 20)
The DSNTYPE parameter used by **c89/cc/c++** for unnamed temporary (work) data sets. The default value is `"" (null)`.

*prefix*_**WORK_MGMTCLAS** (18)
The MGMTCLAS parameter used by **c89/cc/c++** for unnamed temporary (work) data sets. The default value is `"" (null)`.

*prefix*_**WORK_SPACE** (18, 19)
The SPACE parameters used by **c89/cc/c++** for unnamed temporary (work) data sets. The default value is (32000,(30,30)).

*prefix*_**WORK_STORCLAS** (18)
The STORCLAS parameter used by **c89/cc/c++** for unnamed temporary (work) data sets. The default value is `"" (null)`.

*prefix*_**WORK_UNIT** (18)
The UNIT parameter used by **c89/cc/c++** for unnamed temporary (work) data sets. The default value is SYSDA.

*prefix*_**XSUFFIX** (15)
The suffix by which **c89/cc/c++** recognizes a definition side-deck file of exported symbols. The default value is `x`.

*prefix*_**XSUFFIX_HOST** (15)
The suffix by which **c89/cc/c++** recognizes a definition side-deck data set of exported symbols. The default value is EXP.

Files

libc.a z/OS XL C/C++ run-time library function library (see Usage Note 7 on page 565).

libm.a C/C++ Runtime Library math function library (see Usage Note 7 on page 565)

libl.a lex function library

liby.a yacc function library

/dev/fd0, **/dev/fd1**, **...**
Character special files required by **c89/cc/c++**. For installation information, see *z/OS UNIX System Services Planning*.

/usr/include
The usual place to search for include files (see Usage Note 4 on page 564).

/lib The usual place to search for run-time library bindings (see Usage Note 7 on page 565).

/usr/lib
The usual place to search for run-time library bindings (see Usage Note 7 on page 565).

Usage notes

1. To be able to specify an operand that begins with a dash (–), before specifying any other operands that do not, you must use the double dash (—) end-of-options delimiter. This also applies to the specification of the `–l` operand. (See the description of environment variable *prefix*_CCMODE for an alternate style of argument parsing.)
2. When invoking **c89/cc/c++** from the shell, any option-arguments or operands specified that contain characters with special meaning to the shell must be escaped. For example, some -W option-arguments contain parentheses. Source files specified as PDS member names contain parentheses; if they are specified as fully qualified names, they contain single quotation marks.

   To escape these special characters, either enclose the option-argument or operand in double quotation marks, or precede each character with a backslash.
3. Some **c89/cc/c++** behavior applies only to hierarchical files (and not to data sets).
   - If the compile or assemble is not successful, the corresponding object file (*file.o*) is always removed.
   - If the DLL option is passed to the link-editing phase, and afterwards the *file.x* file exists but has a size of zero, then that file is removed.
4. MVS data sets may be used as the usual place to resolve C and C++ `#include` directives during compilation.

   Such data sets are installed with the Language Environment run-time library. When it is allocated, searching for these include files can be specified on the **-I** option as *//DD:SYSLIB*. (See the description of environment variable *`prefix`*`_CSYSLIB` for information.

   When include files are MVS PDS members, z/OS XL C/C++ uses conversion rules to transform the include (header) file name on a `#include` preprocessor directive into a member name. If the "//'*dataset_prefix*.+'" syntax is not used for the MVS data set which is being searched for the include file, then this transformation strips any directory name on the `#include` directive, and then takes the first 8 or fewer characters up to the first dot (`.`).

If the "//'*dataset_prefix*.+'" syntax is used for the MVS data set which is being searched for the include file, then this transformation uses any directory name on the `#include` directive, and the characters following the first dot (`.`), and substitutes the "+" of the data set being searched with these qualifiers.

In both cases the data set name and member name are converted to uppercase and underscores (`_`) are changed to at signs (`@`).

If the include (header) files provided by the Language Environment run-time library are installed into the hierarchical file system in the default location (in accordance with the *prefix*_INCDIRS environment variable), then the compiler will use those files to resolve `#include` directives during compilation. **c89/cc/c++** by default searches the directory `/usr/include` as the usual place, just before searching the data sets just described. See the description of environment variables *prefix*_CSYSLIB, *prefix*_INCDIRS, and *prefix*_INCLIBS for information on customizing the default directories to search.

5. Feature test macros control which symbols are made visible in a source file (typically a header file). **c89/cc/c++** automatically defines the following feature test macros along with the errno macro, according to whether or not **cc** was invoked.
   - Other than **cc**
     –D "errno=(*__errno())"
     –D _OPEN_DEFAULT=1
   - **cc**
     –D "errno=(*__errno())"
     –D _OPEN_DEFAULT=0
     –D _NO_PROTO=1

   c89/cc/c++ add these macro definitions only after processing the command string. Therefore, you can override these macros by specifying -D or -U options for them on the command string.
6. The default LANGLVL and related compiler options are set according to whether **cc**, **c89**, or **c++** (**cxx**) was invoked. These options affect various aspects of the compilation, such as z/OS XL C/C++ predefined macros, which are used like feature test macros to control which symbols are made visible in a source file (typically a header file), but are normally not defined or undefined except by this compiler option. They can also affect the language rules used by the compiler. For more information about z/OS XL C/C++ predefined macros, see *z/OS XL C/C++ Language Reference*. The options are shown here in a syntax that the user can specify on the **c89/cc/c++** command line to override them:
   - **c89** (also **c++** (**cxx**) when using a C++ compiler older than z/OS v1r2)
     -W "c,langlvl(ansi),noupconv"
   - **c++** (**cxx**)
     -W "c,langlvl(extended,nolibext,nolonglong)
   - **cc**
     -W "c,langlvl(commonc),upconv"
7. By default the usual place for the -L option search is the `/lib` directory followed by the `/usr/lib` directory. See the description of environment variable *prefix*_LIBDIRS for information on customizing the default directories to search.

   The archive libraries `libc.a` and `libm.a` exist as files in the usual place for consistency with other implementations. However, the run-time library bindings are not contained in them. Instead, MVS data sets installed with the Language

Environment run-time library are used as the usual place to resolve run-time library bindings. In the final step of the link-editing phase, any MVS load libraries specified on the **-l** operand are searched in the order specified, followed by searching these data sets. See the *prefix*_PLIB_PREFIX description, as well as descriptions of the environment variables featured in the following list.

*prefix*_ILSYSLIB
*prefix*_ILSYSIX
*prefix*_LSYSLIB
*prefix*_PSYSIX
*prefix*_PSYSLIB

This list of environment variables affects the link-editing phase of c89, but only for non-XPLINK link-editing. See XPLINK (Extra Performance Linkage) in "Options" on page 537.

The following list of environment variables affects the link-editing phase of c89, but only for ILP32 XPLINK link-editing. See XPLINK (Extra Performance Linkage) in "Options" on page 537.

*prefix*_ILXSYSLIB
*prefix*_ILXSYSIX
*prefix*_LXSYSLIB
*prefix*_LXSYSIX

The following list of environment variables affects the link-editing phase of c89, but only for LP64 link-editing. See the description of LP64 in "Options" on page 537.

*prefix*_IL6SYSLIB
*prefix*_IL6SYSIX
*prefix*_L6SYSLIB
*prefix*_L6SYSIX

8. Because archive library files are searched when their names are encountered, the placement of **-l** operands and *file.a* operands is significant. You may have to specify a library multiple times on the command string, if subsequent specification of *file.o* files requires that additional symbols be resolved from that library.
9. When the prelinker is used during the link-editing phase, you cannot use as input to **c89/cc/c++** an executable file produced as output from a previous use of **c89/cc/c++**. The output of **c89/cc/c++** when the **-r** option is specified (which is not an executable file) may be used as input.
10. All MVS data sets used by **c89/cc/c++** must be cataloged (including the system data sets installed with the z/OS XL C/C++ compiler and the Language Environment run-time library).
11. **c89/cc/c++** operation depends on the correct setting of their installation and configuration environment variables (see "Environment variables" on page 550). Also, they require that certain character special files are in the `/dev` directory. For additional installation and configuration information, see *z/OS UNIX System Services Planning*.
12. Normally, options and operands are processed in the order read (from left to right). Where there are conflicts, the last specification is used (such as with **-g** and **-s**). However, some c89 utility flag options will override others, regardless of the order in which they are specified. The option priorities, in order of highest to lowest, are as follows:

    -v specified twice
    : The pseudo-JCL is printed only, but the effect of all the other options and operands as specified is reflected in the pseudo-JCL.

-E Overrides -0, -O, -1, -2, -3, -V, **-c**, **-g** and **-s** (also ignores any *file.s* files).

-g Overrides -0, -O, -1, -2, -3, and **-s**.

-s Overrides **-g** (the last one specified is honored).

-0 (zero), -O (capital letter O), -1, -2, -3, -V, -c
All are honored if not overridden. -0, -O, -1, -2, -3 override each other (the last one specified is honored).

Note: The preferred way for specifying optimization options, is -O (capital letter O) followed by a number; for example, -O2.

13. For options that have option-arguments, the meaning of multiple specifications of the options is as follows:

-D All specifications are used. If the same name is specified on more than one -D option, only the first definition is used.

-e The entry function used will be the one specified on the last **-e** option.

-I All specifications are used. If the same directory is specified on more than one -I option, the directory is searched only the first time.

-L All specifications are used. If the same directory is specified on more than one -L option, the directory is searched only the first time.

-o The output file used will be the one specified on the last **-o** option.

-U All specifications are used. The name is *not* defined, regardless of the position of this option relative to any -D option specifying the same name.

-u All specifications are used. If a definition cannot be found for any of the functions specified, the link-editing phase will be unsuccessful.

-W All specifications are used. All options specified for a phase are passed to it, as if they were concatenated together in the order specified.

14. The following environment variables can be at most eight characters in length. For those whose values specify the names of MVS programs to be executed, you can dynamically alter the search order used to find those programs by using the STEPLIB environment variable.

c89/cc/c++ environment variables do not affect the MVS program search order. Also, for **c89/cc/c++** to work correctly, the setting of the STEPLIB environment variable should reflect the Language Environment library in use at the time that **c89/cc/c++** is invoked.

For more information on the STEPLIB environment variable, see *z/OS UNIX System Services Planning*. It is also described under the **sh** command. Note that the STEPLIB allocation in the pseudo-JCL produced by the **-v** verbose option is shown as a comment, and has no effect on the MVS program search order. Its appearance in the pseudo-JCL is strictly informational.

*prefix*_CMSGS
*prefix*_CNAME
*prefix*_DAMPNAME
*prefix*_ILCTL
*prefix*_ILNAME
*prefix*_ILMSGS
*prefix*_PMSGS
*prefix*_PNAME

*prefix*_SNAME

15. The following environment variables can be at most 15 characters in length. You should not specify any dots `(.)` when setting these environment variables since they would then never match their corresponding operands:
    *prefix*_ASUFFIX
    *prefix*_ASUFFIX_HOST
    *prefix*_CSUFFIX
    *prefix*_CSUFFIX_HOST
    *prefix*_CXXSUFFIX
    *prefix*_CXXSUFFIX_HOST
    *prefix*_ISUFFIX
    *prefix*_ISUFFIX_HOST
    *prefix*_ILSUFFIX
    *prefix*_ILSUFFIX_HOST
    *prefix*_IXXSUFFIX
    *prefix*_IXXSUFFIX_HOST
    *prefix*_OSUFFIX
    *prefix*_OSUFFIX_HOST
    *prefix*_PSUFFIX
    *prefix*_PSUFFIX_HOST
    *prefix*_SSUFFIX
    *prefix*_SSUFFIX_HOST
    *prefix*_XSUFFIX
    *prefix*_XSUFFIX_HOST
16. The following environment variables are parsed as colon-delimited data set names, and represent a data set concatenation or a data set list. The maximum length of each specification is 1024 characters:
    *prefix*_CSYSLIB
    *prefix*_IL6SYSIX
    *prefix*_IL6SYSLIB
    *prefix*_ILSYSIX
    *prefix*_ILSYSLIB
    *prefix*_ILXSYSIX
    *prefix*_ILXSYSLIB
    *prefix*_L6SYSIX
    *prefix*_L6SYSLIB
    *prefix*_LSYSLIB
    *prefix*_LXSYSIX
    *prefix*_LXSYSLIB
    *prefix*_PSYSIX
    *prefix*_PSYSLIB
    *prefix*_SSYSLIB
    *prefix*_SUSRLIB
17. The following environment variables can be at most 44 characters in length:
    *prefix*_CLASSLIB_PREFIX
    *prefix*_CLIB_PREFIX
    *prefix*_PLIB_PREFIX
    *prefix*_SLIB_PREFIX
18. The following environment variables can be at most 63 characters in length:
    *prefix*_NEW_DATACLAS
    *prefix*_NEW_DSNTYPE
    *prefix*_NEW_MGMTCLAS
    *prefix*_NEW_SPACE
    *prefix*_NEW_STORCLAS
    *prefix*_NEW_UNIT

*prefix*_WORK_DATACLAS
*prefix*_WORK_DSNTYPE
*prefix*_WORK_MGMTCLAS
*prefix*_WORK_SPACE
*prefix*_WORK_STORCLAS
*prefix*_WORK_UNIT

19. The following environment variables are for specification of the SPACE parameter, and support only the syntax as shown with their default values (including all commas and parentheses). Also as shown with their default values, individual subparameters can be omitted, in which case the system defaults are used.
    *prefix*_NEW_SPACE
    *prefix*_WORK_SPACE
20. The following environment variables are for specification of the DSNTYPE parameter, and support only the subparameters LIBRARY or PDS (or null for no DSNTYPE):
    *prefix*_NEW_DSNTYPE
    *prefix*_WORK_DSNTYPE
21. The following environment variables can be at most 127 characters in length:
    *prefix*_DCBF2008
    *prefix*_DCBU
    *prefix*_DCB121M
    *prefix*_DCB133M
    *prefix*_DCB137
    *prefix*_DCB137A
    *prefix*_DCB3200
    *prefix*_DCB80
    *prefix*_DEBUG_FORMAT

    These environment variables are for specification of DCB information, and support only the following DCB subparameters, with the noted restrictions:

    RECFM
        Incorrect values are ignored.
    LRECL
        None
    BLKSIZE
        None
    DSORG
        Incorrect values are treated as if no value had been specified.
22. The following environment variables are parsed as blank-delimited words, and therefore no embedded blanks or other white-space is allowed in the value specified. The maximum length of each word is 1024 characters:
    *prefix*_INCDIRS
    *prefix*_INCLIBS
    *prefix*_LIBDIRS
    *prefix*_OPTIONS
    *prefix*_OPERANDS
23. An S-name is a *short* external symbol name, such as produced by the z/OS XL C/C++ compiler when compiling C programs with the NOLONGNAME option. An L-name is a *long* external symbol name, such as produced by the z/OS XL C/C++ compiler when compiling C programs with the LONGNAME option.
24. The z/OS XL C/C++ run-time library supports a file naming convention of // (the filename can begin with exactly two slashes). **c89/cc/c++** indicate that the file naming convention of // can be used.

However, the Shell and Utilities feature *does not* support this convention. Do not use this convention (//) unless it is specifically indicated (as here in **c89/cc/c++**). The z/OS Shell and Utilities feature does support the POSIX file naming convention where the filename can be selected from the set of character values excluding the slash and the null character.

25. When coding in C and C++, **c89**, **cc**, and **c++**, by default, produce reentrant executables. For more information on reentrancy, see *z/OS XL C/C++ Programming Guide*. When coding in assembly language, the code must not violate reentrancy. If it does, the resulting executable may not be reentrant.
26. The *prefix*_CVERSION, *prefix*_PVERSION and *prefix*_CLASSVERSION environment variables are set to a hex string in the format 0xPVVRRMMM where P is product, VV is version, RR is release and MMM is modification level. For example, the *prefix*_CVERSION and *prefix*_CLASSVERSION for the z/OS V1R2 compiler is 0x41020000.
27. c89 passes some options to the compiler so that expected behavior is achieved; for example, POSIX behavior. These options are passed onto the compiler as defaults that the user can overwrite. When default options passed by c89 are in conflict with options or pragmas that the user specified, the compiler issues diagnostic messages and may terminate processing. Since the user did not specify options that c89 passed as defaults, these messages may confuse the user. Prior to the z/OS V1R5 release, the compiler was unable to differentiate between the options that c89 passed as defaults and the user-specified options so it was unable to correctly resolve conflicting pragma/option combinations. In some cases, the compiler would overwrite pragmas with the options that c89 passed as defaults thus limiting a user's ability to use pragmas. As of z/OS V1R5, the compiler is now able to recognize c89 defaults and avoid confusion from messages for options, which were not explicitly specified by the user, and overriding pragmas, when the user did not explicitly request it. It is believed that most users will benefit from this feature so it is the default behavior. To enable the old behavior, environment variable *prefix*_NOCMDOPTS must have a nonzero value. **Example:** The following sequence will preserve the old behavior:

    ```
    export _C89_NOCMDOPTS=1
    c89 -o hello hello.c
    ```
28. The following example shows the concatenation of data sets in environment variables. It shows how to use an environment variable to setup the SYSLIB DD when using the **c89** command name:

    ```
    export _C89_LSYSLIB="CEE.SCEELKEX:CEE.SCEELKED:CBC.SCCNOBJ:SYS1.CSSLIB"
    ```

    This environment variable will produce the following SYSLIB concatenation:

    ```
    //SYSLIB   DD  DSN=CEE.SCEELKEX,DISP=SHR
    //         DD  DSN=CEE.SCEELKED,DISP=SHR
    //         DD  DSN=CBC.SCCNOBJ,DISP=SHR
    //         DD  DSN=SYS1.CSSLIB,DISP=SHR
    ```

Localization

c89/cc/c++ use the following localization environment variables:
- LANG
- LC_ALL
- LC_CTYPE
- LC_MESSAGES

Exit values

0	Successful completion.
1	Failure due to incorrect specification of the arguments.
2	Failure processing archive libraries: • Archive library was not in any of the library directories specified. • Archive library was incorrectly specified, or was not specified, following the **-l** operand.
3	Step of compilation, assemble, or link-editing phase was unsuccessful.
4	Dynamic allocation error, when preparing to call the compiler, assembler, IPA linker, prelinker, or link editor, for one of the following reasons: • The file or data set name specified is incorrect. • The file or data set name cannot be opened.
5	Dynamic allocation error, when preparing to call the compiler, assembler, prelinker, IPA linker, or link editor, due to an error being detected in the allocation information.
6	Error copying the file between a temporary data set and a hierarchical file system file (applies to the **-2** option, when processing assembler source files, and **-r** option processing).
7	Error creating a temporary control input data set for the link-editing phase.
8	Error creating a temporary system input data set for the compile or link-editing phase.

Portability

For **c89**, X/Open Portability Guide, POSIX.2 C-Language Development Utilities Option.

For **cc**, POSIX.2 C-Language Development Utilities Option, UNIX systems.

The following are extensions to the POSIX standard:
- The **-v**, -V, -0, -1, -2 and -3 options
- DLL support
- IPA optimization support
- The behavior of the **-o** option in combination with the **-c** option and a single source file.

Note: **-Ox** (where **x** is 0, 1, 2, or 3) is equivalent to **-x** because **-x** overrides -O. This happens to match the standard compliant syntax of optimization level **x** (**-Ox**), but **Ox** is not treated as a single entity. It may appear redundant to use **-Ox** but it is recommended because it improves portability. In order to avoid creating non-portable legacy, the xlc utility does not support **-x** extension syntax. For example, the following are equivalent but the first syntax is recommended:

```
c89 -O2 hello.c
c89 -2 hello.c
```

Features have been added to z/OS releases, which have made it easier to port applications from other platforms to z/OS and improve performance. For compatibility reasons, these portability and performance enhancements could not be made the default. If you are porting an application from another platform to z/OS, you may want to start by specifying the following options:

```
c89 -o HelloWorld -2 -Wc,NOANSIALIAS -Wc,XPLINK\
-Wl,XPLINK -Wc,'FLOAT(IEEE)' -Wc,'GONUM' HelloWorld.c
```

Note: The string shown in this example is one line (it had to be split to fit the page). A space exists between `-Wc,XPLINK` and `-Wl,XPLINK`.

Related information

See the information on the following utilities in *z/OS UNIX System Services Command Reference*: ar, dbx, file, lex, makedepend, nm, strings, strip, and yacc.

Chapter 22. dbgld — Create a module map for debugging

Format

dbgld
  [*option*] ...
  file

Description

The compiler creates a `.dbg` file for each compilation unit if the DEBUG compiler option is specified. The path names of all of the `.dbg` files are then stored in the module, which is an executable file or a DLL. The **dbgld** command opens all of the `.dbg` files associated with the module and stores all of the functions, global variables, external types, and source files in a single module map file with a `.mdbg` extension. In addition, the contents of all of the `.dbg` files are packaged together into this same `.mdbg` file. Debuggers that support demand load can use the `.mdbg` file for faster access to debug information.

The **dbgld** command only needs to be executed once after binding. If the original source files are not available at debugging time (for example, the source files are moved into a different directory or the debugging and compiling are performed on different machines), you can add the source file contents to the debug information. A debugger that supports captured source can then retrieve the source file contents from the debug side file. For more information on using the module map to improve debugger performance, see *z/OS Common Debug Architecture User's Guide*.

Options

option

-c Adds captured source file to the module map, which consists of all files that contain executable lines of code.

-v Writes the version information to `stderr`.

file Is the module name, which can be:

- The absolute path name of a z/OS UNIX System Services file
- The relative path name of a z/OS UNIX System Services file
- A fully qualified MVS data set (PDS member)

The output of the **dbgld** command is a file with the name of the module followed by a `.mdbg` extension. The file will always be written in the current directory. For example, if the module name is `/mypath/mymodule`, a file called `mymodule.mdbg` will be created in the current directory. If the file already exists, it will be overwritten.

Restrictions

The following restrictions apply to the use of **dbgld**:

- The source files must be compiled with the DEBUG compiler option.
- The name of a valid module must be passed into the dbgld utility. The module must be bound with the EDIT=YES binder option, which is the default. An error message will be generated if EDIT=NO.
- The `.dbg` files associated with the module must exist in the directories where they were during compilation. Otherwise, they will not be added to the module

map and no debug information will be available to the compilation units via the module map during debugging. An error message will be generated for each `.dbg` file that is not found.

- Because the **dbgld** command always creates the `.mdbg` file in the current directory, the command must be run from a directory that has write permission.

Examples

The following example shows how to compile `hello1.c` and `hello2.c` and create a module map in a file called `hello.mdbg`.

```
xlc -g hello1.c hello2.c -o hello
dbgld hello
```

The following example shows how to compile `hello1.c` and `hello2.c` and create a module map in a file called `hello.mdbg` which contains captured source.

```
xlc -g hello1.c hello2.c -o hello
dbgld -c hello
```

The following example shows how to display the version information for the dbgld utility and the Common Debug Architecture run times when creating the module map.

```
dbgld -v hello
```

The output is:

```
CDA0000I Utility(dbgld ) Level(level name)
CDA0000I Library(elf ) Level(level name)
CDA0000I Library(dwarf ) Level(level name)
CDA0000I Library(ddpi ) Level(level name)
```

If the **-g** option is missing during compilation, `hello.mdbg` will not be generated and a warning message will be printed, as shown in the following example:

```
xlc hello1.c hello2.c -o hello
dbgld hello
```

The output is a warning message stating that no debug information was found in `hello`.

Exit values

The exit values for **dblgd** are:

0	Successful completion
4	Warning
8	Error
12	Severe error

Chapter 23. CDADBGLD — Create a debug side file for the module map

Description

The CDADBGLD utility is the MVS batch equivalent of the **dbgld** command. You can use this utility if you do not have z/OS UNIX System Services.

The compiler creates a debug side file for each compilation unit if the DEBUG compiler option is specified. The path names or data set names of all the debug side files are then stored in the module, which is an executable file or a DLL. The CDADBGLD utility opens all of the debug side files associated with the module and stores all of the functions, global variables, external types, and source files in a module map. In addition, the contents of all of the debug side files are packaged together into this same module map. Debuggers that support demand load can use the side file for faster access to debug information.

The CDADBGLD utility only needs to be executed once after binding. The performance of the debugger, especially the start time, will be significantly improved if the CDADBGLD utility is executed before the execution of a debugger. For more information on using the module map to improve debugger performance, see *z/OS Common Debug Architecture User's Guide*. For information on the CDADBGLD cataloged procedure, which executes the CDADBGLD utility, see Chapter 12, "Cataloged procedures and REXX EXECs," on page 465.

Options

INFILE
Specifies the module name.

OUTFILE
Specifies the module level debug side file that will be written to.

CPARM
Passes options into the CDADBGLD utility. The valid options are:

- **VERSION**: Displays the version of CDADBGLD as well as the Common Debug Architecture run-time `phaseid` information.
- **CAPSRC**: Adds captured source to the module map.

Restrictions

The following restrictions apply to the use of CDADBGLD:

- The source files must be compiled with the DEBUG compiler option.
- The name of a valid module must be passed into the CDADBGLD utility. The module must be bound with the EDIT=YES binder option, which is the default. An error message will be generated if EDIT=NO.
- The compilation unit debug side files associated with the module must exist in the directories where they were during compilation. Otherwise, they will not be added to the module map and no debug information will be available to the compilation units via the module map during debugging.
- If the module map file already exists, this file must have write permission.

© Copyright IBM Corp. 1998, 2009

Example

The following example shows how to invoke the CDADBGLD utility, specify the module name of `MYHLQ.MOD(TEST)`, display the version of CDADBGLD, and create a module level debug side file called `MYHLQ.MDBG(TEST)`:

```
//CDADBGLD EXEC CDADBGLD,
// INFILE='MYHLQ.MOD(TEST)',
// CPARM='VERSION',
// OUTFILE='MYHLQ.MDBG(TEST),DISP=SHR'
```

Exit values

The exit values for CDADBGLD are:

0	Successful completion
4	Warning
8	Error
12	Severe error

Chapter 24. xlc — Compiler invocation using a customizable configuration file

Format

```
xlc    | xlc_x    | xlc_64
xlC    | xlC_x    | xlC_64
xlc++  | xlc++_x  | xlc++_64
cc     | cc_x     | cc_64
c89    | c89_x    | c89_64
c99    | c99_x    | c99_64
cxx    | cxx_x    | cxx_64
c++    | c++_x    | c++_64
```

Description

xlc is a utility that uses an external configuration file to control the invocation of the compiler. **xlc** and related commands compile C and C++ source files. They also process assembler source files and object files.

Note: Unless the **-c** option is specified, xlc calls the binder to produce an executable module.

All commands accept the following input files with their default z/OS UNIX file system and host suffixes:

z/OS UNIX System Services files:
- filename with .C suffix (C++ source file)
- filename with .c suffix (C source file)
- filename with .i suffix (preprocessed C or C++ source file)
- filename with .o suffix (object file for binder/IPA Link)
- filename with .s suffix (assembler source file)
- filename with .a suffix (archive library)
- filename with .p suffix (prelinker output file for the binder/IPA Link)
- filename with .I suffix (IPA Link output file for the binder)
- filename with .x suffix (definition side-file or side deck)

Host files:
- filename with .CXX suffix (C++ source host file)
- filename with .C suffix (C source host file)
- filename with .CEX suffix (preprocessed C or C++ source host file)
- filename with .OBJ suffix (object host file for the binder/IPA Link)
- filename with .ASM suffix (assembler source host file)
- filename with .LIB suffix (host archive library)
- filename with .CPOBJ suffix (prelinker output host file for the binder/IPA Link)
- filename with .IPA suffix (IPA Link output host file for the binder
- filename with .EXP suffix (host definition side-file or side deck)

© Copyright IBM Corp. 1998, 2009

Note: For host files, the host data set name must by preceded by a double slash (//). The last qualifier of the data set name is .C instead of a file name with a .C suffix.

The xlc utility invokes the assembler, the z/OS XL C/C++ compiler, and the binder. Invocation of the compiler and the binder is described in “Invoking the compiler” on page 589 and “Invoking the binder” on page 589.

Invocation commands

The xlc utility provides two basic compiler invocation commands, **xlc** and **xlC** (**xlc++**), along with several other compiler invocation commands to support various C/C++ language levels and compilation environments. In most cases, you would use the **xlc** command to compile C source files and **xlC** (**xlc++**) command to compile C++ source files.

You can however, use other forms of the command if your particular environment requires it. The various compiler invocation commands for C are:
- **xlc**
- **cc**
- **c89**
- **c99**
- **xlc_x**
- **cc_x**
- **c89_x**
- **c99_x**
- **xlc_64**
- **cc_64**
- **c89_64**
- **c99_64**

The various compiler invocation commands for C++ are:
- **xlC** (**xlc++**)
- **cxx**
- **c++**
- **xlC_x** (**xlc++_x**)
- **c++_x**
- **cxx_x**
- **xlC_64** (**xlc++_64**)
- **c++_64**
- **cxx_64**

The two basic compiler invocation commands appear as the first entry of each of these list items. Select an invocation command using the following criteria:

xlc
Invokes the compiler for C source files with a default language level of ANSI, the compiler option **-qansialias** to allow type-based aliasing, and the compiler option **-qcpluscmt** to allow C++ style comments (//).

xlC (xlc++)
Invokes the compiler so that source files are compiled as C++ language source code.

Files with .c suffixes, assuming you have not used the **-+** compiler option, are compiled as C language source code with a default language level of ANSI, and compiler option **-qansialias** to allow type-based aliasing.

If any of your source files are C++, you must use this invocation to link with the correct run-time libraries.

cc Invokes the compiler for C source files with a default language level of extended and compiler options **-qnoro** and **-qnoroconst** (to provide placement of string literals or constant values in read/write storage).

Use this invocation for legacy C code that does not require compliance with ANSI C. This invocation is intended to provide the same compiler behavior as when invoked by the **cc** command name of the c89 utility.

c89 Invokes the compiler for C source files, with a default language level of ANSI, and specifies compiler options **-qansialias** (to allow type-based aliasing) and **-qnolonglong** (disabling use of long long). Use this invocation for strict conformance to the *ISO/IEC 9899:1990* standard. This invocation is intended to provide the same compiler behavior as when invoked by the **c89** command name of the c89 utility.

c99 Invokes the compiler for C source files, with a default language level of STDC99 and specifies compiler option **-qansialias** (to allow type-based aliasing). Use this invocation for strict conformance to the *ISO/IEC 9899:1999* standard.

cxx/c++
cxx and **c++** invoke the compiler for C++ language source code. Both are intended to provide the same compiler behavior as when invoked using the **cxx** and **c++** command names of the c89 utility.

You can combine the previously described command names with the following suffixes:

_x Command invocations using command names with suffix `_x` are the same as invocations using names without suffixes, except the **-qxplink** option is also specified and appropriate XPLINK libraries are used in the link step. If you are building an XPLINK application, you no longer need to use command names with suffix `_x` to link with the correct run-time libraries. This can be achieved through the new configuration attributes that have been introduced to enable XPLINK behavior without the use of suffixes. See “Configuration file attributes” on page 581 for further information.

_64 Command invocations using command names with suffix `_64` are the same as invocations using names without suffixes, except the **-q64** option is also specified and appropriate 64-bit libraries are used in the link step. If you are building a 64-bit application, you no longer need to use command names with suffix `_64` to link with the correct run-time libraries. This can be achieved through the new configuration attributes that have been introduced to enable 64-bit behavior without the use of suffixes. See “Configuration file attributes” on page 581 for further information.

Notes:

1. Suffixes are used as a naming convention and do not enforce behavior. The content of the command line will take precedence over the suffixes.
2. When compiling and linking a C++ application using a single command line invocation, the application will be correctly link edited with any stanza if at least one C++ source file is specified on the command line. If only object files or a

mix of C sources and C++ object files are specified on the command line, a C++ stanza must be used to correctly link edit the application.

Setting up the compilation environment

Before you compile your C and C++ programs, you must set up the environment variables and the configuration file for your application. For more information on the configuration file, see "Setting up a configuration file" on page 581.

Environment variables

You can use environment variables to specify necessary system information.

Setting environment variables

Different commands are used to set the environment variables depending on whether you are using the z/OS UNIX System Services shell (sh), which is based on the Korn Shell and is upward-compatible with the Bourne shell, or tcsh shell, which is upward-compatible with the C shell. To determine the current shell, use the echo command, which is **echo $SHELL**.

The z/OS UNIX System Services shell path is /bin/sh. The tcsh shell path is /bin/tcsh.

For more information about the NLSPATH and LANG environment variables, see *z/OS XL C/C++ Programming Guide* and *z/OS UNIX System Services Command Reference*.

Setting environment variables in z/OS shell

The following statements show how you can set environment variables in the z/OS shell:

```
LANG=En_US
NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/%L/%N.cat
PATH=/bin:/usr/lpp/cbclib/xlc/bin${PATH:+:${PATH}}
export LANG NLSPATH PATH
```

To set the variables so that all users have access to them, add the commands to the file /etc/profile. To set them for a specific user only, add the commands to the .profile file in the user's home directory. The environment variables are set each time the user logs in.

Setting environment variables in tcsh shell

The following statements show how you can set environment variables in the tcsh shell:

```
setenv LANG En_US
setenv NLSPATH /usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/%L/%N.cat
setenv PATH /bin:/usr/lpp/cbclib/xlc/bin${PATH:+:${PATH}}
```

To set the variables so that all users have access to them, add the commands to the file /etc/csh.cshrc. To set them for a specific user only, add the commands to the .tcshrc file in the user's home directory. The environment variables are set each time the user logs in.

Setting environment variables for the message file

Before using the compiler, you must install the message catalogs and set the environment variables:

LANG Specifies the national language for message and help files.

NLSPATH
: Specifies the path name of the message and help files.

XL_CONFIG
: Specifies the name of an alternative configuration file (.cfg) for the xlc utility. Note: For the syntax of the configuration file, see the description for the -F flag option in "Flag options syntax" on page 591.

The LANG environment variable can be set to any of the locales provided on the system. See the description of locales in *z/OS XL C/C++ Programming Guide* for more information.

The national language code for United States English may be En_US or C. If the Japanese message catalog has been installed on your system, you can substitute Ja_JP for En_US.

To determine the current setting of the national language on your system, see the output from both of the following echo commands:
- **echo $LANG**
- **echo $NLSPATH**

The LANG and NLSPATH environment variables are initialized when the operating system is installed, and may differ from the ones you want to use.

Setting up a configuration file

The configuration file specifies information that the compiler uses when you invoke it. This file defines values used by the compiler to compile C or C++ programs. You can make entries to this file to support specific compilation requirements or to support other C or C++ compilation environments.

A configuration file is a UNIX file consisting of named sections called stanzas. Each stanza contains keywords called configuration file attributes, which are assigned values. The attributes are separated from their assigned value by an equal sign. A stanza can point to a default stanza by specifying the ″use″ keyword. This allows specifying common attributes in a default stanza and only the deltas in a specific stanza, referred to as the local stanza.

For any of the supported attributes not found in the configuration file, the xlc utility uses the built-in defaults. It uses the first occurrence in the configuration file of a stanza or attribute it is looking for. Unsupported attributes, and duplicate stanzas and attributes are not diagnosed.

Notes:
1. The difference between specifying values in the stanza and relying on the defaults provided by the xlc utility is that the defaults provided by the xlc utility will not override pragmas.
2. Any entry in the configuration file must occur on a single line. You cannot continue an entry over multiple lines.

Configuration file attributes

A stanza in the configuration file can contain the following attributes:

acceptable_rc
: Enables you to specify a number that represents a return code value for a program invoked by the xlc utility. The xlc utility does not

place any restriction on the value assigned to the `acceptable_rc` attribute. `acceptable_rc` can appear as part of any stanza in the configuration file.

Note: If the `acceptable_rc` attribute is not specified in the configuration file, the xlc utility will assign the value from a c89 *prefix*_ACCEPTABLE_RC environment variable, if it is exported, to the `acceptable_rc`, otherwise it will default to 4. The command name used to invoke the xlc utility determines the prefix that the xlc utility will use when looking for a *prefix*_ACCEPTABLE_RC environment variable. For example, if the xlc utility is invoked using the **xlC** command name, the xlc utility will look for _CXX_ACCEPTABLE_RC and, if found, use it. If the `acceptable_rc` attribute is specified in the configuration file, the xlc utility will use the value specified in the configuration file and will ignore an exported *prefix*_ACCEPTABLE_RC environment variable. For more information on the *prefix*_ACCEPTABLE_RC environment variable, see "Environment variables" on page 550.

as
Path name to be used for the assembler. The default is /bin/c89.

asopt
The list of options for the assembler and not for the compiler. These override all normal processing by the compiler and are directed to the assembler specified in the `as` attribute. Options are specified following the c89 utility syntax.

asuffix
The suffix for archive files. The default is `a`.

asuffix_host
The suffix for archive data sets. The default is LIB.

ccomp
The C compiler. The default is usr/lpp/cbclib/xlc/exe/ccndrvr.

cinc
A comma separated list of directories or data set wild cards used to search for C header files. The default for this attribute is: -I//'CEE.SCEEH.+'. For further information on the list of search places used by the compiler to search for system header files, see the note at the end of this list of configuration file attributes.

classversion
The USL class library version. The default matches the current release, as described in "TARGET" on page 240.

cppcomp
The C++ compiler. The default is /usr/lpp/cbclib/xlc/exe/ccndrvr.

cppinc
A comma separated list of directories or data set wild cards used to search for C++ header files. The default for this attribute is: -I//'CEE.SCEEH.+',-I//'CBC.SCLBH.+'.For further information on the list of search places used by the compiler to search for system header files, see the note at the end of this list of configuration file attributes.

csuffix
The suffix for source programs. The default is `c` (lowercase c).

csuffix_host
The suffix for C source data sets. The default is `C` (uppercase C).

cversion
The compiler version. The default matches the current release, as described in "TARGET" on page 240. The oldest release supported is z/OS V1R6.

cxxsuffix
The suffix for C++ source files. The default is `C` (uppercase C).

cxxsuffix_host
: The suffix for C++ source data sets. The default is CXX.

exportlist
: A colon separated list of data sets with member names indicating definition side-decks to be used to resolve symbols during the link-editing phase. This attribute is only used for compatibility with configuration files that are defined using the z/OS V1R6 release. Attributes with an appropriate suffix should be used instead (see descriptions for `exportlist` attributes with a suffix). The default for this attribute should match the type of stanza for which it is specified.

  Suffix-less C stanzas do not have a default.

  The default for suffix-less C++ stanzas is:

  ```
  CEE.SCEELIB(C128N):CBC.SCLBSID(IOSTREAM,COMPLEX)
  ```

  The default for C stanzas with an `_x` suffix is:

  ```
  CEE.SCEELIB(CELHS003,CELHS001)
  ```

  The default for C++ stanzas with an `_x` suffix is:

  ```
  CEE.SCEELIB(CELHS003,CELHSCPP,CELHS001,C128):CBC.SCLBSID
  (IOSTREAM,COMPLEX)
  ```

  The default for C stanzas with a `_64` suffix is:

  ```
  CEE.SCEELIB(CELQS003)
  ```

  The default for C++ stanzas with a `_64` suffix is:

  ```
  CEE.SCEELIB(CELQS003,CELQSCPP,C64):CBC.SCLBSID(IOSQ64)
  ```

exportlist_c
: A colon separated list of data sets with member names indicating definition side-decks to be used to resolve symbols during the link-editing phase of non-XPLINK C applications. The default for this attribute is NONE.

exportlist_cpp
: A colon separated list of data sets with member names indicating definition side-decks to be used to resolve symbols during the link-editing phase of non-XPLINK C++ applications. The default for this attribute is:

  ```
  CEE.SCEELIB(C128n):CBC.SCLBSID(IOSTREAM,COMPLEX)
  ```

exportlist_c_x
: A colon separated list of data sets with member names indicating definition side-decks to be used to resolve symbols during the link-editing phase of XPLINK C applications. The default for this attribute is:

  ```
  CEE.SCEELIB(CELHS003,CELHS001)
  ```

exportlist_cpp_x
: A colon separated list of data sets with member names indicating definition side-decks to be used to resolve symbols during the link-editing phase of XPLINK C++ applications. The default for this attribute is:

  ```
  CEE.SCEELIB(CELHS003,CELHSCPP,CELHS001,C128):CBC.SCLBSID
  (IOSTREAM,COMPLEX)
  ```

exportlist_c_64
: A colon separated list of data sets with member names indicating

definition side-decks to be used to resolve symbols during the link-editing phase of 64-bit C applications. The default for this attribute is:

`CEE.SCEELIB(CELQS003)`

exportlist_cpp_64
A colon separated list of data sets with member names indicating definition side-decks to be used to resolve symbols during the link-editing phase of 64-bit C++ applications. The default for this attribute is:

`CEE.SCEELIB(CELQS003,CELQSCPP,C64):CBC.SCLBSID(IOSQ64)`

isuffix The suffix for C preprocessed files. The default is `i`.

isuffix_host The suffix for C preprocessed data sets. The default is CEX.

ilsuffix The suffix for IPA output files. The default is `I`.

ilsuffix_host The suffix for IPA output data sets. The default is IPA.

ixxsuffix The suffix for C++ preprocessed files. The default is `i`.

ixxsuffix_host
The suffix for C++ preprocessed data sets. The default is CEX.

ld The path name to be used for the binder. The default is /bin/c89.

ld_c The path name to be used for the binder when only C sources appear on the command line invoked with a C stanza. The default is: /bin/c89.

ld_cpp The path name to be used for the binder when at least one C++ source appears on the command line, or when a C++ stanza is used. The default is: /bin/cxx.

libraries `libraries` specifies the default libraries that the binder is to use at bind time. The libraries are specified using the **-l***libname* syntax, with multiple library specifications separated by commas. The default is empty.

libraries2 `libraries2` specifies additional libraries that the binder is to use at bind time. The libraries are specified using the **-l***libname* syntax, with multiple library specifications separated by commas. The default is empty.

options A string of option flags, separated by commas, to be processed by the compiler as if they had been entered on the command line.

osuffix The suffix for object files. The default is `.o`.

osuffix_host The suffix for object data sets. The default is OBJ.

psuffix The suffix for prelinked files. The default is `p`.

psuffix_host The suffix for prelinked data sets. The default is CPOBJ.

pversion The run-time library version. The default matches the current release, as described in “TARGET” on page 240.

ssuffix The suffix for assembler files. The default is `.s`.

ssuffix_host The suffix for assembler data sets. The default is ASM.

steplib A colon separated list of data sets or keyword NONE used to set the STEPLIB environment variable. The default is NONE, which causes all programs to be loaded from LPA or linklist.

syslib
A colon separated list of data sets used to resolve run-time library references. Data sets from this list are used to construct the SYSLIB DD for the IPA Link and the binder invocation for non-XPLINK applications. For compatibility with configuration files defined using the z/OS V1R6 release, this attribute is also used with XPLINK applications as a fallback when the `syslib_x` attribute is not specified. When the `syslib_x` attribute is not specified, the default for this attribute should match the type of stanza for which it is specified. When the `syslib_x` attribute is specified, the default for this attribute matches the default for suffix-less stanzas.

The default for suffix-less stanzas is:

```
CEE.SCEELKEX:CEE.SCEELKED:CBC.SCCNOBJ:SYS1.CSSLIB
```

The default for stanzas with `_x` and `_64` suffixes is:

```
CEE.SCEEBND2:CBC.SCCNOBJ:SYS1.CSSLIB
```

syslib_x
A colon separated list of data sets used to resolve run-time library references. Data sets from this list are used to construct the SYSLIB DD for the IPA Link and the binder invocation when building XPLINK applications (31-bit and 64-bit).

The default for this attribute is:

```
 CEE.SCEEBND2:CBC.SCCNOBJ:SYS1.CSSLIB
```

sysobj
A colon separated list of data sets containing object files used to resolve run-time library references. Data sets from this list are used to construct the LIBRARY control statements and the SYSLIB DD for the IPA Link and the binder invocation. This attribute is ignored for XPLINK and 64-bit applications.

The default is:

```
CEE.SCEEOBJ:CEE.SCEECPP
```

use
Values for attributes are taken from the named stanza and from the local stanza. For single-valued attributes, values in the use stanza apply if no value is provided in the local, or default stanza. For comma-separated lists, the values from the use stanza are added to the values from the local stanza.

| **usuffix**
| The suffix for make dependency file names. The default make dependency file name suffix is ″.u″, but it is overwritten by the value assigned to this attribute.

| There is no host version of this attribute, because make dependency feature only applies to z/OS UNIX files.

xlC
The path name of the C++ compiler invocation command. The default is /usr/lpp/cbclib/xlc/bin/xlc.

xlCcopt
A string of option flags, separated by commas, to be processed when the **xlc** command is used for compiling a C file.

xsuffix
The suffix for definition side-deck files. The default is `x`.

xsuffix_host
The suffix for definition side-deck data sets. The default is EXP.

Note: When using the xlc utility to invoke the compiler, the compiler uses the following list of search places to search for system header files:

- If the **-qnosearch** option is not specified on the command line or in the configuration file:
  1. search places defined in the customizable defaults module (CCNEDFLT)
  2. followed by those specified on the command line using the -I flag option
  3. followed by those specified in the configuration file
- If the **-qnosearch** is specified only in the configuration file:
  1. search places specified on the command line using the -I flag option
  2. followed by those specified in the configuration file
- If the **-qnosearch** option is specified on the command line:
  1. search places specified on the command line following the last specified **-qnosearch** option
  2. followed by those specified in the configuration file

Tailoring a configuration file

The default configuration file is installed in /usr/lpp/cbclib/xlc/etc/xlc.cfg.

You can copy this file and make changes to the copy to support specific compilation requirements or to support other C or C++ compilation environments. The -F option is used to specify a configuration file other than the default. For example, to make **-qnoro** the default for the **xlC** compiler invocation command, add **-qnoro** to the **xlC** stanza in your copied version of the configuration file.

You can link the compiler invocation command to several different names. The name you specify when you invoke the compiler determines which stanza of the configuration file the compiler uses. You can add other stanzas to your copy of the configuration file to customize your own compilation environment.

Only one stanza, in addition to the one referenced by the ″use″ attribute, is processed for any one invocation of the xlc utility. By default, the stanza that matches the command name used to invoke the xlc utility is used, but it can be overridden using the -F flag option as described in the example below.

Example: You can use the -F option with the compiler invocation command to make links to select additional stanzas or to specify a stanza or another configuration file:

```
xlC myfile.C -Fmyconfig:SPECIAL
```

would compile `myfile.C` using the `SPECIAL` stanza in a `myconfig` configuration file that you had created.

Default configuration file

The default configuration file, (/usr/lpp/cbclib/xlc/etc/xlc.cfg.), specifies information that the compiler uses when you invoke it. This file defines values used by the compiler to compile C or C++ programs. You can make entries to this file to support specific compilation requirements or to support other C or C++ compilation environments. Options specified in the configuration file override the default settings of the option. Similarly, options specified in the configuration file are in turn overridden by options set in the source file and on the command line. Options that do not follow this scheme are listed in “Specifying compiler options” on page 596.

Example: The following example shows a default configuration file:

```
*
* FUNCTION: z/OS 1.11 XL C/C++ Compiler Configuration file
*
* Licensed Materials - Property of IBM
* 5694-A01 Copyright IBM Corp. 2004, 2009
* All Rights Reserved
* US Government Users Restricted Rights - Use, duplication or
* disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
*
* C compiler, extended mode
xlc:      use              = DEFLT

* XPLINK C compiler, extended mode
xlc_x:    use              = DEFLT

* 64 bit C compiler, extended mode
xlc_64:   use              = DEFLT

* C compiler, common usage C
cc:       use              = DEFLT

* XPLINK C compiler, common usage C
cc_x:     use              = DEFLT

* 64 bit C compiler, common usage C
cc_64:    use              = DEFLT

* Strict ANSI C 89 compiler
c89:      use              = DEFLT

* XPLINK Strict ANSI C 89 compiler
c89_x:    use              = DEFLT

* 64 bit Strict ANSI C 89 compiler
c89_64:   use              = DEFLT

* ISO/IEC 9899:1999 Standard Compliant C Compiler
c99:      use              = DEFLT

* XPLINK ISO/IEC 9899:1999 Standard Compliant C Compiler
c99_x:    use              = DEFLT

* 64 bit ISO/IEC 9899:1999 Standard Compliant C Compiler
c99_64:   use              = DEFLT

* ANSI C++ compiler
cxx:      use              = DEFLT
          xlC              = /usr/lpp/cbclib/xlc/bin/.orig/xlC
          ipa              = /bin/cxx

* XPLINK ANSI C++ compiler
cxx_x:    use              = DEFLT
          xlC              = /usr/lpp/cbclib/xlc/bin/.orig/xlC
          ipa              = /bin/cxx

* 64 bit ANSI C++ compiler
cxx_64:   use              = DEFLT
          xlC              = /usr/lpp/cbclib/xlc/bin/.orig/xlC
          ipa              = /bin/cxx

* ANSI C++ compiler
c++:      use              = DEFLT
```

```
          xlC              = /usr/lpp/cbclib/xlc/bin/.orig/xlC
          ipa              = /bin/cxx

* XPLINK ANSI C++ compiler
c++_x:    use              = DEFLT
          xlC              = /usr/lpp/cbclib/xlc/bin/.orig/xlC
          ipa              = /bin/cxx

* 64 bit ANSI C++ compiler
c++_64:   use              = DEFLT
          xlC              = /usr/lpp/cbclib/xlc/bin/.orig/xlC
          ipa              = /bin/cxx

* C++ compiler, extended mode
xlC:      use              = DEFLT
          xlC              = /usr/lpp/cbclib/xlc/bin/.orig/xlC
          ipa              = /bin/cxx

* XPLINK C++ compiler, extended mode
xlC_x:    use              = DEFLT
          xlC              = /usr/lpp/cbclib/xlc/bin/.orig/xlC
          ipa              = /bin/cxx

* 64 bit C++ compiler, extended mode
xlC_64:   use              = DEFLT
          xlC              = /usr/lpp/cbclib/xlc/bin/.orig/xlC
          ipa              = /bin/cxx

* C++ compiler, extended mode
xlc++:    use              = DEFLT
          xlC              = /usr/lpp/cbclib/xlc/bin/.orig/xlC
          ipa              = /bin/cxx

* XPLINK C++ compiler, extended mode
xlc++_x:  use              = DEFLT
          xlC              = /usr/lpp/cbclib/xlc/bin/.orig/xlC
          ipa              = /bin/cxx

* 64 bit C++ compiler, extended mode
xlc++_64: use              = DEFLT
          xlC              = /usr/lpp/cbclib/xlc/bin/.orig/xlC
          ipa              = /bin/cxx

* common definitions
DEFLT:    cppcomp          = /usr/lpp/cbclib/xlc/exe/ccndrvr
          ccomp            = /usr/lpp/cbclib/xlc/exe/ccndrvr
          ipacomp          = /usr/lpp/cbclib/xlc/exe/ccndrvr
          ipa              = /bin/c89
          as               = /bin/c89
          ld_c             = /bin/c89
          ld_cpp           = /bin/cxx
          xlC              = /usr/lpp/cbclib/xlc/bin/xlc
          xlCcopt          = -D_XOPEN_SOURCE
          sysobj           = cee.sceeobj:cee.sceecpp
          syslib           = cee.sceelkex:cee.sceelked:cbc.sccnobj:sys1.csslib
          syslib_x         = cee.sceebnd2:cbc.sccnobj:sys1.csslib
          exportlist_c     = NONE
          exportlist_cpp   = cee.sceelib(c128n):cbc.sclbsid(iostream,complex)
          exportlist_c_x   = cee.sceelib(celhs003,celhs001)
          exportlist_cpp_x = cee.sceelib(celhs003,celhs001,celhscpp,c128):
cbc.sclbsid(iostream,complex)
          exportlist_c_64   = cee.sceelib(celqs003)
          exportlist_cpp_64 = cee.sceelib(celqs003,celqscpp,c64):cbc.sclbsid(iosx64)
          steplib          = NONE
```

Invoking the compiler

The z/OS XL C/C++ compiler is invoked using the following syntax, where *invocation* can be replaced with any valid z/OS XL C/C++ invocation command:

```
                  ┌──────────────────────────────────────────┐
                  ▼                                          │
►►──invocation────┴─┬──────────────────────┬─┬─────────────┬─┴────────────►◄
                    └─command_line_options─┘ └─input_files─┘
```

The parameters of the compiler invocation command can be names of input files, compiler options, and linkage-editor options. Compiler options perform a wide variety of functions such as setting compiler characteristics, describing object code and compiler output to be produced, and performing some preprocessor functions.

To compile without binding, use the **-c** compiler option. The **-c** option stops the compiler after compilation is completed and produces as output, an object file `file_name.o` for each `file_name.c` input source file, unless the **-o** option was used to specify a different object filename. The binder is not invoked. You can bind the object files later using the invocation command, specifying the object files without the **-c** option.

Notes:

1. Any object files produced from an earlier compilation with the same name as expected object files in this compilation are deleted as part of the compilation process, even if new object files are not produced.
2. By default, the invocation command calls both the compiler and the binder. It passes binder options to the binder. Consequently, the invocation commands also accept all binder options.

Invoking the binder

All invocation commands invoke the binder using the c89 utility, so all binder options must follow the syntax supported by the c89 utility. Standard libraries required to bind your program are controlled by the `sysobj`, `syslib`, and `exportlist` attributes in the configuration file.

The specified object files are processed by the binder to create one executable file. Invoking the compiler with one of the invocation commands, automatically calls the binder unless you specify one of the following compiler options: -E, **-c**, -P, **-qsyntaxonly**, **-qpponly**, or **-#**.

All input and output files supported by the c89 utility are valid for all invocation commands.

Supported options

In addition to -W syntax for specifying keyword options, the xlc utility supports AIX **-q** options syntax and several new flag options.

–q options syntax

The following principles apply to the use of z/OS option names with **-q** syntax:

- Any valid abbreviation of a z/OS option name that matches (in full or in part) the spelling of the corresponding option on AIX, can be specified using **-q** syntax. For example, ATTRIBUTE can be specified as **-qatt**, **-qattr**, **-qattri**, **-qattrib**, **-qattribu**, **-qattribut**, and **-qattribute**. This is true even if the AIX option name is longer, as in the case of **-qbitfields**, which can be specified as **-qbitf**, **-qbitfi**, **-qbitfie**, **-qbitfiel**, **-qbitfield**, and **-qbitfields**. This is the common case that applies to most z/OS options.

- Any z/OS-specific option name and its valid abbreviation can also be specified using **-q** syntax; for example, DBRMLIB.
- Any z/OS option name that has a different spelling from the corresponding AIX option name can not be specified using **-q** syntax. For example, CHECKOUT, EXH, ILP32, LP64, SSCOMM, and TEST can not be specified using **-q** syntax. Instead use, **-qinfo**, **-qeh**, **-q32**, **-q64**, **-qcpluscmt**, and **-qdebug=format=isd**. For historical reasons, OBJECTMODEL and PHASEID are exceptions to this principle, as both can be specified using **-q** syntax. However, **-qobjmodel** and **-qphsinfo** should be used instead to enhance portability with AIX.

Options that do not exist on AIX, and are not required to accomplish a z/OS-specific task, and their effect can be accomplished by other means, are not supported with **-q** syntax. For example, use -D instead of DEFINE, -U instead of UNDEFINE, and **-co** instead of OBJECT.

Suboptions with negative forms of **-q** options are not supported, unless they cause an active compiler action, as in the case of **-qnokeyword=<keyword>**.

Compiler options for AIX that do not apply to z/OS are accepted and ignored with a diagnostic message. For a brief description of the compiler options that can be specified with xlc, type **xlc** or any other supported command name. For detailed descriptions of the compiler options that can be specified with xlc, refer to Chapter 4, “Compiler Options,” on page 41.

The following syntax diagram shows how to specify keyword options using **-q** syntax:

```
►►──-q──option_keyword──┬─────────────────────┬──────────────────────►◄
                        │    ┌─:──────────┐   │
                        │    ▼            │   │
                        └─=────suboption──┴───┘
```

In the diagram, *option_keyword* is an option name and the optional *suboption* is a value associated with the option. Keyword options with no suboptions represent switches that may be either on or off. The *option_keyword* by itself turns the switch on, and the *option_keyword* preceded by the letters NO turns the switch off. For example, **-qLIST** tells the compiler to produce a listing and **-qNOLIST** tells the compiler not to produce a listing. If an option that represents a switch is set more than once, the compiler uses the last setting.

Some keyword options only have values. Keywords which have values are specified as `keyword=value` pairs.

Example:

```
-qfloat=ieee
```

where `ieee` is a value.

Some keyword options have suboptions, which in turn have values. Suboptions which have values are specified as `suboption=value` pairs.

Example:

```
-qipa=level=2
```

where `level` is a suboption and `2` is a value.

Keyword options and suboptions may appear in mixed case letters in the command that invokes the xlc utility. Keyword options that have suboptions can also be preceded by the letters NO in which case they are similar to off switches and do not allow suboptions. This is a noticeable departure from the z/OS options, which allow suboptions even if they are preceded by the letters NO. However, the function that the z/OS behavior provides can easily be emulated by specifying all desired suboptions with an *option_keyword* followed by the same *option_keyword* that is preceded by the letters NO. The subsequent specification of the same *option_keyword* unlocks all previously specified suboptions.

Example: NODEBUG(FORMAT(DWARF)) is equivalent to **-qdebug=format=dwarf -qnodebug**

The compiler recognizes all AIX **-q** options, but only those that have a matching z/OS native option are accepted and processed. All other AIX **-q** options are ignored with an informational message.

Note: The GENASM compiler option is not supported with **-q** syntax. Use the -S flag option instead, which is described in "Flag options syntax."

Flag options syntax

Except for the -W, -D, and -U flag options, all flag options that are supported by the c89 utility are supported by the xlc utility with the same semantics as documented in Chapter 21, "c89 — Compiler invocation using host environment variables," on page 535. The xlc utility does not recognize constructs such as **-Wl,I** or **-Wl,p**. All other aspects of the -W flag are the same as with the c89 utility. -D and -U flag options are not preprocessed by the xlc utility. Instead, they are converted to the DEFINE and UNDEFINE native options and are passed to the compiler. The xlc utility also supports several additional flag options, which are described below:

-#
Displays language processing commands but does not invoke them; output goes to stdout.

```
►►── -#──────────────────────────────────────────────────────►◄
```

-B
Determines substitute path names for programs such as the assembler and binder, where program can be:

- `a` (assembler)
- `c` (z/OS XL C/C++ compiler)
- `l` (binder)
- `L` (IPA Link)

```
►►──-B──┬────────┬──┬──────────────┬───────────────────────►◄
        └─prefix─┘  └─ -t──program─┘
```

Notes:

1. The optional prefix defines part of a path name to the new programs. The compiler does not add a / between the prefix and the program name.
2. To form the complete path name for each program, the xlc utility adds prefix to the program names indicated by the **-t** option. The program names can be any combination of z/OS XL C/C++ compiler, assembler, IPA Link and binder.
3. If **-Bprefix** is not specified, or if -B is specified without the prefix, the default path (/usr/lpp/cbclib/xlc/bin/) is used.
4. **-tprograms** specifies the programs for which the path name indicated by the -B option is to be applied.
5. **-Bprefix** and **-tprograms** options override the path names of the programs that are specified inside the configuration file indicated by the **-Fconfig_file** option.

   Example: To compile `myprogram.c` using a substitute compiler and binder from `/lib/tmp/mine/`, enter:

   ```
   xlc myprogram.c -B/lib/tmp/mine/
   ```

   Example: To compile `myprogram.c` using a substitute binder from `/lib/tmp/mine/`, enter:

   ```
   xlc myprogram.c -B/lib/tmp/mine/ -tl
   ```

-F

Names an alternative configuration file (.cfg) for the xlc utility.

Suboptions are:

- *config_file* (specifies the name of an xlc configuration file.)
- *stanza* (specifies the name of the command used to invoke the compiler. This directs the compiler to use the entries under *stanza* in the *config_file* to set up the compiler environment.)

```
►►──-F──┬─config_file──┬──────────┬─┬──────────────────────────►◄
        │              └─:──stanza─┘ │
        └─:──stanza──────────────────┘
```

Notes:

1. The default configuration file supplied at installation time is called /usr/lpp/cbclib/xlc/etc/xlc.cfg. Any file names or stanzas that you specify on the command line override the defaults specified in the /usr/lpp/cbclib/xlc/etc/xlc.cfg configuration file.
2. The -B, **-t**, and -W options override entries in the configuration file indicated by the -F option.

Example: To compile `myprogram.c` using a configuration file called `/usr/tmp/mycbc.cfg`, enter:

```
xlc myprogram.c -F/usr/tmp/mycbc.cfg
```

-M

Instructs the compiler to generate a dependency file or dependency files that can be used by the make utility. By default, the dependency file name and path is the same as the -o compile option but with `.u` suffix. Dependency file name can be overridden by the **-MF** option. The default suffix can be modified through ″usuffix″ configuration file attribute.

The compiler will generate as many dependency files as there are source files specified.

-M is the equivalent of specifying **-qmakedep**.

```
►►── -M──────────────────────────────────────────────────────────►◄
```

Example: To compile `myprogram.c` and create an output file named `myprogram.u`, enter:

```
xlc -c -M myprogram.c
```

Example: The following is a simple makefile that uses **-M** feature. You can refer to *z/OS UNIX System Services Programming Tools* for more information about the make utility and makefiles.

```
CFLAGS = -M -qLSE=lib1 -qfloat=ieee

all: program

# Compile recipe;  will also regenerate dependencies, used on the
next compile %.o: %.c
 xlc -c $(CFLAGS) $^

program: file1.o file2.o
 xlc $(CFLAGS) -o program $&

#This line will include the generated file1.u and/or file2.u only
if they exist .INCLUDE .IGNORE : file1.u file2.u
```

-MF If **-M** or **-qmakedep** is specified, this option is used to override the file path name of the dependency file.

```
►►── -q──makedep──┬──────────┬──────────────────────────────►◄
                  └─-MF──file─┘
```

In the syntax, *file* can be either a file name or a directory. If a directory is specified, the default dependency file name is used (as per the **-M** option) and placed in this directory. The directory must exist. If a relative file name is specified, it is relative to the current working directory.

Notes:

1. The argument of *file* can not be the name of a dataset.
2. If the file specified by **-MF** already exists, it will be overwritten. Moreover, if the output path specified does not exist or is write-protected, an error message will be issued.
3. If you specify a single file name for the **-MF** option when compiling multiple source files, each generated dependency file overwrites the previous one. Only a single output file will be generated for the last source file specified on the command line.

Example: You can refer to the following table for detail usage of **-M** and **-MF**:

Table 49. Example of using -M and -MF

Description	Command	Dependency File
-MF is not specified	`xlc -c -M t.c`	./t.u is generated
	`xlc -M -c -o obj.o t.c`	./obj.u is generated
	`xlc -c -M -o dir/ t.c`	./dir/t.u is generated if ./dir is writable
-MF specifies a file	`xlc -c -qmakedep -MF dep.u t.c`	./dep.u is generated
	`xlc -c -o obj.o -M -MF ../dep.x t.c`	../dep.x is generated
	`xlc -c -M -MF dir/dep.d a.c b.c`	./dir/dep.d is generated for b.c **only**.
-MF specifies a directory	`xlc -c -M -MF dir/ a.c b.c`	./dir/a.u and ./dir/b.u are generated for a.c and b.c respectively if ./dir/ is writable

-O
Optimizes generated code.

```
►►──-O──────────────────────────────────────────────────────►◄
```

-O2
Same as -O.

```
►►──-O2─────────────────────────────────────────────────────►◄
```

-O3
Performs memory and compile-time intensive optimizations in addition to those executed with -O2. The -O3 specific optimizations have the potential to alter the semantics of a user's program. The compiler guards against these optimizations at -O2 and the option **-qstrict** is provided at -O3 to turn off these aggressive optimizations.

```
►►──-O3─────────────────────────────────────────────────────►◄
```

-O4
Equivalent to -O3 **-qipa** and **-qhot**.

```
►►──-O4─────────────────────────────────────────────────────►◄
```

-O5
Equivalent to -O3 **-qipa=level=2** and **-qhot**.

```
►►──-O5─────────────────────────────────────────────────────►◄
```

-P
Produces preprocessed output in a file that has a suffix that is defined by `isuffix`, `isuffix_host`, `ixxsuffix`, and `ixxsuffix_host`. The default for host files is `.CEX` and for z/OS UNIX files is `.i`.

As with the -E option, the -C option can be combined with the -P option to preserve the comments.

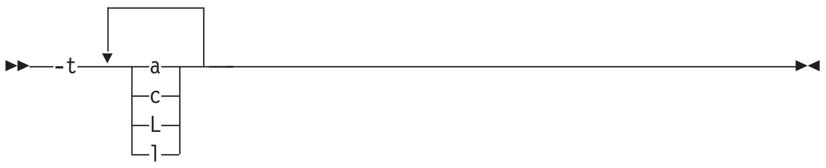
-S
Produces an assembler source file for C source that is compiled with the METAL compiler option. The **-o** option can be used to override the default file name produced by -S. The default file name

is the C source file name with the suffix determined by the `ssuffix` and `ssuffix_host` attributes in the configuration file.

-t

Adds the prefix specified by the -B option to the designated programs, where programs are:

- `a` (assembler)
- `c` (z/OS XL C/C++ compiler)
- `L` (Interprocedural Analysis tool - link phase)
- `l` (binder)



Note: This option must be used together with the -B option.

If -B is specified but the prefix is not, the default prefix is /usr/lpp/cbclib/xlc/bin/. If **-Bprefix** is not specified at all, the prefix of the standard program names is /usr/lib/cbclib/xlc/bin/.

If -B is specified but **-tprograms** is not, the default is to construct path names for all of the standard program names: `a`, `c`, `L`, and `l`.

Example: To compile `myprogram.c` so that the name `/u/new/compilers/` is prefixed to the binder and assembler program names, enter:

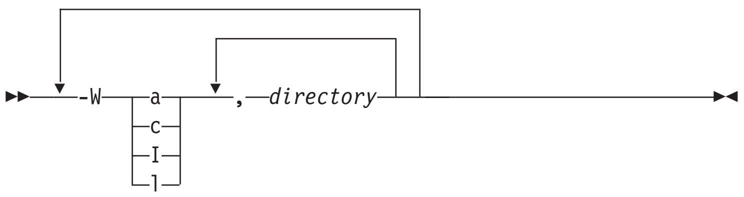
```
xlc myprogram.c -B/u/new/compilers/ -tla
```

-W

Passes the listed options to a designated compiler program where programs are:

- `a` (assembler)
- `c` (z/OS XL C/C++ compiler)
- `I` (Interprocedural Analysis tool - compile phase)
- `l` (binder)

Note: When used in the configuration file, the -W option requires the escape sequence back slash comma (\,) to represent a comma in the parameter string.



Example: To compile myprogram.s so that the option **map** is passed to the binder and the option list is passed to the assembler, enter:

```
xlc myprogram.s -Wl,map -Wa,list
```

Example: In a configuration file, use the `\,` sequence to represent the comma (,):

```
-Wl\,map,-Wa\,list
```

Specifying compiler options

Compiler options perform a wide variety of functions, such as setting compiler characteristics, describing the object code and compiler output to be produced, and performing some preprocessor functions. You can specify compiler options in one or more of the following ways:

- On the command line
- In your source program
- In a configuration file

The compiler uses default settings for the compiler options not explicitly set by you in these listed ways. The defaults can be compiler defaults, installation defaults, or the defaults set by the c89 or the xlc utility. The compiler defaults are overridden by installation defaults, which are overridden by the defaults set by the c89 or the xlc utilities.

When specifying compiler options, it is possible for option conflicts and incompatibilities to occur. z/OS XL C/C++ resolves most of these conflicts and incompatibilities in a consistent fashion, as follows:

```
Source  overrides    Command   overrides   Configuration    overrides   Default
 file  ----------->   line    ---------->      file        -----------> settings
```

Options that do not follow this scheme are summarized in the following table:

Table 50. Compiler option conflict resolution

Option	Conflicting Options	Resolution
-qxref	-qxref=FULL	-qxref=FULL
-qattr	-qattr=FULL	-qattr=FULL
-E	-o	-E
-#	-v	-#
-F	-B | -t |-W | -qpath| *configuration file settings*	-B| -t | -W |-qpath
-qpath	-B| -t	-qpath overrides -B and -t

In general, if more than one variation of the same option is specified (with the exception of xref and attr), the compiler uses the setting of the last one specified. Compiler options specified on the command line must appear in the order you want the compiler to process them.

If a command-line flag is valid for more than one compiler program (for example -B, -W, or -I applied to the compiler, binder, and assembler program names), you must specify it in `options`, or `asopt` in the configuration file. The command-line flags must appear in the order that they are to be directed to the appropriate compiler program.

Three exceptions to the rules of conflicting options are the -I*directory* or `-I//dataset_name`, **-l***library*, and -L*directory* options, which have cumulative effects when they are specified more than once.

Specifying compiler options on the command line

There are two kinds of command-line options:

- **-qoption_keyword** (compiler-specific)
- Flag options (available to z/OS XL C/C++ compilers in z/OS UNIX System Service environment)

Command-line options in the **-q** *option_keyword* format are similar to on and off switches. For most **-q** options, if a given option is specified more than once, the last appearance of that option on the command line is the one recognized by the compiler. For example, **qsource** turns on the source option to produce a compiler listing, and **-qnosource** turns off the source option so that no source listing is produced.

Example: The following example would produce a source listing for both `MyNewProg.C` and `MyFirstProg.C` because the last source option specified (**-qsource**) takes precedence:

```
 xlC -qnosource MyFirstProg.C -qsource MyNewProg.C
```

You can have multiple **-q** *option_keyword* instances in the same command line, but they must be separated by blanks. Option keywords can appear in mixed case, but you must specify the **-q** in lowercase.

Example: You can specify any **-q** *option_keyword* before or after the file name:

```
xlC -qLIST -qnomaf file.c
xlC file.c -qxref -qsource
```

Some options have suboptions. You specify these with an equal sign following the **-qoption**. If the option permits more than one suboption, a colon (:) must separate each suboption from the next.

Example: The following example compiles the C source file file.c using the option **-qipa** to specify the inter procedural analysis options. The suboption `level=2` tells the compiler to use the full inter procedural data flow and alias analysis, **map** tells the compiler to produce a report, and the **noobj** tells the compiler to produce only an IPA object without a regular object. The option **-qattr** with suboption `full` will produce an attribute listing of all identifiers in the program.

```
xlc -qipa=level=2:map:noobj -qattr=full file.c
```

Specifying flag options

The z/OS XL C/C++ compilers use a number of common conventional flag options. Lowercase flags are different from their corresponding uppercase flags. For example, **-c** and -C are two different compiler options:

- **-c** specifies that the compiler should only preprocess, compile, and not invoke the binder
- -C can be used with -E or -P to specify that user comments should be preserved

Some flag options have arguments that form part of the flag.

Example:

```
 xlC stem.c -F/home/tools/test3/new.cfg:myc -qflag=w
```

where `new.cfg` is a custom configuration file.

You can specify flags that do not take arguments in one string.

Example:

```
xlc -Ocv file.c
```

has the same effect as:

```
 xlc -O -v -c test.c
```

Specifying compiler options in a configuration file

The default configuration file, (/usr/lpp/cbclib/xlc/etc/xlc.cfg), specifies information that the compiler uses when you invoke it. This file defines values used by the compiler to compile C or C++ programs. You can make entries to this file to support specific compilation requirements or to support other C or C++ compilation environments.

Options specified in the configuration file override the default settings of the option. Similarly, options specified in the configuration file are in turn overridden by options set in the source file and on the command line.

Specifying compiler options in your program source files

You can specify compiler options within your program source by using `#pragma` directives. Options specified with pragma directives in program source files override all other option settings.

Specifying compiler options for architecture-specific 32-bit or 64-bit compilation

You can use z/OS XL C/C++ compiler options to optimize compiler output for use on specific processor architectures. You can also instruct the compiler to compile in either 32-bit or 64-bit mode.

The compiler evaluates compiler options in the following order, with the last allowable one found determining the compiler mode:

1. Compiler default (32-bit mode)
2. Configuration file settings
3. Command line compiler options (**-q32**, **-q64**, **-qarch**, **-qtune**)
4. Source file statements (**#pragma options(ARCH(suboption),TUNE(suboption))**)

The compilation mode actually used by the compiler depends on a combination of the settings of the **-q32**, **-q64**, **-qarch**, and **-qtune** compiler options, subject to the following conditions:

- Compiler mode is set according to the last-found instance of the **-q32**, or **-q64** compiler options. If neither of these compiler options is chosen, the compiler mode is set to 32-bit.
- Architecture target is set according to the last-found instance of the **-qarch** compiler option, provided that the specified **-qarch** setting is compatible with the compiler mode setting. If the **-qarch** option is not set, the compiler assumes a **-qarch** setting of `5`.
- Tuning of the architecture target is set according to the last-found instance of the **-qtune** compiler option, provided that the **-qtune** setting is compatible with the architecture target and compiler mode settings. If the **-qtune** option is not set, the compiler assumes a default **-qtune** setting according to the **-qarch** setting in use.

Possible option conflicts and compiler resolution of these conflicts are described below:

- **-q32** or **-q64** setting is incompatible with user-selected **-qarch** option.

  Resolution: **-q32** or **-q64** setting overrides **-qarch** option; compiler issues a warning message, sets **-qarch** to `5`, and sets the **-qtune** option to the **-qarch** setting's default **-qtune** value.
- **-q32** or **-q64** setting is incompatible with user-selected **-qtune** option.

  Resolution: **-q32** or **-q64** setting overrides **-qtune** option; compiler issues a warning message, and sets **-qtune** to the **-qarch** settings's default **-qtune** value.
- **-qarch** option is incompatible with user-selected **-qtune** option.

  Resolution: Compiler issues a warning message, and sets **-qtune** to the **-qarch** setting's default **-qtune** value.
- Selected **-qarch** and **-qtune** options are not known to the compiler.

  Resolution: Compiler issues a warning message, sets **-qarch** to `5`, and sets **-qtune** to the **-qarch** setting's default **-qtune** setting. The compiler mode (32 or 64-bit) is determined by the **-q32** or **-q64** compiler settings.

Appendix A. Prelinking and linking z/OS XL C/C++ programs

Instead of using the prelinker and linkage editor, you can use the binder. See Chapter 9, “Binding z/OS XL C/C++ programs,” on page 401 for more information.

This information shows how to prelink and link your z/OS XL C/C++ programs. The Language Environment prelinker combines the object modules that comprise a C or C++ application into a single object module. The linkage editor then processes this object module and generates a load module that can be retrieved for execution.

You do not need to prelink object modules that:
- Do not refer to writable static
- Do not contain long names
- Do not contain DLL code

You must use the Language Environment prelinker before linking your application when any of the following are true:
- Your application contains C++ code.
- Your application contains C code that is compiled with the RENT, LONGNAME, DLL, or IPA compiler options.
- Your application is compiled to run under z/OS UNIX System Services.

If you do not need to prelink your application, continue to the information in “Linking an application” on page 606. For information on creating object libraries in z/OS XL C++, refer to Chapter 13, “Object library utility,” on page 481. For information on prelinking and linking object modules under z/OS UNIX System Services, refer to “Prelinking and link-editing under the z/OS Shell” on page 632.

Note: When you use the prelinker to prelink C++ object modules, you may get duplicate symbol warnings due to virtual function symbols generated by the compiler. You can ignore these symbols and warnings. You will not get these messages if you use the binder.

Restrictions on using the prelinker

You cannot use the prelinker if you specified either the XPLINK or GOFF compiler option when you compiled.

Note: The prelinker cannot be used for 64-bit compiled object modules, therefore you cannot use the prelinker if any of the object modules were compiled using the LP64 option.

Prelinking an application

To prelink multiple object modules and then link with a load module, you must run the multiple object modules through the prelinker and add the load module in the link step (for example, when prelinking and linking a CICS program).

You must prelink together all components that require prelinking prior to linking. For example, `LINK(PRELINK(XOBJ1,XOBJ2))` and `LINK(PRELINK(XOBJ1,XOBJ2),OBJ3)` are valid but `LINK(PRELINK(XOBJ1), PRELINK(XOBJ2))` is not. The prelinker only handles a subset of what the linker handles, in particular, it does not understand load modules (or program objects).

© Copyright IBM Corp. 1998, 2009

For object modules with writable static references:

- The prelinker combines writable static initialization information
- The prelinker assigns relative offsets to objects in writable static storage
- The prelinker removes writable static name and relocation information

For object modules that contain long names, the prelinker maps long names to short names on output. Long names are mixed-case external names of up to 1024 characters. Short names are eight character, uppercase external names.

For object modules that contain DLL code (C++ code, or C code that was compiled with the DLL compiler option), the prelinker does the following:

- It generates a function descriptor (linkage section) in writable static for each DLL referenced function
- It generates a variable descriptor (linkage section) for each unresolved DLL referenced variable
- It generates an IMPORT control statement in the SYSDEFSD data set for each exported function and variable
- It generates internal information for the load module that describes which symbols are exported and which symbols are imported from other load modules
- It combines static DLL initialization information

Language Environment Library functions are not included as part of automatic library calls. This omission can result in warning messages about unresolved references to C library functions or C library objects. These unresolved C library functions or objects will be resolved in a following link-edit step.

For C or C++ object modules from applications that were compiled with the DLL compiler option, the prelinker uses long names to resolve exported and imported symbols. For information on how to create a DLL or an application that uses DLLs, see *z/OS XL C/C++ Programming Guide*.

Using DD Statements for the standard data sets - prelinker

The prelinker always requires three standard data sets. You must define these data sets in DD statements with the ddnames SYSIN, SYSMOD, and SYSMSGS.

You may need five other data sets that are defined by DD statements with the names STEPLIB, SYSLIB, SYSDEFSD, SYSOUT, and SYSPRINT. For a list of the data sets and their usage see Table 51. For details on the attributes of specific data sets see “Description of data sets used” on page 470.

Table 51. Data sets used for prelinking

ddname	Type	Function
SYSIN	Input	Primary input data, usually the output of the compiler
SYSMSGS	Input	Location of prelinker message file
STEPLIB[2]	Utility Library	Location of prelinker and Language Environment run-time data sets
SYSLIB	Library	Secondary input
SYSDEFSD[1]	Output	Definition side-deck
SYSOUT	Output	Prelinker Map
SYSMOD	Output	Output data set for the prelinked object module

Table 51. Data sets used for prelinking (continued)

ddname	Type	Function
SYSPRINT	Output	Destination of error messages generated by the prelinker
User-specified	Input	Obtain additional object modules and load modules

Notes:

1 Required output from the prelinker if you are exporting variables or functions.

2 Optional data sets, if the compiler and run-time library are installed in the LPA or ELPA. To save resources and improve compile time, especially in z/OS UNIX System Services, do not unnecessarily specify data sets on the STEPLIB DD name.

Primary input (SYSIN)

Primary input to the prelinker consists of a sequential data set, a member of a partitioned data set, or an inline object module. The primary input must consist of one or more separately compiled object modules or prelinker control statements. (See "INCLUDE control statement" on page 635.)

If the primary input to the prelinker consists of a mix of object modules and include control statements, include control statements must be placed last (or after all object modules).

If you are prelinking an application that imports symbols from a DLL, you must include the definition side-deck for that DLL in SYSIN. The prelinker uses the definition side-deck to resolve external symbols for functions and variables that are imported by your application. If you call more than one DLL, you need to include a definition side-deck for each.

Prelinker message file (SYSMSGS)

With this DD statement name, you provide the prelinker with the information it needs to generate error messages and the Prelinker Map.

Prelinker and Language Environment library (STEPLIB)

To prelink your program, the system must be able to locate the data sets that contain the prelinker and Language Environment run-time library. The DD statement with the name STEPLIB points to these data sets. If the run-time library is installed in the LPA or ELPA, it is found automatically. Otherwise, SCEERUN and SCEERUN2 must be in the JOBLIB or STEPLIB. For information on the search order, see Chapter 11, "Running a C or C++ application," on page 455.

Secondary input (SYSLIB)

Secondary input to the prelinker consists of object modules that are not part of the primary input data set, but are to be included in the output prelinked object module from the automatic call library. The *automatic call library* contains object modules that will be used as secondary input to the prelinker to resolve external symbols left undefined after all the primary input has been processed. Concatenate multiple object module libraries by using the DD statement with the name SYSLIB. For more information on concatenating data sets, see "Specifying include files" on page 345.

Note: SYSLIB data sets that are used as input to the prelinker must be cataloged.

Definition side-deck (SYSDEFSD)

The prelinker generates a definition side-deck if you are prelinking an application that exports external symbols for functions and variables (a DLL). You must provide this side-deck to any user of your DLL. The users of the DLL must prelink the side-deck of the DLL with their other object modules. The definition side-deck (SYSDEFSD) is not generated if the NODYNAM option is in effect.

Listing (SYSOUT)

If you specify the MAP prelinker option, the prelinker writes a map to the SYSOUT data set. This map provides you with warnings, files that are included in input to the prelinker, and names of external symbols.

Output (SYSMOD)

The prelinker produces a single prelinked object module, and stores it in the SYSMOD data set. The linkage editor uses this data set as input.

Prelinker error messages (SYSPRINT)

If the prelinker encounters problems in its attempt to prelink your program, it generates error messages and places them in the SYSPRINT data set.

Input to the prelinker

Input to the prelinker can be:

- One or more object modules (not previously prelinked)
- Prelinker control statements (INCLUDE, LIBRARY ...)
- Object module libraries

The process of resolving or including input from these sources depends on the type of the source and the current input and prelink options.

Unresolved references or undefined writable static objects often result if you give the prelinker input object modules produced with a mixture of inconsistent compiler options (for example, RENT | NORENT, LONGNAME | NOLONGNAME, or DLL options). These options may expose symbol names in different ways in your object file, so that the prelinker may be unable to find the matching definition of a referenced symbol if the definition and the reference are exposed differently.

Primary input

Primary input to the prelinker consists of a sequential data set (file) that contains one or more separately compiled object modules, possibly with prelinker control statements. Specify the primary input data set through the SYSIN ddname.

Secondary input

Secondary input to the prelinker consists of object modules that are not part of the primary input data set but are to be included as a result of processing of primary input. Object modules that are brought in because of INCLUDE control statements are secondary input. Object modules brought in as a result of automatic call library (library search) processing of currently unresolved symbols through a LIBRARY control statement or through SYSLIB are also secondary input.

An automatic call library may be in the form of:

- PDS Libraries that contain object modules
- PDSE Libraries that contain object modules
- Archive Libraries that contain object modules (if you used OMVS prelinker option)

Prelinker output

Writable static references that are not resolved by the prelinker cannot be resolved later. Only the prelinker can be used to resolve writable static. The output object module of the prelinker should not be used as input to another prelink.

Prelinker Map

When you use the MAP prelinker option, the Language Environment prelinker produces a Prelinker Map. The default is to generate a listing file. The listing contains several individual sections that are only generated if they are applicable. Unresolved references generate error or warning messages to the Prelinker Map.

Mapping long names to short names

You can use the output object module of the prelinker as input to a system linkage editor.

Because system linkage editors accept only short names, the Language Environment prelinker maps long names to short names on output. It does not change short names. Long names can be up to 1024 characters in length. Truncation of the long names to the 8 character short name limit is therefore not sufficient because name collisions may occur.

The Language Environment prelinker maps a given long name to a short name on output according to the following hierarchy:

1. If any occurrence of the long name is a reserved run-time name, or was caused by a **#pragma map** or C **#pragma csect** directive, then that same name is chosen for all occurrences of the name. This name must not be changed, even if a RENAME control statement for the name exists. For information on the RENAME control statement, see "RENAME control statement" on page 637.
2. If the long name was found to have a matching short name, the same name is chosen. For example, `DOTOTALS` is coded in both a C (or C++) and an assembler program. This name must not be changed, even if a RENAME statement for the name exists. This rule binds the long name to its short name.
3. If a valid RENAME statement for the long name is present, then the short name specified on the RENAME statement is chosen.
4. If the name corresponds to a Language Environment Library function or library object for which you did not supply a replacement, the name chosen is the truncated, uppercased version of the long name library name (with _ mapped to `@`).
5. If you specify the prelinker OMVS option and the name corresponds to a POSIX Language Environment Library function for which you did not supply a replacement, the name chosen is the internal Language Environment Library short name.

   This short name is not chosen, if either:

   - A valid RENAME statement renames another long name to this short name. For example, the RENAME statement `RENAME mybigname PRINTF` would make the library function `printf()` unavailable if `mybigname` is found in input.
   - Another long name is found to have the same name as this short name. For example, explicitly coding and referencing SPRINTF in the C or C++ source program would make the library function `sprintf()` unavailable.

   Avoid such practices to ensure that the appropriate Language Environment Library function is chosen.
6. If the UPCASE option is specified for a C application, names that are 8 characters or fewer are changed to uppercase, with _ mapped to `@`. Names that begin with IBM or CEE will be changed to `IB$`, and `CE$`, respectively. Because of this rule, two different names can map to the same name. You should therefore exercise care when using the UPCASE option. The prelinker issues a warning message is issued if it finds a collision, but it still maps the names.

7. If none of these rules apply, a default mapping is performed. This mapping is the same as the one the compiler option NOLONGNAME uses for external names, taking collisions into account. That is, the name is truncated to 8 characters and changed to uppercase (with _ mapped to `@`). Names that begin with IBM or CEE will be changed to `IB$` and `CE$`, respectively. If this name is the same as the original name, it is always chosen. This name is also chosen if a name collision does not occur. A name collision occurs if either
   - The short name has already been seen in *any* input; that is, the name is not new.
   - After applying this default mapping, the same name is generated for at least two, previously unmapped, names.

   If a name collision occurs, a unique name is generated for the output name. For example, the name `@ST00033` is generated.

A C application that is compiled with the NOLONGNAME compiler option and link-edited, except for collisions, presents the linkage editor with the same names as when the application is compiled with the LONGNAME option and prelinked.

See *z/OS Language Environment Debugging Guide* for a list of error messages that the prelinker returns.

Linking an application

The linkage editor processes your compiled program (object module) and readies it for loading and execution. The processed object module becomes a load module which is stored in a program library or z/OS UNIX System Services file system directory and can be retrieved for execution at any time.

Using DD statements for standard data sets—linkage editor

The linkage editor always requires four standard data sets. You must define these data sets in DD statements with the ddnames SYSLIN, SYSLMOD, SYSUT1, and SYSPRINT.

A fifth data set, defined by a DD statement with the name SYSLIB, is necessary if you want to use the automatic call library. Table 52 shows the five data set names and their characteristics.

Table 52. Data sets used for linking

ddname	Type	Function
SYSLIN	Input	Primary input data, the output of the prelinker, compiler, or assembler
SYSPRINT	Output	Diagnostic messages Informational messages Module map Cross-reference list
SYSLMOD	Output	Output data set for the linkage editor
SYSUT1	Utility	Temporary workspace
SYSLIB[1]	Library	Secondary input
User-specified	Input	Obtain additional object modules and load modules

Notes:
1 Required for library run-time routines

Primary input (SYSLIN)

Primary input to the linkage editor consists of a sequential data set, a member of a partitioned data set, or an inline object module. The primary input must be composed of one or more separately compiled object modules or linkage control statements. A load module cannot be part of the primary input, although the control statement INCLUDE can introduced it. (See "INCLUDE control statement" on page 635.)

Listing (SYSPRINT)

The linkage editor generates a listing that includes reference tables that are related to the load modules that it produces. You must define the data set where you want the linkage editor to store its listing in a DD statement with the name SYSPRINT.

Output (SYSLMOD)

Output (one or more linked load modules) from the linkage editor is always stored in a partitioned data set that is defined by the DD statement with the name SYSLMOD, unless you specify otherwise. This data set is known as a library.

Temporary workspace (SYSUT1)

The linkage editor requires a data set for use as a temporary workspace. The data set is defined by a DD statement with the name SYSUT1. This data set must be on a direct access device.

Secondary input (SYSLIB)

Secondary input to the linkage editor consists of object modules or load modules that are not part of the primary input data set, but are to be included in the load module from the automatic call library. The automatic call library contains load modules or object modules that are to be used as secondary input to the linkage editor to resolve external symbols that remain undefined after all the primary input has been processed.

The call library used as input to the linkage editor or loader can be a system library, a private program library, or a subroutine library.

Input to the linkage editor

Input to the linkage editor can be:

- One or more object modules (created through theOBJECT compiler option)
- Linkage editor control statements (NAME and ALIAS) that are generated by the ALIAS compiler option
- Previously link-edited load modules that you want to combine into one load module
- Language Environment library stub routines (SYSLIB)
- Other libraries

Primary input

Primary input to the linkage editor consists of a sequential data set that contains one or more separately compiled object modules, possibly with linkage editor control statements.

Specify the primary input data set with the SYSLIN statement. For more information on the data sets that are used with z/OS XL C/C++, refer to "Description of data sets used" on page 470.

Secondary input

Secondary input to the linkage editor consists of object modules or load modules that are not part of the primary input data set but are to be included in the load module as the automatic call library.

The automatic call library contains object modules to be used as secondary input to the linkage editor to resolve external symbols left undefined after all primary input has been processed.

The automatic call library may be in the form of:

- Libraries that contain object modules, with or without linkage editor control statements
- Libraries that contain load modules
- The Language Environment Library, if any of the library functions are needed to resolve external references.

Secondary input is either all object modules or all load modules, but it cannot contain both types.

Specify the secondary input data sets with a SYSLIB statement and, if the data sets are object modules, add the linkage editor LIBRARY and INCLUDE control statements.

Additional object modules as input

You can use the INCLUDE and LIBRARY linkage editor control statements to do the following:

1. Specify additional object modules that you want included in the output load module (INCLUDE statement).
2. Specify additional libraries to be searched for object modules to be included in the load module (LIBRARY statement). This statement has the effect of concatenating any specified member names with the automatic call library.

Linkage editor control statements in the primary input must specify any linkage editor processing beyond the basic processing.

Output from the linkage editor

The output from the linkage editor can be a single load module, or multiple load modules, that are generated by using the NAME control statement of the linkage editor.

For more information on using linkage editor control statements, see *z/OS MVS Program Management: User's Guide and Reference*.

SYSLMOD and SYSPRINT are the data sets that are used for link-edit output. The output from the linkage editor varies, depending on the options you select, as shown in Table 53.

Table 53. Options for controlling link-edit output

To Get This Output	Use This Option
A map of the load modules generated by the linkage editor.	MAP
A cross-reference list of data variables	XREF
Informational messages	Default

Table 53. Options for controlling link-edit output (continued)

To Get This Output	Use This Option
Diagnostic messages	Default
Listing of the linkage editor control statements	LIST
One or more load modules (which you must assign to a library)	Default

By default, you receive diagnostic and informative messages as the result of link-editing. You can get the other output items by specifying options in the PARM parameter in the EXEC statement in your link-edit JCL.

The load modules that are created are written in the data set that is defined by the SYSLMOD DD statement in your link-edit JCL. All diagnostic output to be listed is written in the data set that is defined by the SYSPRINT DD statement.

Detecting link-edit errors

You receive a listing of diagnostic messages in SYSPRINT. Check the linkage editor map to make sure that all the object and load modules you expected were included.

You can find a description of link-edit messages in *z/OS MVS Program Management: User's Guide and Reference* .

The instructions for link-edit processing vary, depending on whether you are running under z/OS batch or TSO.

Note: For information on link-editing modules for interlanguage calls, refer to *z/OS Language Environment Programming Guide*.

Library routine considerations

The Language Environment Library consists of one run-time component that contains all Language Environment-enabled languages, such as C, C++, COBOL, and PL/I. For detailed instructions on linking and running z/OS XL C/C++ programs under the Language Environment element, refer to *z/OS Language Environment Programming Guide*.

The Language Environment Library is *dynamic*. This means that many of the functions, such as library functions, available in z/OS XL C/C++ are not physically stored as a part of your executable program. Instead, only a small portion of code is stored with your executable program, resulting in a smaller executable module size. This portion of code is known as a stub routine. The stub routine represents each required library function. Each of these stub routines has:

- The same name as the library function which it represents.
- Enough code to locate the true library function at run time.

The C stub routines are in the file CEE.SCEELKED, which is part of the Language Environment run-time library and must be specified as one of the libraries to be searched during autocall.

Link-editing multiple object modules

z/OS XL C generates a CEESTART CSECT at the beginning of the object module for any source program that contains the function `main()` (and for which the START compiler option was specified) or a function for which a **#pragma linkage (***name***, FETCHABLE)** preprocessor directive applies. When multiple object modules are link-edited into a single load module, the entry point of the resulting load module is resolved to the external symbol CEESTART. Run-time errors occur if the load module entry point is forced to some other symbol by use of the linkage editor ENTRY control statement.

If a C `main()` function is link-edited with object modules produced by C, other language processors or by assembler, the module containing the C `main()` must be the first module to receive control. You must also ensure that the entry point of the resulting load module is resolved to the external symbol CEESTART. To ensure this, the input to the linkage editor can include the following linkage editor ENTRY control statement:

```
ENTRY CEESTART
```

If you are building a DLL, you may need to use the ENTRY control statement.

Building DLLs

Note: This topic does not describe all of the steps that are required to build a DLL. It only describes the prelink step. For a complete description of how to build DLLs, see *z/OS XL C/C++ Programming Guide*.

Except for the object modules you require for creating the DLL, you do not require additional object modules. The prelinker automatically creates a definition side-deck that describes the functions and the variables that DLL applications can import.

Note: Although some C applications may need only the linkage editor to link them, all DLLs require either the use of the binder with the DYNAM(DLL) option, or the prelinker before the linkage editor.

When you build a DLL, the prelinker creates a definition side-deck, and associates it with the SYSDEFSD ddname. You must provide the generated definition side-deck to all users of the DLL. Any DLL application which implicitly loads the DLL must include the definition side-deck when they prelink.

Example: The following is an example of a definition side-deck generated by the prelinker when prelinking a C object module:

```
IMPORT CODE 'BASICIO'   bopen
IMPORT DATA 'BASICIO'   bclose
IMPORT DATA 'BASICIO'   bread
IMPORT DATA 'BASICIO'   bwrite
IMPORT DATA 'BASICIO'   berror
```

You can edit the definition side-deck to remove any functions or variables that you do not want to export. For instance, in this example, if you do not want to expose function `berror`, remove the control statement `IMPORT DATA 'BASICIO'   berror` from the definition side-deck.

Note: You should also provide a header file that contains the prototypes for exported functions and external variable declarations for exported variables.

Example: The following is an example of a definition side-deck generated by the prelinker when prelinking a C++ object module:

```
IMPORT CODE 'TRIANGLE' getarea__8triangleFv
IMPORT CODE 'TRIANGLE' getperim__8triangleFv
IMPORT CODE 'TRIANGLE' __ct__8triangleFv
```

You can edit the definition side-deck to remove any functions and variables that you do not want to export. For instance, in this example, if you do not want to expose `getperim()`, remove the control statement `IMPORT CODE 'TRIANGLE' getperim__8triangleFv` from the definition side-deck.

The definition side-deck contains mangled names, such as `getarea__8triangleFv`. If you want to know what the original function or variable name was in your source module, look at the compiler listing created. Alternatively, use the CXXFILT utility to see both the mangled and demangled names. For more information on the CXXFILT utility, see Chapter 14, "Filter utility," on page 493.

Note: You should also provide users of your DLL with a header file that contains the prototypes for exported functions and `extern` variable declarations for exported variables.

The prelinker NODYNAM option must not be in effect when building DLLs.

Linking your code

When you link your code, ensure that you specify the RENT or REUS(SERIAL) options.

Using DLLs

The prelinker is used to build DLLs that export defined external functions and variables, and to build programs or DLLs that import external functions and variables from other DLLs.

Note: The prelinker NODYNAM option must not be in effect when using or building DLLs.

To assign a name to a DLL, use either the DLLNAME() prelinker option, or the `NAME` control statement. If you do not assign a name, and the data set SYSMOD is a PDS member, the member name is used as the DLL name. Otherwise, the name TEMPNAME is used.

To build a DLL, you need to compile object code that exports external functions or variables, then prelink and link that code into a load module. During the prelink step you need to capture the definition side-deck which is written to the ddname SYSDEFSD. The definition side-deck is a list of IMPORT control statements that correspond to the external functions and variables exported by the DLL.

Include the IMPORT statements at prelink time for any program that imports variables or functions from the DLL.

Example: In the following C example, `EXPONLY` is a DLL which only exports a single variable `year`:

```
/* EXPONLY.C */
int year = 2001;      /* exported from this DLL */
```

Example: In the following example, `IMPEXP` is a DLL that both imports and exports external functions and variables. It imports the external variable `year` from DLL

EXPONLY, and exports external functions `next_year` and `get_year`.

```
/* IMPEXP.C */
extern int year;       /* imported from DLL EXPONLY */

void next_year(void) { /* exported from this DLL    */
  ++year;              /* load DLL EXPONLY, modify 'year' in DLL */
}

int get_year(void) {   /* exported from this DLL    */
  return year;         /* get value of 'year' from DLL EXPONLY */
}
```

Example: In the following example, `IMPONLY` is a program that only imports functions and variables. It imports the variable `year` from DLL `EXPONLY`, and it imports functions `next_year` and `get_year` from DLL `IMPEXP`.

```
/* IMPONLY.C */
#include <stdio.h>
extern int  get_year(void);    /* import from DLL IMPEXP */
extern void next_year(void);   /* import from DLL IMPEXP */
extern int year;               /* import from DLL EXPONLY */
int main(void)
{
  int y;
  next_year();         /* load DLL IMPEXP, call function from DLL */
  y = get_year();      /* call function in DLL IMPEXP */
  if (   y == 2002
      && year == 2002) /* get value of 'year' from DLL EXPONLY */
    printf("pass\n");
  else
    printf("fail\n");
  return 0;
}
```

Example: The following JCL builds the DLLs `EXPONLY`, `IMPEXP`, and the program `IMPONLY`, and then runs `IMPONLY`:

```
//* ------------------------------------------------------------
//CEXPONLY  EXEC EDCC,
//  INFILE='USERID.DLL.C(EXPONLY)',
//  OUTFILE='USERID.DLL.OBJECT(EXPONLY),DISP=SHR ',
//  CPARM='LONG RENT EXPORTALL'
//* ------------------------------------------------------------
//CIMPEXP   EXEC EDCC,
//  INFILE='USERID.DLL.C(IMPEXP)',
//  OUTFILE='USERID.DLL.OBJECT(IMPEXP),DISP=SHR ',
//  CPARM='LONG RENT DLL EXPORTALL'
//* ------------------------------------------------------------
//CIMPONLY  EXEC EDCC,
//  INFILE='USERID.DLL.C(IMPONLY)',
//  OUTFILE='USERID.DLL.OBJECT(IMPONLY),DISP=SHR ',
//  CPARM='LONG RENT DLL'
//* ------------------------------------------------------------
//LINK1     EXEC CBCL,PPARM='DLLNAME(EXPONLY)',
//  OUTFILE='USERID.DLL.LOAD(EXPONLY),DISP=SHR '
//PLKED.SYSIN    DD DSN=USERID.DLL.OBJECT(EXPONLY),DISP=SHR
//PLKED.SYSDEFSD DD DSN=USERID.DLL.IMPORTS(EXPONLY),DISP=SHR
//* ------------------------------------------------------------
//LINK2     EXEC CBCL,PPARM='DLLNAME(IMPEXP)',
//  OUTFILE='USERID.DLL.LOAD(IMPEXP),DISP=SHR '
//PLKED.SYSIN    DD DSN=USERID.DLL.OBJECT(IMPEXP),DISP=SHR
//               DD DSN=USERID.DLL.IMPORTS(EXPONLY),DISP=SHR
//PLKED.SYSDEFSD DD DSN=USERID.DLL.IMPORTS(IMPEXP),DISP=SHR
```

Figure 47. JCL to build DLLs (Part 1 of 2)

```
//* ------------------------------------------------------------
//LINK3     EXEC CBCL,
//  OUTFILE='USERID.DLL.LOAD(IMPONLY),DISP=SHR '
//PLKED.SYSIN    DD DSN=USERID.DLL.OBJECT(IMPONLY),DISP=SHR
//               DD DSN=USERID.DLL.IMPORTS(EXPONLY),DISP=SHR
//               DD DSN=USERID.DLL.IMPORTS(IMPEXP),DISP=SHR
//* ------------------------------------------------------------
//GO       EXEC PGM=IMPONLY
//STEPLIB  DD DSN=USERID.DLL.LOAD,DISP=SHR
//         DD DSN=CEE.SCEERUN,DISP=SHR
//         DD DSN=CEE.SCEERUN2,DISP=SHR
//SYSOUT   DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
```

Figure 47. JCL to build DLLs (Part 2 of 2)

- Both `EXPONLY` and `IMPEXP` are compiled with the option EXPORTALL because they export external functions and variables.
- Both `IMPEXP` and `IMPONLY` are compiled with the option DLL because they import functions and variables from other DLLs.
- Step `LINK1` generates a definition side-deck `USERID.DLL.IMPORTS(EXPONLY)` which is a list of external functions and variables that are exported by DLL `EXPONLY`.
- Step `LINK2` uses the definition side-deck that is generated in step `LINK1` as part of the prelinker input to import the variable `year` from DLL `EXPONLY`.
- Step `LINK2` generates a definition side-deck `USERID.DLL.IMPORTS(IMPEXP)` that is a list of external functions and variables that are exported by DLL `IMPEXP`.
- Both steps `LINK1` and `LINK2` use the prelinker DLLNAME option to set the DLL name seen on `IMPORT` statements generated in the definition side-decks.
- Step `LINK3` uses the definition side-decks generated in step `LINK1` and `LINK2` as part of the prelinker input to import the variable `year` from DLL `EXPONLY` and to import the functions `get_year` and `set_year` from DLL `IMPEXP`.

- Step LINK3 does not specify a definition side-deck; program IMPONLY does not export any functions or variables.
- If you explicitly specify link-time parameters, be sure to specify the RENT option. The IBM-supplied cataloged procedure CBCL does this by default.
- The load module name of a DLL must match the DLLNAME seen on the corresponding IMPORT statements.
- Step GO has the program IMPONLY and the DLLs. EXPONLY and IMPEXP in its STEPLIB concatenation so that the DLLs can be dynamically loaded at run time.

To see which functions and variables are imported or exported use the Prelinker Map. The following is a portion of the Prelinker Map from step LINK2:

```
========================================================================
|                           Load Module Map  1                         |
========================================================================

MODULE ID  MODULE NAME

  00001    EXPONLY


========================================================================
|                          Import Symbol Map  2                        |
========================================================================

*TYPE    FILE ID  MODULE ID  NAME

   D      00001     00001    year

*TYPE:  D=imported data  C=imported code


========================================================================
|                          Export Symbol Map  3                        |
========================================================================

*TYPE    FILE ID  NAME

   C      00001   get_year
   C      00001   next_year

*TYPE:  D=exported data  C=exported code
```

1 Load Module Map
This section lists the load modules from which functions and variables are imported. The load module names come from the input IMPORT control statements processed.

2 Import Symbol Map
This section lists the imported functions and variables. The MODULE ID indicates the DLL from which the function or variable is imported. The FILE ID indicates the file in which the IMPORT control statement was processed that resulted in this import.

3 Export Symbol Map
This section lists the external functions and variables which are exported. For each symbol that is listed in this section, an IMPORT control statement is written out to the DDname SYSDEFSD, the definition side-deck.

Note: The export symbol map will not be produced when the NODYNAM option is in effect.

Prelinking and linking an application under z/OS batch and TSO

Figure 48 shows the basic prelinking and linking process for your C or C++ application.

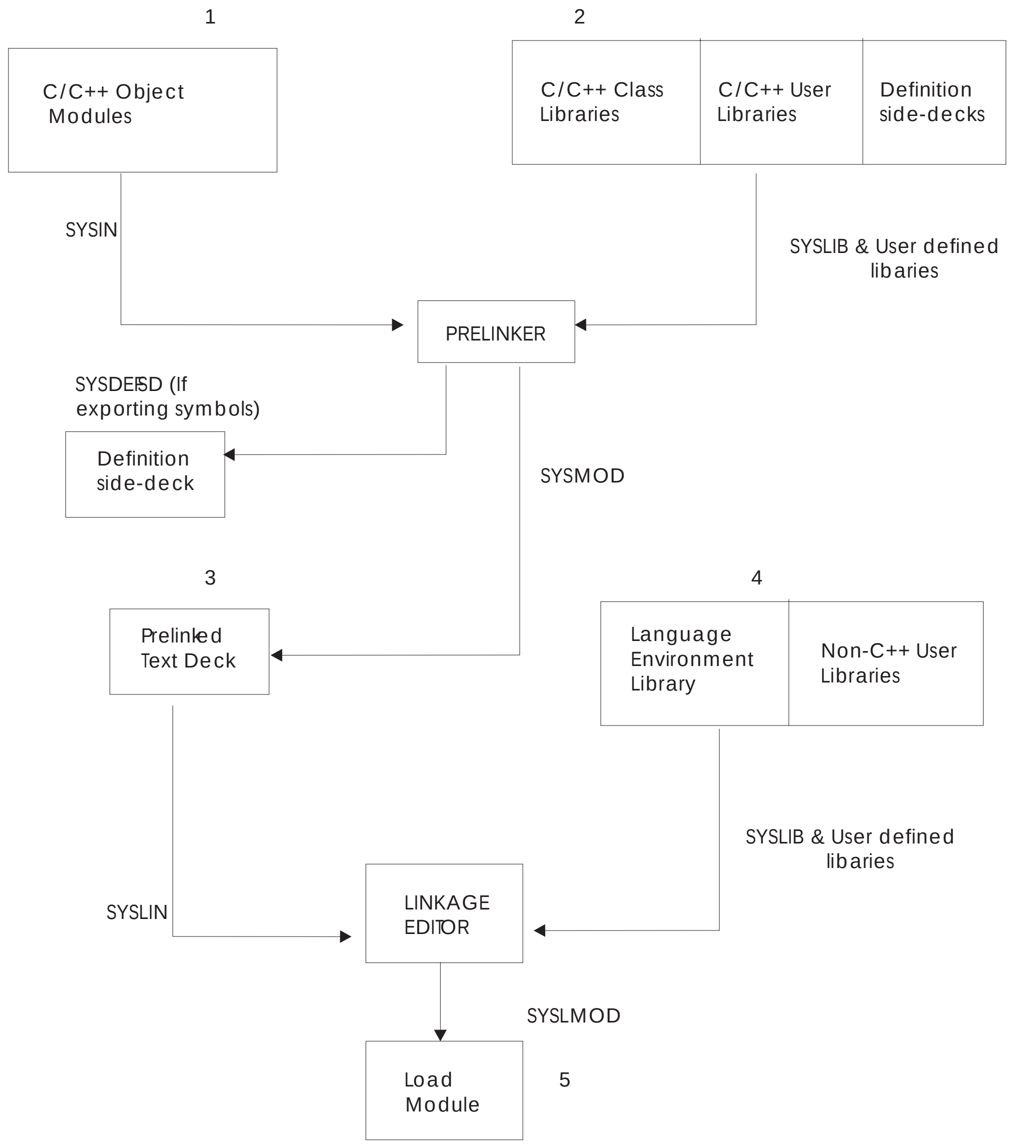


Figure 48. Basic prelinker and linkage editor processing

The data set SYSIN, **1** , that contains your object modules forms the primary input of the prelinker.

Note: If you are creating an application that imports symbols from DLLs, you must provide the definition side-deck for each DLL referenced in SYSIN.

The prelinker uses its primary input, and its secondary input, **2** , from SYSLIB to produce a prelinked object module and, if you are exporting symbols, a definition side-deck. SYSLIB points to PDS libraries or PDSE libraries which may contain the following:
- Object modules with long names
- Object modules with writable static references

- C/C++ object module libraries
- DLL definition side-decks

The prelinked output object module is put in SYSMOD. If a definition side-deck is generated, it is put in SYSDEFSD, which is a sequential data set or a PDS member.

The linkage editor takes its primary input from SYSLIN which refers to the prelinked object module data set, **3**. The linkage editor uses the primary input and secondary input, **4**, to produce a load module, **5**. The secondary input consists of non-C++ user defined libraries, and the Language Environment run-time library (SCEELKED) specified using SYSLIB.

The load module, **5**, is put in the SYSLMOD data set. The load module becomes a permanent member of SYSLMOD. You can be retrieve it at any time to run in the job that created it, or in any other job.

Language Environment Prelinker Map

When you use the MAP prelinker option, the Language Environment prelinker produces a Prelinker Map. The listing contains several individual sections that are only generated if they are applicable.

Example: Consider the following example. The data set `USERID.DLL.SOURCE(EXPONLY)` contains

```
/* EXPONLY.C */
  int year = 2001;  /* exported from this DLL */
```

After step `LINK0` in Figure 50 on page 617, the definition side-deck `USERID.DLL.IMPORTS(EXPONLY)` contains the record `IMPORT DATA 'EXPONLY'  year`.

The map that is shown in Figure 51 on page 617 was created by compiling the program that is shown in Figure 49. Figure 51 on page 617 is the corresponding Prelinker Map from step `LINK1` The linkage editor places the resulting load module in `USERID.DLL.LOAD(IMPEXP2)`.

```
/* IMPEXP2.C */
#pragma variable(this_int_not_in_writable_static, NORENT)
int this_int_not_in_writable_static = 2001;
extern int year;
int this_int_is_in_writable_static = 1900;
int get_year(void) {
  return year;
}
void next_year(void) {
  year++;
}
void Name_Collision_In_First8(void) {
}
void Name_Collision_In_First_Eight(void) {
}
```

Figure 49. z/OS XL C++ source file used for the example Prelinker Map

```
//*
//COMP0  EXEC  CBCC,CPARM='EXPORTALL',
//  INFILE='USERID.DLL.SOURCE(EXPONLY)',
//  OUTFILE='USERID.DLL.OBJECT(EXPONLY),DISP=SHR'
//LINK0  EXEC  CBCL,PPARM='DLLNAME(EXPONLY) NONCAL MAP',
//  OUTFILE='USERID.DLL.LOAD(EXPONLY),DISP=SHR'
//PLKED.SYSIN     DD  DSN=USERID.DLL.OBJECT(EXPONLY),DISP=SHR
//PLKED.SYSDEFSD  DD  DSN=USERID.DLL.DEFSD(EXPONLY),DISP=SHR
//*
//COMP1  EXEC  CBCC,CPARM='EXPORTALL',
//  INFILE='USERID.DLL.SOURCE(IMPEXP2)',
//  OUTFILE='USERID.DLL.OBJECT(IMPEXP2),DISP=SHR'
//LINK1  EXEC  CBCL,PPARM='DLLNAME(IMPEXP2) NONCAL MAP',
//  OUTFILE='USERID.DLL.LOAD(IMPEXP2),DISP=SHR'
//PLKED.SYSIN     DD  DSN=USERID.DLL.OBJECT(IMPEXP2),DISP=SHR
//                DD  DSN=USERID.DLL.DEFSD(EXPONLY),DISP=SHR
//PLKED.SYSDEFSD  DD  DSN=USERID.DLL.DEFSD(IMPEXP2),DISP=SHR
```

Figure 50. Example of JCL used to generate the example Prelinker Map for a C++ program.

```
=========================================================================
|                            Prelinker Map  1                           |
|                                                                       |
| CPLINK:5694A01 V1 R11 M0 IBM LANGUAGE ENVIRONMENT 2009/04/29 04:55:08|
=========================================================================

Command Options. . . . . : NONCAL   NOMEMORY ER       DUP      MAP
                         : OMVS     NOUPCASE DYNAM


=========================================================================
|                      Object Resolution Warnings  2                    |
=========================================================================

WARNING EDC4015: Unresolved references are detected:
CEESTART CEESG003 @@TRGLOR
```

Figure 51. Prelinker Map (Part 1 of 3)

```
========================================================================
|                               File Map  3                            |
========================================================================

*ORIGIN  FILE ID  FILE NAME

   P      00001   DD:SYSIN   (IMPEXP2)
   A      00002   CEE.SCEECPP(EDC400BA)
   IN     00003   *** DESCRIPTORS ***

*ORIGIN:  P=primary input     PI=primary INCLUDE    SI=secondary INCLUDE
          A=automatic call     R=RENAME card         L=C Library
         IN=internal


========================================================================
|                         Writable Static Map  4                       |
========================================================================

  OFFSET    LENGTH  FILE ID  INPUT NAME

       0         4   00001   this_int_is_in_writable_static
       8        10   00003   <year>
      18         4   00001   @STATIC


========================================================================
|                           Load Module Map  5                         |
========================================================================

MODULE ID  MODULE NAME

  00001    EXPONLY


========================================================================
|                          Import Symbol Map  6                        |
========================================================================

*TYPE    FILE ID  MODULE ID  NAME

   D      00001     00001    year

*TYPE:  D=imported data  C=imported code
```

Figure 51. Prelinker Map (Part 2 of 3)

```
=========================================================================
|                           Export Symbol Map 7                         |
=========================================================================

*TYPE    FILE ID  NAME

   C      00001   get_year()
   D      00001   this_int_is_in_writable_static
   D      79552   this_int_not_in_writable_static
   C      00001   Name_Collision_In_First_Eight()
   C      00001   Name_Collision_In_First8()

*TYPE:  D=exported data  C=exported code


=========================================================================
|                   ESD Map of Defined and Long Names 8                 |
=========================================================================

                   OUTPUT
*REASON  FILE ID  ESD NAME   INPUT NAME

   P              CEESTART   CEESTART
   D      00001   THIS@INT   this_int_not_in_writable_static
   D      00001   GET@YEAR   get_year()
   D      00001   @ST00003   Name_Collision_In_First_Eight()
   D      00001   @ST00002   Name_Collision_In_First8()
   P              CEESG003   CEESG003
   P      00002   CBCSG003   CBCSG003
   P              @@TRGLOR   @@TRGLOR

*REASON: P=#pragma or reserved    S=matches short name    R=RENAME card
         L=C Library              U=UPCASE option         D=Default


============  E N D   O F   P R E - L I N K A G E   M A P  =============
```

| *Figure 51. Prelinker Map (Part 3 of 3)*

The numbers in the following text correspond to the numbers that are shown in the map.

1 Heading
The heading is always generated. It contains the product number, the library release number, the library version number, and the date and the time the prelink step began. A list of the prelinker options that are in effect for the step follow.

2 Object Resolution Warnings
This section is generated if objects remained undefined at the end of the prelink step, or the IPA Link step, or if duplicate objects were detected during the step. The names of the applicable objects are listed.

3 File Map
This section lists the object modules that were included in input. An object module consisting only of RENAME control statements, for example, is *not* shown. Also provided in this section are source origin (`FILE NAME`), and identifier (`FILE ID`) information. The object module came from primary input because of:
- An INCLUDE control statement in primary or secondary input
- A RENAME control statement
- The resolution of long name library references
- The object module was internal and self-generated by the prelink step

The FILE ID may appear in other sections, and is used as a cross reference to the object module. The FILE NAME can be one of:

- The data set name and, if applicable, the member name
- The ddname and, if applicable, the member name
- The z/OS UNIX System Services file name

If you are prelinking an application that imports variables or functions from a DLL, the variable descriptors and function descriptors are defined in a file called `*** DESCRIPTORS ***`. This file has an origin of internal.

4 Writable Static Map
This section is generated if an object module was encountered that contains defined static external data. This area also contains variable descriptors for any imported variables and, if required, function descriptors. This section lists the names of such objects, their lengths, their relative offset within the writable static area, and a FILE ID for the file containing the definition of the object.

5 Load Module Map
This section is generated if the application imports symbols from other load modules. This section lists the names of the load modules.

6 Import Symbol Map
This section is generated if symbols are imported from other load modules. These otherwise unresolved DLL references are resolved through IMPORT control statements. This section lists those symbols. It describes the type of symbol; that is, D (variable) or C (function). It also lists the file id of the object module containing the corresponding IMPORT control statements, the module id of the load module on that control statement, and the symbol name.

A DLL application would generate this section.

7 Export Symbol Map
This section is generated if an object module is encountered that exports symbols. This section lists those symbols. It describes the type of symbol; that is, D (variable) or C (function). It also lists the file id of the object where the symbol is defined and the symbol name. Only externally defined data objects in writable static or externally defined functions can be exported.

Code that is compiled with the EXPORTALL compiler option or code that contains the **#pragma export** directive would generate an object module that exports symbols.

Note: The export symbol map will not be produced if the NODYNAM option is in effect.

8 ESD Map of Defined and Long Names
This section lists the names of external symbols that are not in writable static. It also shows a mapping of input long names to output short names.

If the object is defined, the FILE ID indicates the file that contains the definition. Otherwise, this field is left blank. For any name, the input name and output short name are listed. If the input name is indeed an long name, the rule that is used to map the long name to the short name is applied. If the name is not an long name, this field is left blank.

Note: Although mangled names exist in the object modules, the Prelinker Map and messages emit the demangled equivalent, which is like the names seen in the C++ source code.

Processing the prelinker automatic library call

The following hierarchy is used to resolve a referenced and currently undefined symbol.

- The undefined name is an short name, for example `SNAME`.
  - If the NONCAL command option is in effect, the partitioned data sets that are concatenated to SYSLIB are searched in order as follows:
    - If the data set contains a C370LIB-directory created using the z/OS XL C/C++ object library utility, and the C370LIB-directory shows that a defined symbol by that name exists, the member of the PDS containing that symbol is read.
    - If the data set does not contain a C370LIB-directory created using the z/OS XL C/C++ object library utility and the reference is not to static external data, the member or alias, with the same name as `SNAME` is read.
- The undefined name is an long name.
  - If the NONCAL command option is in effect, the partitioned data sets that are concatenated to SYSLIB are searched. If the data set contains a C370LIB-directory created using the z/OS XL C/C++ object library utility, and the C370LIB-directory shows that a defined symbol by that name exists, the member of the PDS indicated as containing that symbol is read.

For more information about the z/OS XL C/C++ object library utility, see Chapter 13, “Object library utility,” on page 481.

References to currently undefined symbols (external references)

If the symbol is undefined after the prelink step, and is not a writable static symbol, it may be subsequently defined during the link step. However, the definition must be exactly the same as the output ESD name. For more information, see the Figure 51 on page 617.

If you are writing a C application, and the symbol is an long name that was not resolved by automatic library call and for which a RENAME statement with the SEARCH option exists, the symbol is resolved under the short name on the RENAME statement by automatic library call.

See “RENAME control statement” on page 637 for a complete description of the RENAME control statement.

Unresolved requests generate error or warning messages to the Prelinker Map.

Prelinking and linking under z/OS batch

Using IBM-supplied cataloged procedures

The IBM-supplied catalog procedures and REXX EXECs use the DLL versions of the IBM-supplied class libraries by default. That is, the IBM-supplied Class Libraries definition side-deck data set, SCLBSID, is included in the SYSIN concatenation.

If you are *statically* linking the relevant class library object code, you must override the `PLKED.SYSLIB` concatenation to include the SCLBCPP or SCLBCPP2 data set. The z/OS V1R2 version of the static library is in CBC.SCLBCPP2.

Note: If your application consists of multiple modules (for example, a main module and a DLL) that use the same class library, make sure that all your modules link dynamically to the class library. Otherwise, the class library will be linked in multiple times, and there will be multiple copies in use by your application. You cannot use multiple copies of a class library within a single application. If you do, you can have unexpected results.

You can use one of the following IBM-supplied cataloged procedures that include a link-edit step to link-edit your z/OS XL C program:

EDCCL
: Compile and link-edit

EDCCLG
: Compile, link-edit, and run

EDCCPL
: Compile, prelink, and link-edit

EDCCPLG
: Compile, prelink, link-edit, and run

Note: By default, the procedures EDCCL, EDCCLG, and EDCCPLG do not save the compiled object. EDCCLG and EDCCPLG do not save load modules.

See Chapter 12, "Cataloged procedures and REXX EXECs," on page 465 for more information on REXX EXECs and their uses.

Example: The following example shows the general job control procedure for link-editing a program under z/OS batch using the Language Environment Library.

```
// jobcard
//*
//* THE FOLLOWING STEP LINKS THE MEMBERS TESTFILE AND DECODE FROM
//* THE LIBRARIES USERID.WORK.OBJECT AND USERID.LIBRARY.OBJECT AND
//* PLACES THE LOAD MODULE IN USERID.WORK.LOAD(TEST)
//*
//LKED    EXEC  PGM=IEWL,REGION=1024K,PARM='AMODE=31,RMODE=ANY,MAP'
//SYSLIB    DD  DSNAME=CEE.SCEELKED,DISP=SHR
//SYSLIN    DD  DDNAME=SYSIN
//SYSLMOD   DD  DSNAME=USERID.WORK.LOAD(TEST),DISP=SHR
//OBJECT    DD  DSNAME=USERID.WORK.OBJECT,DISP=SHR
//LIBRARY   DD  DSNAME=USERID.LIBRARY.OBJECT,DISP=SHR
//SYSPRINT  DD  SYSOUT=*
//SYSUT1    DD  UNIT=VIO,SPACE=(32000,(30,30))
//SYSIN     DD  DATA,DLM=@@
     INCLUDE    OBJECT(TESTFILE)
     INCLUDE    LIBRARY(DECODE)
@@
```

Figure 52. Link-editing a program under z/OS batch

You can use one of the following IBM-supplied cataloged procedures that include a prelink and link step to link your C++ program:

CBCCL
: Compile, prelink, and link

CBCL Prelink and link

CBCCLG
: Compile, prelink, link, and run

CBCLG
: Prelink, link, and run.

Specifying prelinker and link-edit options using cataloged procedures

In the cataloged procedures use the PPARM statement to specify prelinker options and the LPARM statement to specify link-edit options as follows:

```
PPARM='"prelinker-options"'
LPARM='"link-edit-options"'
```

where *prelinker-options* is a list of prelinker options and *link-edit-options* is a list of link-edit options. Separate link-edit options and prelinker options with commas.

Writing JCL for the prelinker and linkage editor

You can use cataloged procedures rather than supply all of the job control language (JCL) required for a job step that invokes the prelinker or linkage editor. However, you should be familiar with these JCL statements. This familiarity enables you to make the best use of the prelinker and linkage editor and, if necessary, override the statements of the cataloged procedure.

For a description of the IBM-supplied cataloged procedures that include a prelink and link step, see Chapter 12, “Cataloged procedures and REXX EXECs,” on page 465.

The following sections describe the basic JCL statements for prelinking and linking.

Using the EXEC statement

Use the EXEC job control statement in your JCL to invoke the prelinker. The following example shows an EXEC statement that invokes the prelinker:

```
//PLKED EXEC PGM=EDCPRLK
```

You can also use the EXEC job control statement in your JCL to invoke the linkage editor. The following is a sample EXEC statement that invokes the linkage editor:

```
//LKED EXEC PGM=HEWL
```

Note: If you are using DLLs, you must use the RENT linkage editor option.

Using the PARM parameter

By using the PARM parameter of the EXEC statement, you can select one or more of the optional facilities that the prelinker and linkage editor provide.

For example, if you want the prelinker to use the automatic call library to resolve unresolved references, specify the NONCAL prelinker option using the PARM parameter on the prelinker EXEC statement:

```
//PLKED EXEC PGM=EDCPRLK,PARM='NONCAL'
```

If you want a mapping of the load modules produced by the linkage editor, specify the MAP option with the PARM parameter on the linkage editor EXEC statement:

```
//LKED EXEC PGM=HEWL,PARM='MAP'
```

For a description of prelinker options see “Prelinker options” on page 643, for linkage editor options see “Linkage editor options” on page 645.

Example of JCL to prelink and link

Figure 53 on page 624 shows a typical sequence of job control statements to link-edit an object module into a load module.

```
//*----------------------------------------------------------------
//* PRE-LINKEDIT STEP:
//*----------------------------------------------------------------
//PLKED  EXEC PGM=EDCPRLK,REGION=2048K,PARM='MAP'
//STEPLIB  DD  DSN=CEE.SCEERUN,DISP=SHR
//         DD  DSN=CEE.SCEERUN2,DISP=SHR
//SYSMSGS  DD  DSN=CEE.SCEEMSGP(EDCPMSGE),DISP=SHR
//SYSLIB   DD  DSN=CEE.SCEECPP,DISP=SHR
//         DD  DSN=CBC.SCLBCPP,DISP=SHR
//SYSIN    DD  DSN=USERID.TEXT(PROG1),DISP=SHR
//SYSMOD   DD  DSN=&&PLKSET,UNIT=VIO,DISP=(MOD,PASS),
//             SPACE=(32000,(30,30)),
//             DCB=(RECFM=FB,LRECL=80,BLKSIZE=32000)
//SYSDEFSD DD  DSN=USERID.TEXT(PROG1IMP),DISP=SHR
//SYSOUT   DD  SYSOUT=*
//SYSPRINT DD  SYSOUT=*
//*----------------------------------------------------------------
//* LINKEDIT STEP:
//*----------------------------------------------------------------
//LKED   EXEC PGM=HEWL,REGION=1024K,COND=(8,LE,PLKED),PARM='MAP'
//SYSLIB   DD  DSN=CEE.SCEELKED,DISP=SHR
//SYSLIN   DD  DSN=*.PLKED.SYSMOD,DISP=(OLD,DELETE)
//SYSLMOD  DD  DSN=USERID.LOAD(PROG1),DISP=SHR
//SYSUT1   DD  UNIT=VIO,SPACE=(32000,(30,30))
//SYSPRINT DD  SYSOUT=*
```

Figure 53. Creating a load module under z/OS batch

Note: For a C++ application, this JCL uses static class libraries.

Specifying link-edit options through JCL

In your JCL for link-edit processing, use the PARM statement to specify link-edit options:

```
PARM=(link-edit-options)
PARM.STEPNAME=('link-edit-options') (If a PROC is used)
```

where *link-edit-options* is a list of link-edit options. Separate the link-edit options with commas.

You can prelink and link C/C++ applications under z/OS batch by submitting your own JCL to the operating system or by using the IBM cataloged procedures. See Chapter 12, “Cataloged procedures and REXX EXECs,” on page 465 for more information on the supplied procedures.

Secondary input to the linker

Secondary input is either all object modules or all load modules, but it cannot contain both types.

Specify the secondary input data sets with a SYSLIB statement and, if the data sets are object modules, add the linkage editor LIBRARY and INCLUDE control statements. If you have multiple secondary input data sets, concatenate them as follows:

```
//SYSLIB DD DSNAME=CEE.SCEELKED,DISP=SHR
//       DD DSNAME=AREA.SALESLIB,DISP=SHR
```

To specify additional object modules or libraries, code INCLUDE and LIBRARY statements after your DD statements as part of your job control procedure, such as in Figure 54 on page 625.

```
   ⋮
//SYSLIN DD    DSNAME=&&GOFILE,DISP=(SHR,DELETE)
//       DD    *
   INCLUDE ddname(member)
   LIBRARY ADDLIB(CPGM10)
/*
```

Figure 54. Linkage editor control statements

When the linkage editor encounters the INCLUDE statement, it incorporates the data sets that the control statement specifies. In contrast, the linkage editor uses the data sets that are specified by the LIBRARY statement only when there are unresolved references after it.

When you use cataloged procedures or your own JCL to invoke the linkage editor, external symbol resolution by automatic library call involves a search of the data set defined by the DD statement with the name SYSLIB.

Using additional input object modules under z/OS batch

When you use cataloged procedures or your own JCL to invoke the prelinker and linkage editor, external symbol resolution by automatic library call involves a search of the SYSLIB data set. The prelinker and linkage editor locate the functions in which the external symbols are defined (if such functions exist), and include them in the output module.

You can use prelinker and linkage control statements INCLUDE and LIBRARY to do the following:

1. Specify additional object modules that you want included in the output module (INCLUDE statement).
2. Specify additional libraries to be searched for modules to be included in the output module (LIBRARY statement). This statement has the effect of concatenating any specified member names with the automatic call library.

Example: Code these statements after your DD statements as part of your job control procedure; for example:

```
   ⋮
//SYSIN DD    DSNAME=&&GOFILE,DISP=(SHR,DELETE)
//      DD    *
   INCLUDE ddname(member)
   LIBRARY ADDLIB(CPGM10)
/*
```

Data sets specified by the INCLUDE statement are incorporated as the prelinker and linkage editor encounter the statement. In contrast, data sets specified by the LIBRARY statement are used only when there are unresolved references after all the other input is processed.

Any prelinker and linkage editor processing beyond the basic processing must be specified by linkage editor control statements in the primary input.

Under TSO

The Language Environment prelinker is started under TSO through REXX EXECs. The IBM-supplied REXX EXECs that invoke the prelinker and create an executable module are called CXXMOD and CPLINK. If you want to create a reentrant load module, you must use these REXX EXECs instead of the TSO LINK command. It is recommended that you use CXXMOD instead of CPLINK. For a description of the

CXXMOD REXX EXEC see “Prelinking and linking under TSO.” For a description of the CPLINK command see “Other z/OS XL C utilities” on page 477.

When using the TSO LINK command processor, the data set defined by the LIB operand will be used by the command processor for external symbol resolution. The linkage editor locates the functions in which the external symbols are defined (if such functions exist), and includes them in the load module.

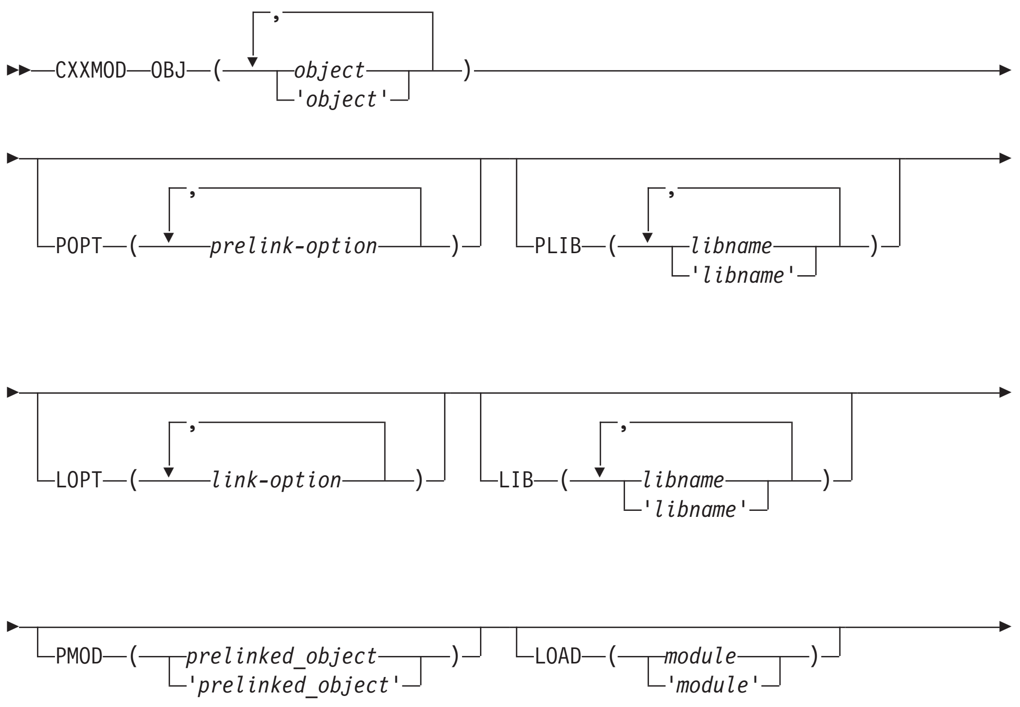
Any linkage editor processing beyond the basic processing must be specified by linkage editor control statements in the primary input. The IBM-supplied catalog procedures and REXX EXECs use the DLL versions of the IBM-supplied class libraries by default.

To link-edit your z/OS XL C program under TSO, use either the CXXMOD, CMOD, or the LINK command. It is recommended that you use CXXMOD, particularly when linking z/OS XL C and z/OS XL C++ object decks. For a description of the CXXMOD REXX EXEC see “Prelinking and linking under TSO.” For a description of CMOD and the TSO LINK command see “Other z/OS XL C utilities” on page 477.

Prelinking and linking under TSO

This topic describes how to prelink and link your z/OS XL C++ or z/OS XL C program by invoking the CXXMOD REXX EXEC. This REXX EXEC creates an executable module.

The syntax for the CXXMOD REXX EXEC is:



```
►─┬─────────────────────────────────────┬─┬──────────────────────────────┬──────────────►
  └─PMAP──(──┬─prelink-map───┬──)─┘ └─LIST──(──┬─listing───┬──)─┘
             └─'prelink-map'─┘                  └─'listing'─┘


►─┬──────────────────────────────────────────┬─────────────────────────────────────────►◄
  └─PDEF──(──┬─prelink-object───┬──)─┘
             └─'prelink-object'─┘
```

CXXMOD

OBJ You must *always* specify the input file names on the OBJ keyword parameter. Each input file must be a C, C++ or assembler object module. Note that the file can be either a PDS member, a sequential file or a z/OS UNIX System Services file.

If the high-level qualifier of a file is not the same as your user prefix, you must use the fully qualified name of the file and place single quotation marks around the entire name.

Note: For z/OS UNIX file names, neither commas nor special characters need to be escaped. But you must place file names containing special characters or commas between single quotation marks. If a single quotation mark is part of the file name, the quotation mark must be specified twice. z/OS UNIX file names must be absolute names, that is they must begin with a slash (/).

POPT Prelinker options can be specified using the POPT keyword parameter. If the MAP prelink option is specified, a prelink map will be written to the file specified under the PMAP keyword parameter. For more details on generating a prelink map, see the information on the PMAP option below.

LOPT Linkage editor options can be specified using the LOPT keyword parameter. For details on how to generate a linkage editor listing, see the option LIST.

PLIB The library names that are to be used by the automatic call library facility of the prelinker must be specified on the PLIB keyword parameter. The default library used is the C++ base library, CEE.SCEECPP.

If the high-level qualifier of a library data set is not the same as your user prefix, you must use the fully qualified name of the data set and place single quotation marks around the entire name.

LIB If you want to specify libraries for the link step to resolve external references, use the LIB keyword parameter. The default library used is CEE.SCEELKED.

If the high-level qualifier of a library data set is not the same as your user prefix, you must use the fully qualified name of the data set and place single quotation marks around the entire name.

PMOD If you want to keep the output prelinked object module, specify the file that it should be placed in by using the PMOD keyword parameter. The default action is to create a file and erase it after the link is complete. The file can be either a data set or a z/OS UNIX file.

If the high-level qualifier of the output prelinked object module is not the same as your user prefix, you must use the fully qualified name of the file and place single quotation marks around the entire name.

LOAD To specify where the resultant load module should be placed, use the LOAD keyword parameter. The file can be either a data set or a z/OS UNIX file.

If the high-level qualifier of the load module is not the same as your user prefix, you must use the fully qualified name of the file and place single quotation marks around the entire name.

LIST To specify where the linkage editor listing should be placed, use the LIST keyword parameter. The file can be either a data set or a z/OS UNIX file. If you specify *, the listing will be directed to your console.

If the high-level qualifier of the linkage editor listing is not the same as your user prefix, you must use the fully qualified name of the file and place single quotation marks around the entire name.

PMAP To specify where the Prelinker Map should be placed, use the PMAP keyword parameter. The file can be either a data set or a z/OS UNIX file. If you specify *, the Prelinker Map will be directed to your console.

If the high-level qualifier of the Prelinker Map is not the same as your user prefix, you must use the fully qualified name of the file and place single quotation marks around the entire name.

PDEF To specify where the generated IMPORT control statements should be placed by the prelinker. The file can be either a data set or a z/OS UNIX file.

If the high-level qualifier of the IMPORT control statement listing is not the same as your user prefix, you must use the fully qualified name of the file and place single quotation marks around the entire name.

Example of prelinking and linking under TSO

In the following example, the user prefix is `RYAN` and the input object module members `MAIN` and `FN` are in the PDS called `RYAN.ACCOUNT.OBJ`. A prelink map is to be generated and placed in `'RYAN.ACCOUNT.MAP(SALES)'`. The load module will be placed in a PDS member called `GROUP.ACCOUNT.LOAD(SALES)`. The linkage editor listing will be written to `RYAN.ACCOUNT.LIST(SALES)`.

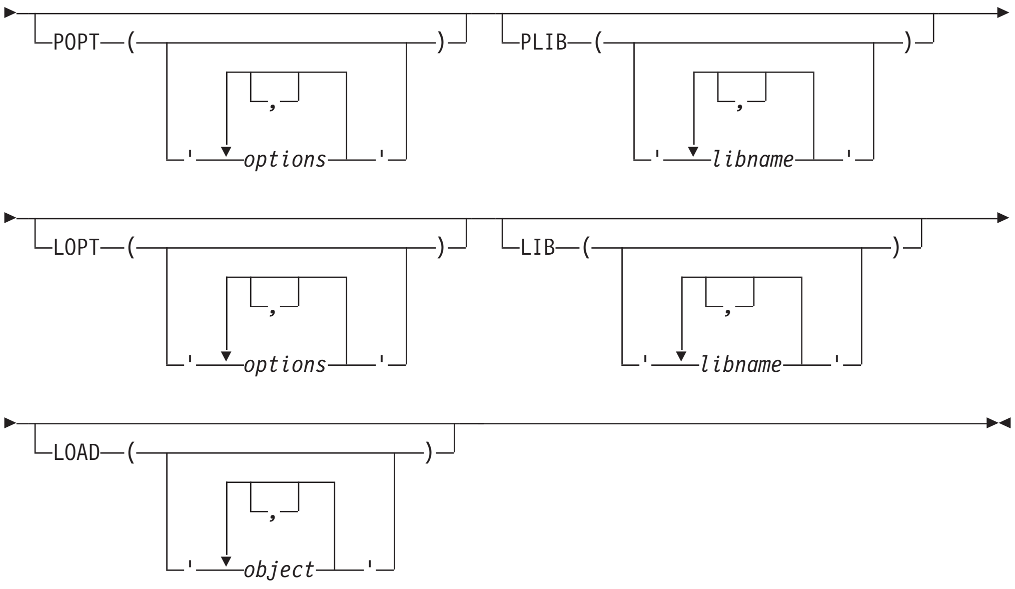
```
CXXMOD OBJ(ACCOUNT.OBJ(MAIN), ACCOUNT.OBJ(FN))
       POPT(MAP) LOPT(XREF, MAP)
       LOAD('GROUP.ACCOUNT.LOAD(SALES)') MAP(ACCOUNT.MAP(SALES))
       LIST(ACCOUNT.LIST(SALES))
```

In this instance, both the Language Environment stub library and the partitioned data set (library) `SALESLIB` are available as the automatic call libraries. The linkage editor LIBRARY control statement has the effect of concatenating any specified member names with the automatic call library.

Using CPLINK

The CPLINK command has the following syntax:

```
►►──CPLINK──OBJ──(──┬──────────────────────┬──)──────────────────────────────►
                    │       ┌─,──┐         │
                    └─'─────▼─object─┴──'──┘
```



OBJ specifies an input data set name.

This is a required parameter. Each input data set must be a C object module compiled with the RENT or LONGNAME compiler options, or a compiled program (C or otherwise) having no static external data.

POPT specifies a string of prelink options.

The prelinker options available for CPLINK are the same as for z/OS batch. For example, if you want the prelinker to use the MAP option, specify the following:

```
CPLINK file name POPT('MAP')..
```

When you specify the prelink MAP option (as opposed to the link MAP option), the prelinker produces a file that shows the mapping of static external data. This map shows name, length, and address information. If there are any unresolved references or duplicate symbols during the prelink step, the map displays them.

PLIB specifies the library names that the prelinker uses for the automatic library call facility.

LOPT specifies a string of linkage editor options.

For example, if you want the prelink utility to use the MAP option, and the linkage editor to use the NOMAP option, use the following CLIST command:

```
CPLINK file name POPT('MAP') LOPT('NOMAP...')
```

LIB specifies any additional library or libraries that the TSO LINK command uses to resolve external references. These libraries are appended to the default C library functions.

LOAD specifies an output data set name.

If you do not specify an output data set name, a name is generated for you. The name that the CLIST generates consists of your user prefix, followed by `CPOBJ.LOAD(TEMPNAME)`. For more information on the file format for output data, refer to *z/OS MVS Program Management: User's Guide and Reference*.

Examples

In the following example, your user prefix is `RYAN`, and the data set that contains the input object module is the partitioned data set `RYAN.C.OBJ(INCCOMM)`. This example will generate a prelink listing without using the automatic call library. After the call, the load module is placed in the partitioned data set `RYAN.CPOBJ.LOAD(TEMPNAME)`, and the prelink listing is placed in the sequential data set `RYAN.CPOBJ.RMAP`.

```
CPLINK OBJ('C.OBJ(INCCOMM)')
```

In the following examples, assume that your user prefix is `PAUL`, and the data set that contains the input object module is the partitioned data set `PAUL.C.OBJ(INCPYRL)`. This example will not generate a prelink listing, and the automatic call facility will use the library `RAINBOW.LIB.SUB`. The load module is placed in the partitioned data set `PAUL.TBD.LOAD(MOD)`.

```
//*--------------------------------------------------------
//* Prelink and link 'PAUL.C.OBJ(INCPYRL)'
//*--------------------------------------------------------
//P0014001  EXEC EDCPL,
//          INFILE='PAUL.C.OBJ(INCPYRL)',
//          OUTFILE='PAUL.TBD.LOAD(MOD),DISP=SHR',
//          PPARM='NOMAP,NONCAL',
//          LPARM='AMODE(31),RMODE(ANY) '
//*-------------------------------------------------------
```

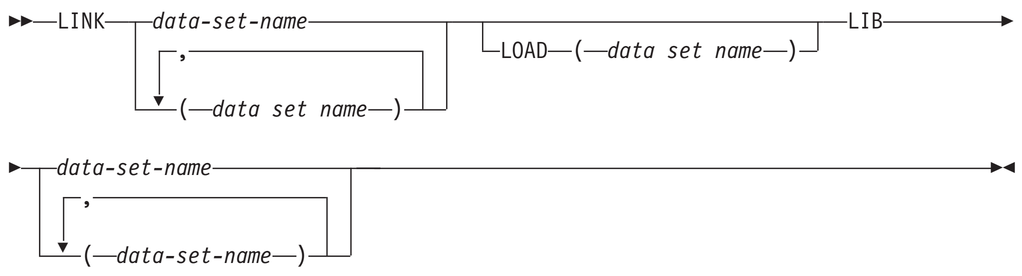
Figure 55. Example of prelinking under z/OS batch

```
CPLINK OBJ('''PAUL.C.OBJ(INCPYRL)''')
      POPT('NOMAP,NONCAL')
      PLIB('''RAINBOW.LIB.SUB''')
      LOAD('TBD.LOAD(MOD)')
```

Figure 56. Example of prelinking under TSO

Using LINK

The general form of the TSO LINK command is:



Input to the LINK command

You must specify one or more object module names, or load module names, after the LINK keyword. For example, to link-edit *program2.obj*, using the Language Environment Library, you would issue the following:

```
LINK program2.obj LIB('CEE.SCEELKED')
```

Notes:

1. You must always specify `'CEE.SCEELKED'` in the LIB operand. It is not required during the execution of a z/OS XL C/C++ program.

LIB operand of the LINK command

The LIB operand specifies the names of data sets that are to be used to resolve external references by the automatic library call facility. Language Environment Library is made available to your program in this manner and must always be specified on the LIB operand. In the following example, *SALESLIB.LIB.SBRT2* is used to resolve external references used in *program2*.

```
LINK program2.obj LIB('CEE.SCEELKED.', 'SALESLIB.LIB.SBRT2')
```

A request coded this way searches CEE.SCEELKED and `SALESLIB.LIB.SBRT2` to resolve external references.

LOAD operand of the LINK command

In the LOAD operand, you can specify the name of the data set that is to hold the load module as follows:

```
LINK LOAD(load-mod-name(member)) LIB('CEE.SCEELKED')
```

The load module produced by the linkage editor must be a member in a partitioned data set.

If you do not specify a data set name for the load module, the system constructs a name by using the first data set name that appears after the keyword LINK, and it will be placed in a member of the *user-prefix.program-name*.`LOAD` data set. If the input data set is sequential and you do not specify a member name, TEMPNAME is used.

Example: The following example shows how to link-edit two object modules and place the resulting load module in member TEMPNAME of the *userid*.`LM.LOAD` data set.

```
LINK program1,program2 LOAD(lm)
```

You can also specify link-edit options in the link statement:

```
LINK program1 LOAD(lm) LET
```

Options for the linkage editor are discussed in “Output from the linkage editor” on page 608.

For more information about using the TSO command LINK, see *z/OS TSO/E Command Reference* .

Specifying link-edit options through the TSO LINK command

TSO users specify link-edit options through the LINK command. For example, to use the MAP, LET, and NCAL options when the object module in `SMITH.PROGRAM1.OBJ` is placed in `SMITH.PROGRAM1.LOAD(LM)`, enter:

```
LINK SMITH.PROGRAM1 'LOAD(LM) MAP LET NCAL'
```

You can use link-edit-options to display a map listing at your terminal:

```
LINK PROGRAM1 MAP PRINT(*)
```

Storing load modules in a load library

If you want to link C functions, to store them in a load library, and to INCLUDE them later with main procedures, use the NCAL and LET linkage editor options.

Prelinking and link-editing under the z/OS Shell

You can prelink and link your application under the shell by using the the OMVS prelinker option. The OMVS option causes the prelinker to change its processing of INCLUDE and LIBRARY control statements. The search library is pointed to immediately for any currently unresolved symbols. If the processing of subsequent INCLUDE or LIBRARY statements results in new or unresolved symbols, a previously encountered library will not be searched again. You may need another LIBRARY statement that points to the same library to search it again. For more information on the OMVS prelinker option, see Appendix B, “Prelinker and linkage editor options,” on page 643.

Using your JCL

The example JCL in Figure 57 links to an archive library and to z/OS data sets. Include files may be PDS members, sequential files, or z/OS UNIX System Services files. Libraries may be partitioned data sets, or archive libraries.

```
//*Add a job card to meet your system requirements
//PLINK    EXEC PGM=EDCPRLK,
//         PARM='OMVS,MEMORY,MAP,NONCAL'
//STEPLIB  DD DSN=CEE.SCEERUN,DISP=SHR
//SYSMSGS  DD DSN=CEE.SCEEMSGP(EDCPMSGE),DISP=SHR
//DDOBJ1   DD DSN=MYUSERID.OBJ(MAINPROG),DISP=SHR
//DDLIB1   DD PATH='/u/myuserid/mylibrary.a'
//SYSLIB   DD DUMMY
//SYSMOD   DD PATH='/u/myuserid/myprog.o',
//         PATHOPTS=(OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU
//SYSDEFSD DD DUMMY
//SYSOUT   DD SYSOUT=*
//SYSIN DD DATA,DLM=@@
  INCLUDE DDOBJ1
  LIBRARY DDLIB1
@@
//*-----------------------------------------------------------
//* LINK EDIT STEP:
//*-----------------------------------------------------------
//LKED   EXEC PGM=HEWL,PARM='MAP'
//SYSLIB   DD  DSN=CEE.SCEELKED,DISP=SHR
//SYSLIN   DD  PATH='/u/myuserid/myprog.o'
//SYSLMOD  DD  DSN=MYUSERID.LOAD(MEM1),DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSIN    DD  DUMMY
```

Figure 57. Using OMVS to prelink and link

The JCL in Figure 57 produces the following Prelinker Map:

```
========================================================================
|                            Prelinker Map                             |
|                                                                      |
| CPLINK:5694A01 V1 R11 M0 IBM LANGUAGE ENVIRONMENT 2009/04/29 04:55:11|
========================================================================

Command Options. . . . . : NONCAL   MEMORY   ER       DUP      MAP
                         : OMVS     NOUPCASE DYNAM


========================================================================
|                      Object Resolution Warnings                      |
========================================================================

WARNING EDC4015: Unresolved references are detected:
CEEBETBL CEEROOTA CEESG003 EDCINPL


========================================================================
|                               File Map                               |
========================================================================

*ORIGIN  FILE ID  FILE NAME

   PI     00001   /u/myuserid.OBJ(MAINPROG)
   A      00002   /u/myuserid/mylibrary.a(sumsqr.o)

*ORIGIN:  P=primary input     PI=primary INCLUDE    SI=secondary INCLUDE
          A=automatic call     R=RENAME card         L=C Library
         IN=internal


========================================================================
|                         Writable Static Map                          |
========================================================================

INFORMATIONAL EDC4013: No map displayed as no writable static was found.


========================================================================
|                  ESD Map of Defined and Long Names                   |
========================================================================

                   OUTPUT
*REASON  FILE ID  ESD NAME   INPUT NAME

          00001   CEESTART   CEESTART
          00001   CEEMAIN    CEEMAIN
          00001   MAIN       MAIN
          00002   SUMSQR     SUMSQR

*REASON: P=#pragma or reserved    S=matches short name    R=RENAME card
         L=C Library              U=UPCASE option         D=Default


============  E N D   O F   P R E - L I N K A G E   M A P  ============
```

Figure 58. Prelinker Map produced when prelinking using OMVS

Setting c89 to invoke the prelinker

The c89, c++, and cc utilities invoke the binder by default, unless the output file of the link-editing phase (-o option) is a PDS, in which case they use the prelinker.

You can set the *prefix*_STEPS environment for each of these utilities to use the prelinker for link-edit output files that are PDSEs or z/OS UNIX System Services files.

Once you set the *prefix*_STEPS environment variable for a utility so that the prelinker bit is turned on, that utility will always use the prelinker. If you want to use the binder, you must reset the *prefix*_STEPS environment variable.

For a complete description of c89, c++, and cc, see Chapter 21, “c89 — Compiler invocation using host environment variables,” on page 535. For a description of the *prefix*_STEPS environment variable, see *z/OS UNIX System Services Command Reference*.

Using the c89 utility

The c89 utility specifies default values for some prelinker and linkage editor options. It also passes prelinker options and linkage editor options by using the -W option.

c89 specifies prelinker and linkage editor options in order for it to provide the user with correct and consistent behavior. In order to determine exactly the prelinker and linkage editor options that c89 specifies, you should use the c89 -V option.

Some c89 options, such as -V, will change the settings of the prelinker options and the linkage editor options that c89 specifies. For example, when you do not specify -V, c89 specifies the prelinker option NOMAP, and when you specify -V, c89 specifies the prelinker option MAP.

To explicitly override the options that c89 specifies, use the c89 -W option. For example, to use the prelinker option MAP even when the c89 -V option is not specified, invoke

```
c89 -Wl,p,map ...
```

For a list of prelinker options and their uses, see “Prelinker options” on page 643.

Prelinker control statement processing

The only control statements that the prelinker processes are IMPORT, INCLUDE, LIBRARY, and RENAME statements. The remaining control statements remain unchanged until the link step.

You can place the control statements in the input stream, or store them in a permanent data set. If you cannot fit all of the information on one control statement, you can use one or more continuations. The long name, for example, can be split across more than one statement. You can enable continuations in one of two ways:

- Place a non-blank character in column 72 of the statement that is to be continued. The continuation must begin in column 16 of the next statement.
- Enclose the name in single quotation marks. When such a name is continued across statements, it extends up to and includes column 71. Although column 72 is not considered part of the name, it must be non-blank for the name to be continued. On the following statement, column 1 must be blank (containing the X'40' character); the name then continues in column 2.

  If you have a name that contains a single quotation mark, and you want to enclose the whole name in single quotation marks, put two single quotation marks next to each other where you want the single one to appear in the name.

  Example: If you want the name

```
SymbolNameWithAQuote'InTheMiddle
```

specify it as follows:

```
'SymbolNameWithAQuote''InTheMiddle'
```

If you mix the two style of continuation in one control statement, after you continue a statement in column 2 due to a quotation mark in the name, all subsequent statements will continue in column two.

IMPORT control statement

The IMPORT control statement has the following syntax:

```
►►──IMPORT──┬──CODE──┬─dll-name─────────┬──┬─function─────────┬─┬────────────────────►◄
            │        └─'──dll-name──'───┘  └─'──function──'───┘ │
            └──DATA──┬─dll-name─────────┬──┬─variable─────────┬─┘
                     └─'──dll-name──'───┘  └─'──variable──'───┘
```

dll-name
: The name or alias of the load module for the DLL. The maximum length of an alias is 8 characters. However, the name itself can be a long name. The *dll-name* comes from the value specified on the DLLNAME prelinker option. For more information, see "Prelinker options" on page 643.

variable
: An exported *variable name*. It is a mixed case long name. To indicate a continuation across statements, either use a non-blank character in column 72 of the card and begin the next line in column 16, or enclose the name in single quotation marks, end the first line in column 71, and put a blank character in column 1 of the next line.

function
: An exported *function name*. It is a mixed case long name. You can indicate a continuation the same way you would for a variable.

The prelinker processes IMPORT statements, but does not pass them on to the link step.

INCLUDE control statement

The INCLUDE control statement has the following syntax:

```
►►──INCLUDE──┬─ddname─────────┬──(──┬─member─────────────┬──)───────────────────►◄
             └─'──ddname──'───┘     │ ┌─,──────────────┐ │
                                    │ ▼                │ │
                                    └───'──member──'───┴─┘
```

ddname
: A *ddname* associated with a file to be included. You can use the same kinds of continuations that you can for the *variable* on the IMPORT control statement.

member
: The *member* of the DD to be included. You can use the same kinds of continuations that you can for the *variable* on the IMPORT control statement.

The prelinker processes INCLUDE statements like the z/OS linkage editor with the following exceptions:

An attempt is made to read the DD or member of the DD (whichever is specified). This request is resolved if the read is successful.
- INCLUDEs of identical member names are not allowed.
- INCLUDEs of both a ddname and a member from the same ddname are not allowed. The prelinker ignores the second INCLUDE.

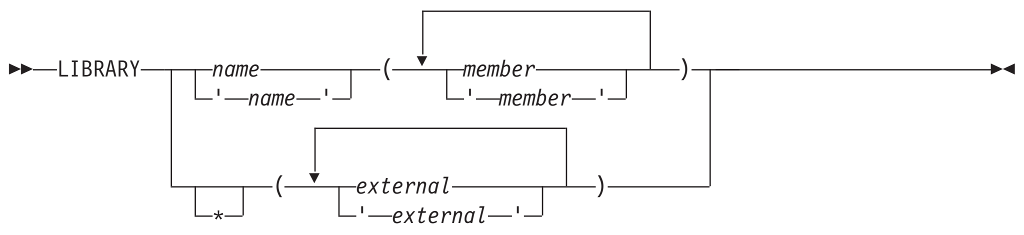
Note: The INCLUDE control statement is removed and not placed in the prelinker output object module; the system linkage editor does not see the INCLUDE control statement.

For more information on the linkage editor, refer to *z/OS MVS Program Management: User's Guide and Reference* .

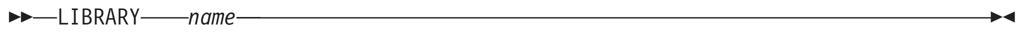
LIBRARY control statement

The LIBRARY control statement has the following syntax:

NOOMVS



OMVS



name
: the *name* of a DD that defines a library, under z/OS. This could be a concatenation of one or more libraries that are created with or without the object library utility. You can use the same kinds of continuations that you can for the *variable* on the IMPORT control statement.

member
: the name or alias of a member of the specified library. Because both short names and long names can be specified, case distinction is significant. If you use an long name, you can use the same kinds of continuations that you can for the *variable* on the IMPORT control statement.

  Under z/OS, automatic library calls search the library and each subsequent library in the concatenation, if necessary, for the name instead of searching the primary input. If you specify the OMVS option, the only form of the LIBRARY card the prelinker accepts is LIBRARY *ddname* statement in SYSLIB.

external
: an external reference that may be unresolved after primary input processing. An Automatic Library call will not resolve this external reference. Because both short names and long names can be specified, case distinction is significant. If you use an long name, you can use the same kinds of continuations that you can for the *variable* on the IMPORT control statement.

Note: The LIBRARY control statement is removed and not placed in the prelinker output object module; the system linkage editor does not see the LIBRARY control statement.

RENAME control statement

The RENAME control statement has the following syntax:

NOOMVS

```
►►──RENAME──┬─long name─────┬─short name──┬────────┬──────────────────►◄
            └─'──long name──'─┘            └─SEARCH─┘
```

OMVS

```
►►──RENAME──┬─long name─────┬─short name──────────────────────────────►◄
            └─'──long name──'─┘
```

long name
: the *name* of the long name to be renamed on output. All occurrences of this long name are renamed. You can use the same kinds of continuations that you can for the `variable` on the IMPORT control statement.

short name
: the *name* of the short name to which the long name will be changed. This name can be at most 8 characters, and case is respected.

SEARCH
: an optional parameter specifying that if the short name is undefined, the prelinker searches by an automatic library call for the definition of the short name. This is not available with the OMVS option.

The RENAME control statement is processed by the prelinker. You can use this statement to do the following:

- Explicitly override the default name that is given to an long name when an long name is mapped to a short name.

  You can explicitly control the names that are presented to the system linkage editor so that external variable and function names are consistent from one linkage editor run to the next. This consistency makes it easier to recognize control section and label names that appear in system dumps and linkage editor listings. Another mapping rule can provide the suitable name, but if you need to replace the linkage editor control section, you need to maintain consistent names. See "Mapping long names to short names" on page 605 for a description of this rule.
- Explicitly bind a long name to a short name. This binding may be necessary when linking with other languages that use a different name for the same object.

A RENAME control statement cannot be used to rename a writable static object because its name is not contained in the output from the prelinker.

You can place RENAME control statements before, between, or after other control statements or object modules. An object module can contain only RENAME statements. RENAME statements can also be placed in input that is included because of other RENAME statements.

Usage notes

- A RENAME statement is ignored if the long name is not encountered in the input.
- A RENAME statement for an long name is valid provided *all* of the following are true:
  - The long name was not already mapped because of a rule that preceded the RENAME statement rule in the hierarchy described in "Mapping long names to short names" on page 605.
  - The long name was not already mapped because of a previous valid RENAME statement for the long name.
  - The short name is not itself an long name. This rule holds true even if the short name has its own RENAME statement.
  - A previous valid RENAME statement did not rename another long name to the same short name.
  - Either the long name or the short name is not defined. Either the long name or the short name can be defined, but not both. This rule holds true even if the short name has its own RENAME statement.

Reentrancy

This information discusses how to use the prelinker to make your program reentrant. For detailed information on reentrancy, see *z/OS XL C/C++ Programming Guide*.

Reentrant programs are structured to allow more than one user to share a single copy of a load module or to use a load module repeatedly without reloading it.

Natural or constructed reentrancy

Reentrant programs can be categorized as having natural or constructed reentrancy. Programs that contain no references to writable static objects have natural reentrancy. Programs that refer to writable static objects must be processed with the IBM Language Environment Prelinker to make them reentrant; such programs have constructed reentrancy.

If you are using C, you do not need to use the RENT compiler option if your program is naturally reentrant.

Because all C++ programs are categorized as having constructed reentrancy, C++ code must be bound by the binder using the DYNAM(DLL) option. Alternatively, the C++ code must be processed by the prelinker before being processed by the linkage editor.

Using the prelinker to make your program reentrant

The prelinker concatenates compile-time initialization information (for writable static) from one or more object modules into a single initialization unit. In the process, the writable static part is mapped.

If you are not using the binder, and your program contains writable static, you can use the prelinker to make your program reentrant. If the program is C and does not contain writable static, you do not need to use the prelinker to ensure reentrancy; the program is naturally reentrant. C++ programs always contain writable static.

If you compile your code and wish to link it using the z/OS system link procedures such as IEWL, you must first call the prelinker.

The z/OS UNIX System Services features require that all z/OS XL C/C++ application programs be reentrant. If you are using the c89 utility, it automatically invokes the z/OS XL C/C++ compiler with the RENT option and also invokes the prelinker.

The prelinker is not a post-compiler. That is, you do not prelink the object modules individually into separate prelinked object modules as if running the prelinker was an extension of the compile step. Instead, you prelink all the object modules together in the same job into one output prelinked object module. This is because the prelinker cannot process each object deck one at a time: it assigns offsets to each data item in the writable static area for the program, and thus needs all of the object decks that refer to data items in writable static input in a single step.

The prelinker does all of the following:

- It maps input long names from the object modules to output short names (8 characters maximum)
- It collects compile-time initialization information on static objects
- It collects constructor calls and destructor calls for static objects in C++
- It collects DLL information
- It collects objects that exist in writable static into one area by assigning an offset within the writable static area to each object
- It removes all relocation and name information of objects in the writable static area

The output of the prelinker is a single prelinked object module. You can link this object module only on the same platform where you prelinked it.

Because the prelinker maps names and removes the relocation information, you cannot use the resulting object module as input for another prelink. Also, you cannot use the linkage editor to replace a control section (CSECT) that either defines or references writable static objects.

Steps for generating a reentrant load module in C

Perform the following steps to generate a reentrant load module in C:

1. Determine whether or not your program contains writable static. If you are unsure about whether your program contains writable static, compile it with the RENT option. Invoking the prelinker with the MAP option and the object module as input produces a Prelinker Map. Any writable static data in the object module appears in the writable static section of the map. Unresolved writable static references may also appear in the map as errors.

   If you see the symbol @STATIC defined in the writable static section, your code contains unnamed writable static such as modifiable literal strings, or variables with the static qualifier. To make literal strings stay in the code area, recompile with **#pragma strings(readonly)**, and prelink again.

2. If your program contains no writable static, compile your program as you would normally (without any special compiler options), and then go directly to step 4.

3. If your program contains writable static, you must compile your C source files with the RENT compiler option.

4. Use the Language Environment prelinker to combine *all* input object modules into a single output object module.

   Notes:

   a. The prelinker can handle compiled programs in languages other than C or C++. However, only C, C++, OO COBOL, or assembler code using the macros EDCDXD and EDCLA may refer to writable static.
   b. You cannot use the output object module as further input to the Language Environment prelinker.

5. Optionally, you can use the output object module to link the program in the LPA or ELPA area of the system.

6. Under the z/OS shell, you can run the installed program by invoking it from z/OS UNIX System Services. To do so you must install the program in z/OS UNIX, and, from a superuser ID, enter a **chmod** Shell command to turn on the sticky bit for the program. See *z/OS UNIX System Services Planning* for more information.

Steps for generating a reentrant load module in C++

Perform the following steps to generate a reentrant load module in C++:

1. Compile your source code.

   If you see the symbol @STATIC defined in the writable static section, your code contains unnamed writable static such as modifiable literal strings, or variables with the static qualifier. To make literal strings stay in the code area, recompile with **#pragma strings(readonly)**, and prelink again.

2. Use the supplied prelink and link utilities on the module. Under TSO, you can use the CXXMOD REXX EXEC to both prelink and link your module. Under z/OS batch, use these JCL procedures:
   - CBCCL: compile and link
   - CBCL: link
   - CBCCLG: compile, link, and go
   - CBCLG: link and go

   For all of these, linking involves two steps: invocation of the prelinker, and then a call to the system linker.

Resolving multiple definitions of the same template function

Note: For complete information on using C++ templates, see *z/OS XL C/C++ Programming Guide*.

When the prelinker generates template functions, it resolves multiple function definitions as follows:

- If a function has both a specialization and a generalization, the specialization takes precedence.
- If there is more than one specialization, the prelinker issues a warning message.

Because the link step does not remove unused instantiations from the executable program, instantiating the same functions in multiple compilation units may generate very large executable programs.

External variables

For more information on external variables, see *z/OS XL C/C++ Programming Guide*.

The POSIX 1003.1 and X/Open CAE Specification 4.2 (XPG4.2) require that the C system header files declare certain external (global) variables. Additional variables are defined for use with POSIX or XPG4.2 functions. If you define one of the POSIX or XPG4 feature test macros and include one of these headers, the global variables will be declared in your program. These global variables are treated differently than other global variables in a multi-threaded environment (values are thread-specific rather than global to the process) and across a call to a fetched module (values are propagated rather than module-specific). To access the global variables, you must use either C with the RENT compiler option, C++, or the XPLINK compiler option. If you are not using XPLINK, you must also specify the SCEEOBJ autocall library. The SCEEOBJ library must be specified before the SCEELKEX and the SCEELKED libraries in the bind step. If the SCEEOBJ library is specified after the SCEELKEX and SCEELKED libraries, the bind step will resolve the external variables to the user application, but the Language Environment run time will not use those same external variables, and so run-time errors can occur. You are also able to access the external variables by defining the _SHARE_EXT_VARS feature test macro during the compile step (or the _SHR_name feature test macro corresponding to the variable names you are accessing). For further information on feature test macros, see *z/OS XL C/C++ Run-Time Library Reference*. In this case, functions which access the thread-specific values of the external variables are provided for use in a multi-threaded environment. If you use the XPLINK compiler option for a 32-bit program, the global variables are resolved by import using the CELHS003 member of the SCEELIB data set. The thread-specific values are always used.

For a dynamically called DLL module to share access to the POSIX external variables, with its caller, the DLL module must define the _SHARE_EXT_VARS feature test macro. For more information, see the section on feature test macros in the *z/OS XL C/C++ Run-Time Library Reference*.

Appendix B. Prelinker and linkage editor options

This information contains the prelink options and link options for your programs, which are provided by the Language Environment services. For more information on using the Language Environment Prelinker, see Appendix A, “Prelinking and linking z/OS XL C/C++ programs,” on page 601.

Prelinker options

The following information describes the prelink options available in z/OS XL C/C++ by using Language Environment services.

DLLNAME(dll-name)

DLLNAME specifies the DLL name that appears on generated IMPORT control statements, described in “IMPORT control statement” on page 635. If you specify the DLLNAME option, the prelinker sets the DLL name to the value that you listed on the option.

If you do not specify DLLNAME, the prelinker sets the DLL name to the name that appeared on the last NAME control statement that it processed. If there are no NAME control statements, and the output object module of the prelinker is a PDS member, it sets the DLL name to the name of that member. Otherwise, the prelinker sets the DLL name to the value TEMPNAME, and issues a warning.

DUP | NODUP

DEFAULT: DUP

DUP specifies that if duplicate symbols are detected, their names should be directed to the console, and the return code minimally set to a warning level of 4. NODUP does not affect the return code setting when the prelinker detects duplicates.

DYNAM | NODYNAM

DEFAULT: DYNAM

When the NODYNAM option is in effect, export symbol processing is not performed by the prelinker even when export symbols are present in the input objects. The side-deck is not created and the resulting module will not be a DLL. Specify NODYNAM for prelinked C/C++ programs involved in COBOL C/C++ ILC calls.

ER | NOER

DEFAULT: ER

Note: For the z/OS UNIX Systems Services environment, the default is NOER.

If there are unresolved symbols, ER instructs the prelinker to write a messages and a list of unresolved symbols to the console. If there are unresolved references, the prelinker sets the return code to a minimum warning level of 4. If there are unresolved writable static references, the prelinker sets the return code to a minimum error level of 8. If you use NOER, the prelinker does not write the list of unresolved symbols to the console. If there are unresolved references, the return code is not affected. If there are unresolved writable static references, prelinker sets the return code to a minimum warning level of 4.

© Copyright IBM Corp. 1998, 2009

MAP | NOMAP

DEFAULT: MAP

In the z/OS UNIX System Services environment, the c89, cc, and c++ utilities specify MAP when you use the -V flag, and NOMAP when you do not.

The MAP option specifies that the prelinker should generate a prelink listing. See "Language Environment Prelinker Map" on page 616 for a description of the map.

MEMORY | NOMEMORY

DEFAULT: NOMEMORY

The MEMORY option instructs the prelinker to retain in storage, for the duration of the prelink step, those object modules that it reads and processes.

You can use the MEMORY option to increase prelinker speed. However, you may require additional memory to use this option. If you use MEMORY and the prelink fails because of a storage error, you must increase your storage size or use the prelinker without the MEMORY option.

NCAL | NONCAL

DEFAULT: NONCAL

The NCAL option specifies that the prelinker should not use the automatic library call to resolve unresolved references.

The prelinker performs an automatic library call when you specify the NONCAL option. An automatic library call applies to a library of user routines. For NOOMVS, the data set must be partitioned, but for OMVS the data set that the prelinker searches can be either a PDS or an archive library. Automatic library call cannot apply to a library that contains load modules.

Note: If you are prelinking C++ object modules, you must use the NONCAL option and include the C++ base library in the CEE.SCEECPP data set in your SYSLIB concatenation.

OMVS | NOOMVS

DEFAULT: NOOMVS

The OMVS option causes the prelinker to change the way that it processes INCLUDE and LIBRARY control statements. The c89 utility turns on the OE option (which maps to the OMVS option) by default. Object files and object libraries from c89 are passed as primary input to the prelinker. Object files are passed via INCLUDE control statements, and object libraries via LIBRARY control statements. Only those LIBRARY control statements that are included in primary input are accepted by the prelinker. Their syntax is:

```
LIBRARY libname
```

where *libname* is the name of a DD that defines a library. The library may be either an archive file created through the `ar` utility or a partitioned data set (PDS) with object modules as members. The prelinker uses LIBRARY control statements like SYSLIBs, to resolve symbols through autocalls.

When you specify the OMVS option, the prelinker accepts INCLUDE and LIBRARY statements which refer to z/OS UNIX System Services files (PATH=) and data set name (DSNAME=) allocations.

When you use the OMVS option, the order in which object files and object libraries are passed is significant. The prelinker processes its primary input sequentially. It searches the library that you specified on the LIBRARY statement only at the point where it encounters the LIBRARY statement. It does not refer to that library or processes it again. For example, if you pass your object files and object libraries as follows:

```
c89 file1.o lib1.a file2.o lib2.a
```

The prelinker processes the INCLUDE control statement for `file1.o`, and incorporates new symbol definitions and unresolved references from the object file into the output file. The prelinker then processes the LIBRARY control statement for `lib1.a`, and searches the library for currently unresolved symbols. It then processes `file2.o` followed by `lib2.a`. If the processing of `file2.o` results in unresolved symbols, the prelinker will not search the library `lib1.a` again, because it has already processed it. If you have unresolved symbols that may be defined in a library that has already been processed, you must specify a new LIBRARY statement after your `INCLUDE` statement to resolve those symbols. You can do this on a **c89** command line as follows:

```
c89 file1.o lib1.a file2.o lib1.a lib2.a
```

RENAME control statements are processed on output from the prelinker, after all of its input has been processed. Because a library can be processed once only, the SEARCH option on the RENAME control statement has no effect.

Note: The OE prelinker option maps to the OMVS prelinker option.

UPCASE | NOUPCASE

DEFAULT: NOUPCASE

The UPCASE option enforces the uppercase mapping of long names that are 8 characters or fewer and have not been explicitly mapped by another mechanism. These long names are uppercased (with _ mapped to `@`), and names that begin with IBM or CEE are changed to `IB$` and `CE$`, respectively.

The UPCASE option is useful when calling routines that are written in languages other than z/OS XL C/C++. For example, in COBOL and assembler, all external names are in uppercase. So, if the names are coded in lowercase in the z/OS XL C/C++ program and you use the LONGNAME option, the names will not match by default. You can use the UPCASE option to enforce this matching. You can also use the RENAME control statement for this purpose.

Note: Use of this option can be dangerous, since names with a length of 8 characters or less will lose their case sensitivity. A better way to get the linkage and names correct is through the use of the appropriate pragmas.

Linkage editor options

You can specify link-edit options in either of two ways:
- Through JCL
- Through the TSO LINK command

For a description of link-edit options, see Chapter 5, “Binder options and control statements,” on page 331 or *z/OS MVS Program Management: User’s Guide and Reference*.

Appendix C. Diagnosing problems

This information tells you how to diagnose failures in the z/OS XL C/C++ compiler. If you discover that the problem is a valid compiler problem, refer to techsupport.services.ibm.com/guides/handbook.html for further information on obtaining IBM service and support.

Problem checklist

The following list contains suggestions to help you rule out some common sources of problems.

- Check that the program has not changed since you last compiled or executed it successfully. If it has, examine the changes. If the error occurs in the changed code and you cannot correct it, note the change that caused the error. Whenever possible, you should retain copies of both the original and the changed source programs.
- Be sure to correct all problems that are diagnosed by error messages, and ensure that the messages that were previously generated have no correlation to the current problem. Be sure to pay attention to warning messages.
- The message prefix can identify the system or subsystem that issued the message. This can help you determine the cause of the problem. Following are some of the prefixes and their origins.
  - CCN - indicates messages from the z/OS XL C/C++ compiler, its utility components, or the z/OS XL C/C++ IPA Link step. Information on the messages is found in *z/OS XL C/C++ Messages*.
  - EDC - a numeric portion between 0090 and 0096 indicates a *severe error*, and the solution should be self-evident from the accompanying text. If it is not, contact your Service Representative. If the numeric portion is in the 4000 series, this specifically relates to the prelinker and *alias* utility. Otherwise, the message relates to the z/OS XL C/C++-specific messages from the run-time environment. Information on Language Environment messages is found in *z/OS Language Environment Run-Time Messages*.
  - CEE - for language-independent messages from the common execution environment (CEE) Language Environment library component. Information on Language Environment messages is found in *z/OS Language Environment Run-Time Messages*.
  - IBM, PLI, IGZ - for language-specific Language Environment messages. Information on Language Environment messages is found in *z/OS Language Environment Run-Time Messages*.
  - CLB - for messages that relate to z/OS class libraries. See *z/OS XL C/C++ Messages* for more information.
  - BPX - messages that relate to z/OS UNIX System Services.
  - FSUM - messages for the c89 and xlc utilities.

  You can cross reference the prefix to the message document in most cases by using the table at the beginning of the *z/OS MVS System Messages* volumes which accompany the z/OS operating system.
- Ensure that you are compiling the correct version of the source code. It is possible that you have incorrectly indicated the location of your source file. For example, check your high-level qualifiers.
- In any program failure, keep a record of the conditions and options in effect at the time the problem occurred. The listing file shows the options. To get the

listing, compile with the SOURCE option. The listing only contains options that appear after the command line is processed, hence C **#pragma options** do not appear.

Information about some of the options appears as a comment at the end of the object file. For both C or C++ compiles, there is always a comment showing the OPTIMIZE level. For C compiles, information about some of the options (for example, ALIAS, GONUMBER, INLINE, RENT, or UPCONV options) is included only if you specify the option when you compile. Note any changes from the previous compilation.

- Your installation may have received an IBM Program Temporary Fix (PTF) for the problem. Verify that you have received all issued PTFs and have installed them, so that your installation is at the current maintenance level. Specifying the compiler option PHASEID when doing a compile provides information about the maintenance level of each compiler component (phase).
- The preventive service planning (PSP) bucket, which is an online database available to IBM customers through IBM service channels. It gives information about product installation problems and other problems. See the z/OS Program Directory for more details.
- Use the Debug Tool, dbx (for z/OS UNIX System Services) or some other debugging aid to determine the statement where the program fails and possible causes of the failure.
- If a failing application is communicating with other IBM products, make sure that it uses the correct interface procedure as documented in *z/OS XL C/C++ Programming Guide*. In many cases, you can localize the failing condition by taking out the function calls or making them no-ops.
- If your application has been developed on a different platform (such as a microcomputer or workstation) and you try to compile and run using the z/OS XL C/C++ compiler, the following may cause problems:
  – The source code does not support the applicable following standards:
    - *ISO C Standard*
    - *ISO C++ Standard*
  – The source code includes dependencies on the ASCII character set or uses the long double data type in the IEEE floating-point format. You need the ASCII compiler option to process the ASCII characters, and you need the FLOAT(IEEE) option to process IEEE floating-point data types. Note that the IEEE long double data types may have different sizes on a different platform.
  – The source code is system dependent
- If your application was prelinked, make sure that the prelinking was successful as indicated in Appendix A, “Prelinking and linking z/OS XL C/C++ programs,” on page 601.

When does the error occur?

Determine when the problem is occurring (at compile time, bind time, prelink time, link time or run time), and use the procedures in the appropriate list on the following pages. If the problem occurs when using Language Environment services, for prelink-time and run-time diagnosis and debugging errors you should use *z/OS Language Environment Customization* and *z/OS Language Environment Debugging Guide*. For bind-time and link-time diagnosis, refer to *z/OS MVS Program Management: User’s Guide and Reference*.

After you identify the failure, you can write a small, self-contained test case that does not have any dependencies on third-party header files and libraries and that recreates the problem. A test case helps you to isolate the problem and to report problems to IBM.

To create a small test case from a large program that appears to be failing, try the suggestions listed below, after you have either backed up or made a copy of your original source code. Begin with the suggestion that seems most appropriate for the problem that you are having. If the problem persists after you have tried one of the steps below, try another in the list. Continue to break your program down until you obtain the smallest possible segment of code that still reproduces the error. Compile with the PPONLY option and send the expanded file as your source code. This is so that all embedded header files are included. Save this last failing test case because you might need it if you have to contact an IBM Support Center.

Remove any code that has not been processed at the time of failure (except for code necessary to ensure the syntactic and semantic validity of the program).

Find unreferenced variables using the IPA(XREF) option, the CHECKOUT(GEN) option, which is for C only, or the INFO(USE) option, which is for C++ only, and remove the unreferenced variables.

Remove all code and declarations from the body of any other functions that are not necessary to reproduce the problem. The function should be removed if it is not necessary.

If your program uses structure variables, try replacing them with scalar variables.

Steps for problem diagnosis using optimization levels

Before you begin: For diagnostic purposes, you should always begin by using the simplest optimization level on your program. Once you address all problems at your current level, progress toward the more complex levels of optimization.

Perform the following steps to progress through the various levels of optimization:

1. Begin with a non-IPA compile and link using progressively higher levels of optimization:
   - OPT(0)
   - OPT(2)
   - OPT(3)

   If your program works successfully at OPT(0) and fails at OPT(2), try rebuilding the program specifying the compiler option NOANSIALIAS and re-running. You may suffer a performance penalty for this as the optimizer has to make worst-case aliasing assumptions but it may resolve the problem.

2. Use IPA(OBJONLY) and OPT(2). This adds the IPA compile-time optimizations and often locates the problematic source file before you invest a lot of time and effort diagnosing problems in your code at IPA Link time.

3. Use the full IPA Compile and IPA(Level(1)) Link path. IPA Compile-time optimizations are performed on the IPA object. IPA Link-time optimizations are performed on the entire application.

4. Use the full IPA Compile and IPA(Level(2)) Link path. IPA Level 2 performs additional link-time optimizations.

You know you are done when you have exploited all optimizations offered by the compiler.

Steps for diagnosing errors that occur at compile time

Perform the following steps to diagnose errors that occur at compile time:

1. If your program uses any of the library routines, insert an `#include` directive for the appropriate header files. Also insert an `#include` directive for any of your own header files. The compiler uses function prototypes, when present, to help detect type mismatches on function calls. You can use the C CHECKOUT option to find missing prototyping. Note that z/OS XL C++ does not allow missing prototypes.

---

2. Compile your program with the INFO option (or you can use the CHECKOUT option as an alternative for C only). These options specify that the compiler is to give informational messages that indicate possible programming errors. These options will give messages about such things as variables that are never used, and the tracing of `#include` files.

---

3. Compile your program with the PPONLY option to see the results of all `#define` and `#include` statements. This option also expands all macros; a macro may have a different result from the one you intended.

---

4. If your program was originally compiled using the OPT(2) compiler option, try to recompile it using the NOOPTIMIZE option, and run it. If you can successfully compile and run the program with NOOPTIMIZE, you have bypassed the problem, but not solved it. This does not however, exclude the possibility of an error in your program. You can run the program as a temporary measure, until you find a permanent solution. If your program works successfully at OPT(0) and fails at OPT(2), try rebuilding the program specifying the compiler option NOANSIALIAS and re-running. You may suffer a performance penalty for this as the optimizer has to make worst-case aliasing assumptions but it may resolve the problem.

---

5. If you compiled your program with either the SEQUENCE or the MARGINS option, the error may be due to a loss of code. If you compiled the source code with the NOSEQUENCE option, the compiler will try to parse the sequence numbers as code, often with surprising results. This can happen in a source file that was meant to be compiled with margins but was actually compiled without margins or different margins (available in z/OS XL C only).

   Either oversight could result in syntax errors or unexpected results when your program runs. Try recompiling the program with either the NOSEQUENCE or the NOMARGINS option.

---

6. Your source file may contain characters that are not supported by your terminal. You have two options at this point:

a. Replace any characters that cannot be displayed in literals with the corresponding digraph (specify the DIGRAPH compiler option), or trigraph representation, or the corresponding escape sequence. Verify that the error did not result from using one of these incorrectly.
b. You can use the **#pragma filetag** support and the LOCALE option to allow the compiler to work with code pages other than the default IBM-1047. See *z/OS XL C/C++ Language Reference* for more details on **#pragma filetag**.

---

7. Check for duplicate static constructors and destructors in your C++ source. Entries for constructors are created in the object and in a table. When a static constructor is removed, the entry in the object is removed, but the table entry stays. This will cause the static constructor and destructor to be called multiple times. If the destructor deletes (or frees) dynamically allocated storage that is associated with a pointer, it will tend to fail on subsequent invocations.

---

8. A compile-time abend can indicate an error in the compiler. An unsuccessful compilation due to an error in the source code or an error from the operating system should result in error messages, not an abend. However, the cause of the compiler failure may be a syntax error or an error from the operating system. Use the PHASEID compiler option to obtain the maintenance service level of the compiler, as well as the name of the failing compiler component, in the output listing.

---

9. The use of an inappropriate compiler option at compile time may cause run-time errors. To ensure that all compiler options were appropriate for compiling all source files, check the Saved Option String information in the executable. For more information on Saved Option String, see *z/OS XL C/C++ Programming Guide*.

If you still have a compilation problem, contact IBM support.

Steps for diagnosing errors that occur at IPA Link time

Perform the following steps to diagnose errors that occur at IPA Link time:

1. Ensure that the region that is used for the IPA Link step is sufficient. The REGION system parameter should be set to at least 1500 MB and the MEMLIMIT system parameter should be set to at least 3000 MB. For further information, see “Specifying compiler options under z/OS UNIX System Services” on page 45.

---

2. Ensure that the object module which defines `main()` contains an IPA object.

---

3. Ensure that all application program parts (object modules, load modules) and all necessary interface libraries (Language Environment object modules and load module, SQL, CICS, etc) are made available to the IPA Link step.

---

4. Ensure that the IPA Compile step has processed all object modules for which source is available.

---

5. Use the IPA(LINK,MAP) option to obtain an IPA Link listing.

6. Do not attempt to IPA Link unsupported file formats, such as Program Objects.

7. Verify that there are no unresolved symbol references. All user symbols must be resolved before invoking the binder (or prelinker and linkage editor). Any run-time symbol references generated by IPA Link must be resolved by the subsequent step to that no unresolved symbols remain.

   If you have unresolved symbols, make sure that the definition of an object and all its references are used consistently in both the code area and the writable static area. Also, make sure that symbol references appear consistently in the same case.

   If you have unresolved symbols after using autocall, and you are searching for longnamed or writable static objects, make sure that each object module library has a current directory generated by the C370LIB utility. Without this directory, autocall can only be done on the member name of the object module and not on what is actually defined within the member.

8. If problems occur during IPA Link processing of DLL code, note that a symbol can only be imported if all of the following conditions hold true:
   - The symbol remains unresolved after `autocall`
   - Only DLL references were seen for the symbol
   - An IMPORT control statement was encountered for the symbol

9. A compiler ABEND during IPA Link step processing can indicate an error in the compiler. An unsuccessful IPA Link due to an error in the program source code, an invalid object module, an invalid load module, or an error from the operating system should result in error messages, not an ABEND.

   If the compiler ABEND during IPA Link step processing is related to an invalid IPA object module, it will require further diagnosis:
   - Save and recompile any IPA object modules created by a previous release. If the problem is corrected, contact IBM service and be prepared to supply the relevant source (PPONLY) and IPA object modules. Try compiling at OPT(2), and then OPT(2) plus IPA(OBJONLY). If you are linking with IPA Level 2, try linking with Level 1. Ensure that you have first tried lower optimization levels. Perform a binary search for the invalid IPA object module. To do this, compile one half of your source files with NOIPA (with or without OBJONLY), and the other half with IPA. When the IPA Link succeeds, reduce the set of NOIPA objects until you identify the compilation unit which produced the invalid IPA objects. Note that the object module which defines `main()` must always contain IPA object. It may be necessary to break the source file with `main()` into multiple pieces to determine the point of failure.

You should now have a clean IPA Link compilation. If you still have a problem with the IPA Link step, contact IBM support.

The error occurs at bind time

For information on bind-time errors, see “Error recovery” on page 448.

The error occurs at prelink time

If the error occurs at prelink time:

- Do not prelink the object modules separately.
- Use the prelinker option MAP to obtain a full map of input data sets and symbols.
- Use the prelinker options DUP and ER to obtain a full list of duplicate and unresolved symbols.
- If you have unresolved symbols, make sure that the definition of an object and all references to that object are used consistently in both the code area and the writable static area. Also, make sure that symbol references appear consistently in the same case.
- A symbol can only be imported if all of the following conditions hold true:
  - The symbol remains unresolved after `autocall`.
  - Only DLL references were seen for the symbol.
  - An IMPORT control statement was encountered for the symbol.

  For more information on using DLL, see "Using DLLs" on page 611, or the DLL description in *z/OS XL C/C++ Programming Guide*.
- If you have unresolved symbols after using `autocall`, make sure that the libraries that are searched contain only object modules and no load modules. If you are searching for longnamed or writable static objects, make sure that each library has a current directory member generated by the C370LIB utility. Without this directory, `autocall` can only be done on the member name of the object module and not on what is actually defined within the member.
- Only naturally reentrant code can be linked with the output of the prelinker. For more information on reentrancy, see *z/OS XL C/C++ Programming Guide*.

The error occurs at link time

If the error occurs at link time:

- If you have a link-time error while working with XL C/C++ using Language Environment services, you can find diagnostics and debugging information in *z/OS MVS Program Management: User's Guide and Reference* .
- If you have a link-time error while working with the common execution environment (CEE) Language Environmentlibrary component, you can find diagnostics and debugging information for link-time errors in *z/OS Language Environment Customization* and *z/OS Language Environment Debugging Guide*.

Steps for diagnosing errors that occur at run time

Before you begin: If you are diagnosing run-time errors when executing with Language Environment services, refer to *z/OS Language Environment Customization* and *z/OS Language Environment Debugging Guide*.

Perform the following steps to diagnose errors that occur at run time:

1. Specify one or more of the following compiler options, in addition to the options originally specified, to produce the most diagnostic information:

Option	**Information produced**
AGGREGATE	(C only). Aggregate layout.
ATTRIBUTE	(C++ only). Cross reference listing with attribute information.
CHECKOUT	(C only). Indication of possible programming errors.
DEBUG	Instructs the compiler to generate debug information based on the DWARF Version 3 debugging information format, which has

	been developed by the UNIX International Programming Languages Special Interest Group (SIG), and is an industry standard format.
EXPMAC	Macro expansions with the original source.
FLAG	Specifies the minimum message severity level that you want returned from the compiler.
GONUMBER	Generates line number information that corresponds to input source files.
INFO	(C++ only). Indication of possible programming errors.
INLINE	Inline Summary and Detailed Call Structure Reports. (Specify with the REPORT suboption.)
INLRPT	Generates a report on status of functions that were inlined. The OPTIMIZE option must also be specified.
LIST	Listing of the pseudo-assembly listing produced by the compiler.
OFFSET	Offset addresses of functions in the listing.
PPONLY	Completely expanded C or C++ source code, by activating the preprocessor (PPONLY). The output shows, for example, all the `#include` and `#define` directives.
SHOWINC	All included text in the listing.
SOURCE	Listing of the source file.
TEST	For 31-bit only, used to obtain information about the contents of variables at the point of the error, and to enable the use of Debug Tool.
XREF	Cross reference listing with reference, definition, and modification information. If you specify ATTRIBUTE, the listing also contains attribute information.

2. If the failure is in a statement that can be isolated, for example, an `if`, `switch`, `for`, `while`, or `do-while` statement, try placing the failing statement in the mainline code. If the problem is occurring as a result of a `switch` statement, make sure that you have "`breaks`" on all appropriate statements.

3. If you have used the compiler options RENT or NORENT in **#pragma options** or **#pragma variable** statements, and compiled your program at OPT(2), you can detect a possible pointer initialization error by compiling your program at OPT(0).

4. Check if you are running IBM C/370™ Version 1 or Version 2 modules. Some IBM C/370 Version 1 and Version 2 modules may not be compatible with the Language Environment element. In some cases, old and new modules that run separately may not run together. You may need to recompile or relink the old modules, or change their source. *z/OS XL C/C++ Compiler and Run-Time Migration Guide for the Application Programmer* documents these solutions.

5. If IPA Link processed the program:
   a. Ensure that the program functions correctly when compiled NOIPA at the same OPT level.
   b. Subprograms (functions and C++ methods) which are not referenced will be removed unless appropriate ″retain″ directives are present in the IPA Link control file.
   c. IPA Link may expose existing problems in the program:

- Ensure that any coalesced global variables which are character strings have sufficient space to contain all characters plus an additional byte for the terminating null.
- Ensure that there are no dependencies on the order in which data items or subprograms (functions, C++ methods) are generated.

d. Do the following to check for a code generation problem:
   - Specify a different OPT level during IPA Link processing. If the program executes correctly, contact IBM service and be prepared to supply the relevant source (PPONLY) and object modules.
   - Specify the option NOOPT during IPA Link processing. If the program executes correctly, contact IBM service and be prepared to supply the relevant source (PPONLY) and object modules.

   If the program executes correctly at a different OPT level or NOOPT, perform a binary search for the IPA object file which contains the function for which code is incorrectly generated. Contact IBM service and be prepared to supply the relevant source (PPONLY) and object modules.

e. Do the following to check for an IPA optimization problem:
   - Specify NOINLINE IPA(LEVEL(1)) during IPA Link processing.

     If the program executes correctly, perform a binary search using INLINE IPA(LEVEL(1)) for the IPA object file which contains the function which is incorrectly optimized. Once you have located the IPA object file with the problem, use `noinline` directives within the IPA Link control file to determine the functions that are not correctly inlined. Contact IBM service and be prepared to supply the relevant source (PPONLY) and object modules and the IPA Link control file.

     Functions that are inconsistently prototyped may cause problems of this type. Verify that all interfaces are consistent and complete.
   - Specify IPA(LEVEL(0)) during IPA Link processing.

     If the program executes correctly, perform a binary search using INLINE IPA(LEVEL(1)) for the IPA object file which contains the function which is incorrectly optimized. Contact IBM service and be prepared to supply the relevant source (PPONLY) and object modules.
   - Specify IPA(LEVEL(1)) instead of IPA(LEVEL(2))

     If you are linking with IPA Level 2, try linking with Level 1.

At this point, if you still encounter problems that you think are the result of the compilation, contact IBM support.

Steps for avoiding installation problems

Perform the following steps to avoid or solve most installation problems:

1. Review the step-by-step installation procedure that is documented in the z/OS Program Directory that is applicable to your environment.

---

2. Consult the PSP bucket as described in “Problem checklist” on page 647.

If you still cannot solve the problem, contact your IBM Support Center.

You may need to reinstall the z/OS XL C/C++ product by using the procedure that is documented in the z/OS Program Directory. This procedure is tested for each product release and successfully installs the product.

Appendix D. Calling the z/OS XL C/C++ compiler from assembler

To invoke the compiler dynamically under z/OS, you can use macro instructions such as ATTACH, LINK, or CALL in an assembly language program. For complete information on these macro instructions, refer to the list of documents in *z/OS Information Roadmap*.

The following is the syntax of each macro instruction:
where:

```
►►─┬───────┬─ATTACH──EP=CCNDRVR──,──PARAM=──(──────────────────────────►
   └─label─┘

►─option_list─┬────────────────┬────────────────────────────────────────►
              └─,──ddname_list─┘

►─)──,──VL=1──,──DCB=dcb_addr──,──TASKLIB=dcb_addr───────────────────►◄
```

```
►►─┬───────┬─LINK──EP=CCNDRVR──,──PARAM=──(──option_list─┬────────────────┬──►
   └─label─┘                                              └─,──ddname_list─┘

►─)──,──VL=1─────────────────────────────────────────────────────────────►◄
```

```
►►─┬───────┬─CALL──EP=CCNDRVR──,──(──option_list─┬────────────────┬──────►
   └─label─┘                                      └─,──ddname_list─┘

►─)──,──VL───────────────────────────────────────────────────────────────►◄
```

EP	Specifies the symbolic name of the z/OS XL C/C++ compiler CCNDRVR. The control program determines the entry point at which execution is to begin.
PARAM	Specifies a list that contains the addresses of the parameters to be passed to the z/OS XL C/C++ compiler
option_list	Specifies the address of a list that contains the options that you want to use for the compilation. The *option_list* must begin on a halfword boundary. The first 2 bytes must contain a count of the number of bytes in the remainder of the list. You specify the options in the same manner as you would on a JCL job, with spaces between options. If you do not want to specify any options, the count must be zero. For C++ compiler invocation, you must include the characters CXX, and a blank before the list of compiler options. The number of bytes therefore should be 4 bytes longer.
ddname_list	Specifies the address of a list that contains alternative ddnames for the data sets that are used during the compiler processing. If you use standard ddnames, you can omit this parameter. The *ddname_list* must begin on a halfword boundary. The first two bytes must contain a count of the number of bytes in the remainder of the list. You must left-justify each name in the list, and pad it with blanks to a length of 8 bytes.

The sequence of ddnames in the list is:
- SYSIN
- SYSLIN
- SYSMSGS - this ddname is no longer used, but is kept in the list for compatibility with old assembler macros.
- SYSLIB
- USERLIB
- SYSPRINT
- SYSCPRT
- SYSPUNCH
- SYSUT1
- SYSUT4
- SYSUT5
- SYSUT6
- SYSUT7
- SYSUT8
- SYSUT9
- SYSUT10
- SYSUT14
- SYSUT15
- SYSEVENT
- TEMPINC
- IPACNTL
- SYSUT16
- SYSUT17

You can omit an alternative ddname from this list by entering binary zeros in its 8-byte entry, or if it is at the end of this list, by shortening the list. If you omit a ddname, the compiler will use the appropriate default ddname from this list.

VL **or** VL=1	Specifies that the sign bit is to be set to 1 in the last fullword of the address parameter.
DCB	Specifies the address of the control block for the partitioned data set that contains the compiler.
TASKLIB	Specifies the address of the DCB for the library that is to be used as the attached tasks library.

The return code from the compiler is returned in register 15.

If you code the macro instructions incorrectly, the compiler is not invoked, and the return code is 32. This error could be caused if the count of bytes in the alternative ddnames list is not a multiple of 8, or is not between 0 to 184.

If you specify an alternative ddname for SYSPRINT, the `stdout` stream is redirected to refer to the alternate ddname.

The following examples show the use of three assembler macros that rename ddnames completely or partially. Following each macro is the JCL that is used to invoke it.

Example of using the assembler ATTACH macro (CCNUAAP)

```
***********************************************************************
*                                                                     *
*  This assembler routine demonstrates DD Name renaming               *
*  (Dynamic compilation) using the Assembler ATTACH macro.            *
*                                                                     *
*  In this specific scenario all the DDNAMES are renamed.             *
*                                                                     *
*  The TASKLIB option of the ATTACH macro is used                     *
*  to specify the steplib for the ATTACHed command (ie. the compiler) *
*                                                                     *
*  The Compiler and Library should be specified on the DD             *
*  referred to in the DCB for the TASKLIB if one or both              *
*  are not already defined in LPA.  The compiler and library do not   *
*  need to be part of the steplib concatenation.                      *
*                                                                     *
***********************************************************************
ATTACH   CSECT
         STM   14,12,12(13)
         BALR  3,0
         USING *,3
         LR    12,15
         ST    13,SAVE+4
         LA    15,SAVE
         ST    15,8(,13)
         LR    13,15
*
*   Invoke the compiler using ATTACH macro
*
         OPEN  (COMPILER)
         ATTACH EP=CCNDRVR,PARAM=(OPTIONS,DDNAMES),VL=1,DCB=COMPILER, X
               ECB=ECBADDR,TASKLIB=COMPILER
         ST    1,TCBADDR
         WAIT  1,ECB=ECBADDR
         DETACH TCBADDR
         CLOSE (COMPILER)
         L     13,4(,13)
         LM    14,12,12(13)
         SR    15,15
         BR    14
*
*   Constant and save area
*
 SAVE DC 18F'0'
ECBADDR DC F'0'
TCBADDR DC F'0'
OPTIONS DC H'12',C'SOURCE EVENT'
```

Figure 59. Using the assembler ATTACH Macro (Part 1 of 2)

```
*   For C++, substitute the above line with
*   OPTIONS  DC    H'10',C'CXX SOURCE'

DDNAMES  DC    H'152'
         DC    CL8'NEWIN'
         DC    CL8'NEWLIN'
         DC    CL8'DUMMY'   PLACEHOLDER - NO LONGER USED
         DC    CL8'NEWLIB'
         DC    CL8'NEWRLIB'
         DC    CL8'NEWPRINT'
         DC    CL8'NEWCPRT'
         DC    CL8'NEWPUNCH'
         DC    CL8'NEWUT1'
         DC    CL8'NEWUT4'
         DC    CL8'NEWUT5'
         DC    CL8'NEWUT6'
         DC    CL8'NEWUT7'
         DC    CL8'NEWUT8'
         DC    CL8'NEWUT9'
         DC    CL8'NEWUT10'
         DC    CL8'NEWUT14'
         DC    CL8'NEWUT15'
         DC    CL8'NEWEVENT'
COMPILER DCB   DDNAME=MYCOMP,DSORG=PO,MACRF=R
         END
```

Figure 59. Using the assembler ATTACH Macro (Part 2 of 2)

Example of JCL for the assembler ATTACH macro (CCNUAAQ)

```
//*------------------------------------------------------------------------
//* Standard DDname Renaming  (ASM ATTACH from driver program)
//*    compiles                   MYID.MYPROG.SOURCE(HELLO)
//*    and places the object in   MYID.MYPROG.OBJECT(HELLO)
//*
//*    User header files come from MYID.MYHDR.FILES
//*    using MYCOMP as the compile time steplib.
//*
//*    Compilation is controlled by the assembler module named
//*    CCNUAAP which is stored in MYID.ATTACHDD.LOAD
//*
//*    This example uses the Language Environment Library
//*------------------------------------------------------------------------
//G001001B  EXEC PGM=CCNUAAP
//STEPLIB   DD DSN=MYID.ATTACHDD.LOAD,DISP=SHR
//MYCOMP    DD DSN=CBC.SCCNCMP,DISP=SHR
//          DD DSN=CEE.SCEERUN,DISP=SHR
//          DD DSN=CEE.SCEERUN2,DISP=SHR
//NEWIN     DD DSN=MYID.MYPROG.SOURCE(HELLO),DISP=SHR
//NEWLIB    DD DSN=CEE.SCEEH.H,DISP=SHR
//NEWLIN    DD DSN=MYID.MYPROG.OBJECT(HELLO),DISP=SHR
//NEWPRINT  DD SYSOUT=*
//NEWCPRT   DD SYSOUT=*,DCB=(RECFM=VBA,LRECL=137,BLKSIZE=882)
//NEWPUNCH  DD DSN=...
//SYSTERM   DD DUMMY
//NEWUT1    DD DSN=...
//NEWUT4    DD DSN=...
//NEWUT5    DD DSN=...
//NEWUT6    DD DSN=...
//NEWUT7    DD DSN=...
//NEWUT8    DD DSN=...
//NEWUT9    DD DSN=...
//NEWUT10   DD SYSOUT=*
//NEWUT14   DD DSN=...
//NEWUT15   DD DSN=...
//NEWEVENT  DD DSN=...
//NEWRLIB   DD DSN=MYID.MYHDR.FILES,DISP=SHR
//*-----------------------------------------------------------------------
```

Figure 60. JCL for the assembler ATTACH macro

Note that the sharing of resources between attached programs is not supported.

Example of using the assembler LINK macro (CCNUAAR)

```
***********************************************************************
*                                                                     *
*  This assembler routine demonstrates DD Name renaming               *
*  (Dynamic compilation) using the assembler LINK macro.              *
*                                                                     *
*  In this specific scenario a subset of all the DDNAMES are          *
*  renamed.  The DDNAMES you do not want to rename are set to zero.   *
*                                                                     *
*  The Compiler and the Library should be in the LPA, or should       *
*  be specified on the STEPLIB DD in your JCL                         *
*                                                                     *
***********************************************************************
*
LINK     CSECT
         STM   14,12,12(13)
         BALR  3,0
         USING *,3
         LR    12,15
         ST    13,SAVE+4
         LA    15,SAVE
         ST    15,8(,13)
         LR    13,15
*
*   Invoke the compiler using LINK macro
*
         LINK  EP=CCNDRVR,PARAM=(OPTIONS,DDNAMES),VL=1
         L     13,4(,13)
         LM    14,12,12(13)
         SR    15,15
         BR    14
```

Figure 61. Using the assembler LINK macro (Part 1 of 2)

```
*
*   Constant and save area
*
*   This macro will compile for the Language Environment Library
*
SAVE     DC    18F'0'
OPTIONS  DC    H'8',C'SO EVENT'
*   For C++, substitute the above line with
*   OPTIONS  DC    H'6',C'CXX SO'
DDNAMES  DC    H'152'
         DC    CL8'NEWIN'
         DC    XL8'0000000000000000'
         DC    XL8'0000000000000000'
         DC    XL8'0000000000000000'
         DC    CL8'NEWRLIB'
         DC    XL8'0000000000000000'
         DC    CL8'NEWCPRT'
         DC    XL8'0000000000000000'
         DC    2XL8'0000000000000000'
         DC    2XL8'0000000000000000'
         DC    2XL8'0000000000000000'
         DC    XL8'0000000000000000'
         DC    XL8'0000000000000000'
         DC    XL8'0000000000000000'
         DC    XL8'0000000000000000'
         DC    XL8'0000000000000000'
         END
```

Figure 61. Using the assembler LINK macro (Part 2 of 2)

Example of JCL for the assembler LINK macro (CCNUAAS)

```
//*-------------------------------------------------------------------
//* Standard DDname Renaming  using the assembler LINK macro
//*    compiles                    MYID.MYPROG.SOURCE(HELLO)
//*    and places the object in    MYID.MYPROG.OBJECT(HELLO)
//*
//*    User header files come from MYID.MYHDR.FILES
//*
//*    Compilation is controlled by the assembler module named
//*    CCNUAAR that is stored in MYID.LINKDD.LOAD
//*
//*    This JCL uses the Language Environment Library.
//*
//*-------------------------------------------------------------------
//G001003A  EXEC PGM=CCNUAAR
//STEPLIB   DD DSN=CBC.SCCNCMP,DISP=SHR
//          DD DSN=CEE.SCEERUN,DISP=SHR
//          DD DSN=CEE.SCEERUN2,DISP=SHR
//          DD DSN=MYID.LINKDD.LOAD,DISP=SHR
//NEWIN     DD DSN=MYID.MYPROG.SOURCE(HELLO),DISP=SHR
//SYSLIB    DD DSN=CEE.SCEEH.H,DISP=SHR
//SYSLIN    DD DSN=MYID.MYPROG.OBJECT(HELLO),DISP=SHR
//SYSPRINT  DD SYSOUT=*
//NEWCPRT   DD SYSOUT=*,DCB=(RECFM=VBA,LRECL=137,BLKSIZE=882)
//SYSPUNCH  DD SYSOUT=*
//SYSTERM   DD DUMMY
//SYSUT1    DD DSN=...
//SYSUT5    DD DSN=...
//SYSUT6    DD DSN=...
//SYSUT7    DD DSN=...
//SYSUT8    DD DSN=...
//SYSUT9    DD DSN=...
//SYSUT10   DD SYSOUT=*
//SYSUT14   DD DSN=...
//SYSEVENT  DD DSN=...
//NEWRLIB   DD DSN=MYID.MYHDR.FILES,DISP=SHR
//*------------------------------------------------------------------
```

Figure 62. JCL for the assembler LINK macro

Example of using the assembler CALL macro (CCNUAAT)

```
***********************************************************************
*                                                                     *
*  This assembler routine demonstrates DD Name renaming               *
*  (Dynamic compilation) using the Assembler CALL macro.              *
*                                                                     *
*  In this specific scenario, a subset of all the DDNAMES are         *
*  renamed.  This renaming is accomplished by shortening              *
*  the list of ddnames.                                               *
*                                                                     *
*  The Compiler and the Library should be either in the LPA or        *
*  specified on the STEPLIB DD in your JCL                            *
*                                                                     *
***********************************************************************
*
LINK     CSECT
         STM   14,12,12(13)
         USING LINK,15
         LA    3,MODE31
         O     3,=X'80000000'
         DC    X'0B03'
MODE31   DS    0H
         USING *,3
         LR    12,15
         ST    13,SAVE+4
         LA    15,SAVE
         ST    15,8(,13)
         LR    13,15
*
*   Invoke the compiler using CALL macro
*
         LOAD  EP=CCNDRVR
         LR    15,0
         CALL  (15),(OPTIONS,DDNAMES),VL
         L     13,4(,13)
         LM    14,12,12(13)
         SR    15,15
         BR    14
```

Figure 63. Using the assembler CALL macro (Part 1 of 2)

```
*
*   Constant and save area
*
SAVE     DC    18F'0'
OPTIONS  DC    H'2',C'SO'
*   For C++, substitute the above line with
*   OPTIONS  DC    H'6',C'CXX SO'
DDNAMES  DC    H'96'
         DC    CL8'NEWIN'
         DC    CL8'NEWLIN'
         DC    CL8'DUMMY'       PLACEHOLDER - NO LONGER USED
         DC    CL8'NEWLIB'
         DC    CL8'NEWRLIB'
         DC    CL8'NEWPRINT'
         DC    CL8'NEWCPRT'
         DC    CL8'NEWPUNCH'
         DC    CL8'NEWUT1'
         DC    CL8'NEWUT4'
         DC    CL8'NEWUT5'
         DC    CL8'NEWUT6'
         END
```

Figure 63. Using the assembler CALL macro (Part 2 of 2)

Example of JCL for assembler CALL macro (CCNUAAU)

```
//*------------------------------------------------------------------------
//* Standard DDname Renaming  using the assembler CALL macro
//*    compiles                     MYID.MYPROG.SOURCE(HELLO)
//*    and places the object in     MYID.MYPROG.OBJECT(HELLO)
//*
//*    User Header files come from  MYID.MYHDR.FILES
//*
//*    Compilation is controlled by the assembler module named
//*    CCNUAAT which is stored in MYID.CALLDD.LOAD
//*
//*    This JCL uses the Language Environment Library.
//*
//*------------------------------------------------------------------------
//G001004C  EXEC PGM=CCNUAAT
//STEPLIB   DD DSN=CBC.SCCNCMP,DISP=SHR
//          DD DSN=CEE.SCEERUN,DISP=SHR
//          DD DSN=CEE.SCEERUN2,DISP=SHR
//          DD DSN=MYID.CALLDD.LOAD,DISP=SHR
//NEWIN     DD DSN=MYID.MYPROG.SOURCE(HELLO),DISP=SHR
//NEWLIB    DD DSN=CEE.SCEEH.H,DISP=SHR
//NEWLIN    DD DSN=MYID.MYPROG.OBJECT(HELLO),DISP=SHR
//NEWPRINT  DD SYSOUT=*
//NEWCPRT   DD SYSOUT=*,DCB=(RECFM=VBA,LRECL=137,BLKSIZE=882)
//NEWPUNCH  DD DSN=...
//SYSTERM   DD DUMMY
//NEWUT1    DD DSN=...
//NEWUT4    DD DSN=...
//NEWUT5    DD DSN=...
//NEWUT6    DD DSN=...
//SYSUT7    DD DSN=...
//SYSUT8    DD DSN=...
//SYSUT9    DD DSN=...
//SYSUT10   DD SYSOUT=*
//SYSUT14   DD DSN=...
//NEWRLIB   DD DSN=MYID.MYHDR.FILES,DISP=SHR
//*-----------------------------------------------------------------------
```

Figure 64. JCL for the assembler CALL macro

Appendix E. Layout of the Events file

This information specifies the layout of the SYSEVENT file. SYSEVENT is an events file that contains error information and source file statistics. The SYSEVENT file is not the same as the binder Input Event Log. Use the EVENTS compiler option to produce the SYSEVENT file. For more information on the EVENTS compiler option, see "EVENTS | NOEVENTS" on page 108.

In the following example, the source file `simple.c` is compiled with the `EVENTS(USERID.LIST(EGEVENT))`compiler option. The file `err.h` is a header file that is included in `simple.c`. Figure 67 is the event file that is generated when `simple.c` is compiled.

```
 1   #include "./err.h"
 2   main() {
 3      add some error messages;
 4      return(0);
 5      here and there;
 6   }
```

Figure 65. simple.c

```
1   add some;
2   errors in the header file;
```

Figure 66. err.h

```
-------  start simple.events  ------
  FILEID 0 1 0 10 ./simple.c
  FILEID 0 2 1 9 ././err.h
  ERROR 0 2 0 0 1 1 0 0 CCN1AAA E 12 48 Definition of function add require
  FILEEND 0 2 2
  ERROR 0 2 0 0 1 5 0 0 CCN1BBB E 12 35 Syntax error: possible missing '{'
  ERROR 0 1 0 0 3 3 0 0 CCN1CCC E 12 26 Undeclared identifier add.
  ERROR 0 1 0 0 5 8 0 0 CCN1DDD E 12 42 Syntax error: possible missing ';'
  ERROR 0 1 0 0 5 3 0 0 CCN1EEE E 12 27 Undeclared identifier here.
  FILEEND 0 1 6
-------  end   simple.events  ------
```

Figure 67. Sample SYSEVENT file

There are three different record types generated in the event file:
- FILEID
- FILEEND
- ERROR

Description of the FILEID field

The following is an example of the FILEID field from the sample SYSEVENT file that is shown in Figure 67. Table 54 describes the FILEID identifiers.

```
FILEID 0 1 0 10 ./simple.c
       A B C D  E
```

Table 54. Explanation of the FILEID field layout

Column	Identifier	Description
A	Revision	Revision number of the event record.

Table 54. Explanation of the FILEID field layout (continued)

Column	Identifier	Description
B	File number	Increments starting with 1 for the primary file.
C	Line number	The line number of the `#include` directive. For the primary source file, this value is 0.
D	File name length	Length of file or data set.
E	File name	String containing file/data set name.

Description of the FILEEND field

The following is an example of the FILEEND field from the sample SYSEVENT file that is shown in Figure 67 on page 669. Table 55 describes the FILEEND identifiers.

```
FILEEND 0 1 6
        A B C
```

Table 55. Explanation of the FILEEND field layout

Column	Identifier	Description
A	Revision	Revision number of the event record
B	File number	File number that has been processed to end of file
C	Expansion	Total number of lines in the file

Description of the ERROR field

The following is an example of the ERROR field from the sample SYSEVENT file that is shown in Figure 67 on page 669. Table 56 describes the ERROR identifiers.

```
ERROR 0 1 0 0 3 3 0 0 CBCMMMM E 12 26 Undeclared identifier add.
      A B C D E F G H I       J K  L  M
```

Table 56. Explanation of the ERROR field layout

Column	Identifier	Description
A	Revision	Revision number of the event record.
B	File number	Increments starting with 1 for the primary file.
C	Reserved	Do not build a dependency on this identifier. It is reserved for future use.
D	Reserved	Do not build a dependency on this identifier. It is reserved for future use.

Table 56. Explanation of the ERROR field layout (continued)

Column	Identifier	Description
E	Starting line number	The source line number for which the message was issued. A value of 0 indicates the message was not associated with a line number.
F	Starting column number	The column number or position within the source line for which the message was issued. A value of 0 indicates the message was not associated with a line number.
G	Reserved	Do not build a dependency on this identifier. It is reserved for future use.
H	Reserved	Do not build a dependency on this identifier. It is reserved for future use.
I	Message identifier	String Containing the message identifier.

J	Message severity character	`I=Informational`
		`W=Warning`
		`E=Error`
		`S=Severe`
		`U=Unrecoverable`

K	Message severity number	Return code associated with the message.
L	Message length	Length of message text.
M	Message text	String containing message text.

Appendix F. Customizing default options for z/OS XL C/C++ compiler

System programmers can customize the default options for the z/OS XL C/C++ compiler by modifying USERMODs and jobs. The USERMOD can then be applied by submitting the jobs, which are also provided. See the following table for information on the USERMODs and jobs that can be modified.

Table 57. z/OS XL C/C++ compiler USERMODs and jobs

Element name	USERMODS	Resides in	Jobs to submit
z/OS XL C compiler	CCNEOPT	CBC.SCCNJCL(CCNJOPT)	CBC.SCCNJCL(CCNJMOD)
z/OS XL C++ compiler	CCNEOPX	CBC.SCCNJCL(CCNJOPX)	CBC.SCCNJCL(CCNJMOX)

For the z/OS XL C compiler, the USERMOD CCNEOPT modifies the CCNEO00C assembler source file, which defines the z/OS XL C compiler options. It also defines the system include file SEARCH path, which includes Language Environment header files.

For the z/OS XL C++ compiler, the USERMOD CCNEOPX modifies the CCNEO00X assembler source file, which defines the z/OS XL C++ compiler options. It also defines the system include file SEARCH path, which includes Language Environment header files.

If you plan to apply these USERMODs and have used a different prefix than the one supplied by IBM for the :Language Environment and the Run-Time Library Extensions elements, please change the value of CEE and CBC to your chosen prefix on the SEARCH statements. Do not accept your USERMOD into the distribution library, as you might want to remove your USERMOD if you find it does not suit the needs of the programmers at your site.

Appendix G. Accessibility

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Vol I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

z/OS information

z/OS information is accessible using screen readers with the BookServer/Library Server versions of z/OS books in the Internet library at:

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

© Copyright IBM Corp. 1998, 2009

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright IBM Corp. 1998, 2009

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:

Lab Director
IBM Canada Ltd. Laboratory
B3/KB7/8200/MKM
8200 Warden Avenue
Markham, Ontario L6G 1C7
Canada

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.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may 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.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

This publication documents *intended* Programming Interfaces that allow the customer to write z/OS XL C/C++ programs.

Policy for unsupported hardware

Various z/OS elements, such as JES, DFSMS, HCD 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 support' 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.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

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

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

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

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

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

Standards

The following standards are supported in combination with the Language Environment element:

- The C language is consistent with *Programming languages - C (ISO/IEC 9899:1999)*. For more information on ISO, visit their web site at: www.iso.org
- The C++ language is consistent with *Programming languages - C++ (ISO/IEC 14882:2003(E))* and *Programming languages - C++ (ISO/IEC 14882:1998)*.

The following standards are supported in combination with the Language Environment and z/OS UNIX System Services elements:

- A subset of *IEEE Std. 1003.1-2001 (Single UNIX Specification, Version 3).* For more information on IEEE, visit their web site at: www.ieee.org.
- *IEEE Std 1003.1—1990, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE P1003.1a Draft 6 July 1991, Draft Revision to Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*, copyright 1992 by the Institute of Electrical and Electronic Engineers, Inc.
- *IEEE Std 1003.2—1992, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shells and Utilities*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- The core features of *IEEE Std P1003.4a/D6—1992, IEEE Draft Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1:*

System Application Program Interface (API)—Amendment 2: Threads Extension [C language], copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

- The core features of *IEEE 754-1985 (R1990) IEEE Standard for Binary Floating-Point Arithmetic (ANSI)*, copyright 1985 by the Institute of Electrical and Electronic Engineers, Inc.
- *X/Open CAE Specification, System Interfaces and Headers, Issue 4 Version 2*, copyright 1994 by The Open Group
- *X/Open CAE Specification, Networking Services, Issue 4*, copyright 1994 by The Open Group
- *X/Open Specification Programming Languages, Issue 3, Common Usage C*, copyright 1988, 1989, and 1992 by The Open Group
- United States Government's *Federal Information Processing Standard (FIPS) publication for the programming language C, FIPS-160*, issued by National Institute of Standards and Technology, 1991

Glossary

This glossary defines technical terms and abbreviations that are used in z/OS XL C/C++ documentation. If you do not find the term you are looking for, refer to the index of the appropriate z/OS XL C/C++ manual or view *IBM Glossary of Computing Terms*, located at: www.ibm.com/ibm/terminology. This glossary includes terms and definitions from:

A

abstract class. (1) A class with at least one pure virtual function that is used as a base class for other classes. (2) In object-oriented programming, a class that represents a concept; classes derived from it represent implementations of the concept. An object cannot be constructed from an abstract class; that is, it cannot be instantiated.See also base class.

abstract code unit (ACU). A measurement used by the z/OS XL C/C++ compiler for judging the size of a function. The number of ACUs that comprise a function is proportional to its size and complexity.

abstract data type. A mathematical model that includes a structure for storing data and operations that can be performed on that data. Common abstract data types include sets, trees, and heaps.

access mode. (1) A form of access permitted for a file. (2) The manner in which files are referred to by a computer.

access specifier. A specifier that defines whether a class member is accessible in an expression or declaration. The three access specifiers are public, private, and protected.

ACU. See abstract code unit.

address space (ASID). The range of addresses available to a computer program or process. Address space can refer to physical storage, virtual storage, or both.

addressing mode (AMODE). The attribute of a program module that identifies the addressing range in which the program entry point can receive control.

aggregate. (1) A structured collection of data objects that form a data type. (2) An array, structure, or union. (3) In C++, an array or a class with no user-declared constructors, no private or protected non-static data members, no base classes, and no virtual functions.

alert. (1) A message or other indication that signals an event or an impending event. (2) To cause the user's terminal to give some audible or visual indication that an error or some other event has occurred.

alert character. A character that in the output stream causes a terminal to alert its user by way of a visual or audible notification. The alert character is the character designated by a '\a' in the C and C++ languages. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the alert function.

alias. (1) An alternative name for an integrated catalog facility (ICF) user catalog, a file that is not a Virtual Storage Access Method (VSAM) file, or a member of a partitioned data set (PDS) or a partitioned data set extended (PDSE). (2) An alternative name used instead of a primary name.

aliasing. A compilation process that attempts to determine what aliases exist, so that optimization does not result in incorrect program results.

alignment. The storing of data in relation to certain machine-dependent boundaries.

alternate code point. A syntactic code point that permits a substitute code point to be used. For example, the left brace ({) can be represented by X'B0' and also by X'C0'.

American National Standards Institute (ANSI). A private, nonprofit organization whose membership includes private companies, U.S. government agencies, and professional, technical, trade, labor, and consumer organizations. ANSI coordinates the development of voluntary consensus standards in the U.S.

American Standard Code for Information Interchange (ASCII). A standard code used for information exchange among data processing systems, data communication systems, and associated equipment. ASCII uses a coded character set consisting of 7-bit coded characters.

AMODE. See addressing mode.

angle bracket. Either the left angle bracket (<) or the right angle bracket (>). In the portable character set, these characters are referred to by the names <less-than-sign> and <greater-than-sign>.

anonymous union. An unnamed object whose type is an unnamed union.

ANSI. See American National Standards Institute.

AP. See application program.

API. See application programming interface.

application. One or more computer programs or software components that provide a function in direct support of a specific business process or processes.

application generator. An application development tool that creates applications, application components (panels, data, databases, logic, interfaces to system services), or complete application systems from design specifications.

application program (AP). A complete, self-contained program, such as a text editor or a Web browser, that performs a specific task for the user, in contrast to system software, such as the operating system kernel, server processes, and program libraries.

application programming interface (API). An interface that allows an application program that is written in a high-level language to use specific data or functions of the operating system or another program.

archive library. A facility for grouping application-program object files. The archive library file, when created for application-program object files, has a special symbol table for members that are object files.

argument. A value passed to or returned from a function or procedure at run time.

argument declaration.

arithmetic object. An integral object or objects having the float, double, or long double type.

array. In programming languages, an aggregate that consists of data objects, with identical attributes, each of which can be uniquely referenced by subscripting.

array element. One of the data items in an array.

ASCII. See American Standard Code for Information Interchange.

ASID. See address space.

assembler. A computer program that converts assembly language instructions into object code.

Assembler H. An IBM licensed program that translates symbolic assembler language into binary machine language.

assembler language. A symbolic programming language that represents machine instructions of a specific architecture.

assembler user exit. A routine to tailor the characteristics of an enclave prior to its establishment.

assignment expression. An expression that assigns the value of the right operand expression to the left operand variable and has as its value the value of the right operand.

auto storage class specifier. A specifier that enables the programmer to define a variable with automatic storage; its scope is restricted to the current block.

automatic call library. A group of modules that are used as secondary input to the binder to resolve external symbols left undefined after all the primary input has been processed. The automatic call library can contain: object modules, with or without binder control statements; load modules; and runtime routines.

automatic library call. The process by which the binder resolves external references by including additional members from the automatic call library.

automatic storage. Storage that is allocated on entry to a routine or block and is freed when control is returned.

B

background process. A process that does not require operator intervention but can be run by the computer while the workstation is used to do other work.

background processing. A mode of program execution in which the shell does not wait for program completion before prompting the user for another command.

backslash. The character \. The backslash enables a user to escape the special meaning of a character. That is, typing a backslash before a character tells the system to ignore any special meaning the character might have.

base class. A class from which other classes or beans are derived. A base class may itself be derived from another base class.

binary expression. An expression containing two operands and one operator.

binary stream. A sequence of characters that corresponds on a one-to-one basis with the characters in the file. No character translation is performed on binary streams.

binder. (1) The z/OS program that processes the output of language translators and compilers into an executable program (a load module or program object). The binder replaces the linkage editor and batch loader. (2) See linkage editor.

bit field. A member of a structure or union that contains 1 or more named bits.

bitwise operator. An operator that manipulates the value of an object at the bit level.

blank character. (1) A graphic representation of the space character. (2) One of the characters that belong to the blank character class as defined via the

LC_CTYPE category in the current locale. In the POSIX locale, a blank character is either a tab or a space character.

block. (1) A string of data elements recorded, processed, or transmitted as a unit. The elements can be characters, words, or physical records. (2) In programming languages, a compound statement that coincides with the scope of at least one of the declarations contained within it. A block may also specify storage allocation or segment programs for other purposes.

block statement. In the C or C++ languages, a group of data definitions, declarations, and statements appearing between a left brace and a right brace that are processed as a unit. The block statement is considered to be a single, C-language statement.

boundary alignment. The position in main storage of a fixed-length field, such as halfword or doubleword, which is aligned on an integral boundary for that unit of information. For example, a word boundary alignment stores the object in a storage address evenly divisible by four.

brace. Either of the characters left brace ({) and right brace (}). When an object is enclosed in braces, the left brace immediately precedes the object and the right brace immediately follows it.

bracket. Either of the characters left bracket ([) and right bracket (]).

break statement. A C or C++ control statement that contains the keyword break and a semicolon (;). It is used to end an iterative or a switch statement by exiting from it at any point other than the logical end. Control is passed to the first statement after the iteration or switch statement.

built-in. In programming languages, pertaining to a language object that is defined in the programming language specification.

built-in function. A function for which the compiler automatically generates inline code at compile time instead of making a runtime call to the function.

byte-oriented stream. A byte-oriented stream refers to a stream which only single byte input/output is allowed.

C

C language. A language used to develop application programs in compact, efficient code that can be run on different types of computers with minimal change.

C library. A system library that contains common C language subroutines for file access, string operations, character operations, memory allocation, and other functions.

C++ language. An object-oriented high-level language that evolved from the C language. C++ takes advantage of the benefits of object-oriented technology such as code modularity, portability, and reuse.

C++ library. A system library that contains common C++ language subroutines for file access, memory allocation, and other functions.

call chain. A trace of all active routines and subroutines, such as the names of routines and the locations of save areas, that can be constructed from information included in a system dump.

callable service. A program service provided through a programming interface.

caller. A function that calls another function.

cancelability point. A specific point within the current thread that is enabled to solicit cancel requests.

carriage return character. A character that in the output stream indicates that printing should start at the beginning of the same physical line in which the carriage-return character occurred.

case clause. In a C or C++ switch statement, a CASE label followed by any number of statements.

case label. The word case followed by a constant expression and a colon. When the selector is evaluated to the value of the constant expression, the statements following the case label are processed.

cast expression. An expression that converts or reinterprets its operand.

cast operator. An operator that is used for explicit type conversions.

cataloged procedure. A set of job control language (JCL) statements that has been placed in a library and that is retrievable by name.

catch block. A block associated with a try block that receives control when an exception matching its argument is thrown.

CCS. See coded character set.

character. (1) A sequence of one or more bytes representing a single graphic symbol or control code. (2) In a computer system, a member of a set of elements that is used for the representation, organization, or control of data.

character class. A named set of characters sharing an attribute associated with the name of the class. The classes and the characters that they contain are dependent on the value of the LC_CTYPE category in the current locale.

character constant. The actual character value (a symbol, quantity, or constant) in a source program that is itself data, instead of reference to a field that contains the data.

character set. A defined set of characters with no coded representation assumed that can be recognized by a configured hardware or software system. A character set may be defined by alphabet, language, script, or any combination of these items.

character special file. A special interface file that provides access to an input or output device, which uses character I/0 instead of block I/0.

character string. A contiguous sequence of characters terminated by and including the first null byte.

child. A node that is subordinate to another node in a tree structure. Only the root node is not a child.

child enclave. The nested enclave created as a result of certain commands being issued from a parent enclave.

child process. A process that is created by a parent process and that shares the resources of the parent process to carry out a request.

CICS. An IBM licensed program that provides online transaction-processing services and management for business applications.

class. In C++, a user-defined data type. A class data type can contain both data representations (data members) and functions (member functions).

class key. One of the C++ keywords: class, struct, and union.

class library. In object-oriented programming, a collection of prewritten classes or coded templates, any of which can be specified and used by a programmer when developing an application.

class name. A unique identifier of a class type that becomes a reserved word within its scope.

class scope. The scope of C++ class members.

class template. A blueprint describing how a set of related C++ classes can be constructed.

class template declaration. A class template declaration introduces the name of a class template and specifies its template parameter list. A class template declaration may optionally include a class template definition.

class template definition. A definition that describes various characteristics of the class types that are its specializations. These characteristics include the names and types of data members of specializations, the signatures and definitions of member functions, accessibility of members, and base classes.See also base class.

client program. A program that uses a C++ class.

CLIST. See command list.

COBOL. See Common Business Oriented Language.

code element set. The result of applying rules that map a numeric code value to each element of a character set. An element of a character set may be related to more than one numeric code value but the reverse is not true. However, for state-dependent encodings the relationship between numeric code values to elements of a character set may be further controlled by state information. The character set may contain fewer elements than the total number of possible numeric code values; that is, some code values may be unassigned. X/Open.

code generator. The part of the compiler that physically generates the object code.

code page. A particular assignment of code points to graphic characters. Within a given code page, a code point can have only one specific meaning. A code page also identifies how undefined code points are handled.See also code point.

code point. (1) A unique bit pattern that represents a character in a code page.See also code page. (2) An identifier in an alert description that represents a short unit of text. The code point is replaced with the text by an alert display program.

coded character set (CCS). A set of unambiguous rules that establishes a character set and the one-to-one relationships between the characters of the set and their coded representations.

collating element. The smallest entity used to determine the logical ordering of strings. A collating element consists of either a single character, or two or more characters collating as a single entity. The value of the LC_COLLATE category in the current locale determines the current set of collating elements.

collating sequence. The relative ordering of collating elements as determined by the setting of the LC_COLLATE category in the current locale. The character order, as defined for the LC_COLLATE category in the current locale, defines the relative order of all collating elements, such that each element occupies a unique position in the order.

collation. The logical ordering of characters and strings according to defined rules.

collection. An abstract class without any ordering, element properties, or key properties.

Collection Class Library. A complete set of abstract data structure such as trees, stacks, queues, and linked lists.

column position. A unit of horizontal measure related to characters in a line. It is assumed that each character in a character set has an intrinsic column width independent of any output device. Each printable character in the portable character set has a column width of one. The standard utilities, when used as described in this document set, assume that all characters have integral column widths. The column width of a character is not necessarily related to the internal representation of the character (numbers of bits or bytes). The column position of a character in a line is defined as one plus the sum of the column widths of the preceding characters in the line. Column positions are numbered starting from 1. X/Open.

comma expression. An expression that contains two operands separated by a comma. Although the compiler evaluates both operands, the value of the right operand is the value of the expression. If the left operand produces a value, the compiler discards this value.

command. A request to perform an operation or run a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

command list (CLIST). A language for performing TSO tasks.

COMMAREA. See communication area.

Common Business Oriented Language (COBOL). A high-level programming language that is used primarily for commercial data processing.

communication area (COMMAREA). A CICS area that is used to pass data between tasks that communicate with a given terminal. The area can also be used to pass data between programs within a task.

compilation unit. A portion of a computer program sufficiently complete to be compiled correctly.

compiler option. A keyword that can be specified to control certain aspects of compilation. Compiler options can control the nature of the load module generated by the compiler, the types of printed output to be produced, the efficient use of the compiler, and the destination of error messages.

complete class name. The complete qualification of a nested C++ class name including all enclosing class names and namespaces.

Complex Mathematics Library. A C++ class library that provides the facilities to manipulate complex numbers and perform standard mathematical operations on them.

concrete class. (1) A class defining objects that can be created. (2) A class that is not abstract.

condition. (1) An exception that has been enabled, or recognized, by the Language Environment and thus is eligible to activate user and language condition handlers. Conditions can be detected by the hardware/operating system and result in an interrupt. They can also be detected by language-specific generated code or language library code. (2) An expression that can be evaluated as true, false, or unknown. It can be expressed in natural language text, in mathematically formal notation, or in a machine-readable language.

condition handler. A user-written routine or language-specific routine (such as a PL/ION-unit or C signal() function call) invoked by the Language Environment condition manager to respond to conditions.

condition manager. The condition manager is the part of the common execution environment that manages conditions by invoking various user-written and language-specific condition handlers.

condition token. In Language Environment, a data type consisting of 96 bits (12 bytes). The condition token contains structured fields that indicate various aspects of a condition including the severity, the associated message number, and information that is specific to a given instance of the condition.

conditional expression. A compound expression that contains a condition (the first expression), an expression to be evaluated if the condition has a nonzero value (the second expression), and an expression to be evaluated if the condition has the value zero (the third expression).

constant. A language element that specifies an unchanging value. Constants are classified as string constants or numeric constants.

constant expression. An expression that has a value that can be determined during compilation and that does not change during the running of the program.

constant propagation. An optimization technique where constants used in an expression are combined and new ones are generated. Mode conversions are done to allow some intrinsic functions to be evaluated at compile time.

constructed reentrancy. The attribute of applications that contain external data and require additional processing to make them reentrant.

constructor. A special C++ class member function that has the same name as the class and is used to create an object of that class.

control character. A character whose occurrence in a particular context initiates, modifies, or stops a control function.

control section (CSECT). The part of a program specified by the programmer to be a relocatable unit, all elements of which are to be loaded into adjoining main storage locations.

control statement. In programming languages, a statement that is used to interrupt the continuous sequential processing of programming statements. Conditional statements such as IF, PAUSE, and STOP are examples of control statements.

controlling process. A session leader that has control of a terminal.

controlling terminal. The active workstation from which the process group for that process was started. Each session may have at most one controlling terminal associated with it, and a controlling terminal is associated with exactly one session.

conversion. (1) In programming languages, the transformation between values that represent the same data item but belong to different data types. Information may be lost because of conversion since accuracy of data representation varies among different data types. (2) The process of changing from one form of representation to another. Changing a code point that is assigned to a character in one code page to its corresponding code point in another code page is an example of conversion. (3) The process of changing from one method of data processing to another or from one data processing system to another. Changing a code point that is assigned to a character in one code page to its corresponding code point in another code page is an example of conversion.

conversion function. A C++ member function that specifies a conversion from its class type to another type.

Coordinated Universal Time (UTC). The international standard of time that is kept by atomic clocks around the world.

copy constructor. A C++ constructor used to make a copy of a class object from another class object of the same class type.

cross-compiler. A compiler that produces executable files that run on a platform other than the one on which the compiler is installed.

CSECT. See control section.

current working directory. See working directory.

cursor. A reference to an element at a specific position in a data structure.

D

data abstraction. A data type with a private representation and a public set of operations (functions or operators) which restrict access to that data type to that set of operations. The C++ language uses the concept of classes to implement data abstraction.

data definition (DD). A program statement that describes the features of, specifies relationships of, or establishes the context of data. A data definition reserves storage and can provide an initial value.

data definition name (ddname). The name of a data definition (DD) statement that corresponds to a data control block that contains the same name.

data definition statement (DD statement). A job control statement that is used to define a data set for use by a batch job step, started task or job, or an online user.

data member. The smallest possible piece of complete data. Elements are composed of data members.

data object. An element of data structure such as a file, an array, or an operand that is needed for the execution of an application.

data set. The major unit of data storage and retrieval, consisting of a collection of data in one of several prescribed arrangements and described by control information to which the system has access.

data stream. The commands, control codes, data, or structured fields that are transmitted between an application program and a device such as printer or nonprogrammable display station.

data structure. In Open Source Initiative (OSI), the syntactic structure of symbolic expressions and their storage allocation characteristics.

data type. A category that identifies the mathematical qualities and internal representation of data and functions.

Data Window Services (DWS). Services provided as part of the Callable Services Library that allow manipulation of data objects such as VSAM linear data sets and temporary data objects known as TEMPSPACE.

DBCS. See double-byte character set.

DCT. See destination control table.

DD. See data definition.

DD statement. See data definition statement.

ddname. See data definition name.

dead code. Code that is never referenced, or that is always branched over.

dead store. A store into a memory location that will later be overwritten by another store without any intervening loads. In this case, the earlier store can be deleted.

decimal constant. A numerical data type used in standard arithmetic operations. Decimal constants can contain any digits 0 through 9.

decimal overflow. A condition that occurs when one or more nonzero digits are lost because the destination field in a decimal operation is too short to contain the results.

declaration. (1) A statement that establishes the names and characteristics of data objects and functions used in a program. (2) In the C and C++ languages, a description that makes an external object or function available to a function or a block statement.

default argument. An argument that is declared with default values in a C++ function prototype or declaration. If a call to the function omits this argument, a default value is used. An arguments with a default value must be the trailing argument in a function prototype argument list.

default clause. In the C or C++ languages, within a switch statement, the keyword default followed by a colon, and one or more statements. When the conditions of the specified case labels in the switch statement do not hold, the default clause is chosen.

default constructor. A C++ constructor that takes no arguments, or if it takes any arguments, all its arguments have default values.

default initialization. The initial value assigned to a data object by the compiler if no initial value is specified by the programmer. In C language, external and static variables receive a default initialization of zero, while the default initialization for auto and register variables is undefined.

definition. A declaration that reserves storage and can provide an initial value for a data object or define a function.

degree. The number of children of a node.

demangling. The conversion of mangled C++ names back to their original source code names to make program debugging easier.

dereference. In the C and C++ languages, to apply the unary operator * to a pointer to access the object the pointer points to.

derivation. The process of deriving a C++ class from an existing class, called a base class.

derived class. See base class.

descriptor. A PL/I control block that holds information such as string lengths, array subscript bounds, and area sizes, and is passed from one PL/I routine to another during run time.

destination control table (DCT). A table describing each of the transient data destinations used in CICS. This table contains an entry for each extrapartition, intrapartition, and indirect destination.

destructor. A special member function of a class with the same name as the class with a ~ (tilde) preceding the name. You cannot specify arguments or a return type for this function. A destructor ″cleans up″ after an object by doing such things as freeing any storage that was dynamically allocated when the object was created.

device. A piece of equipment. Devices can be workstations, printers, disk drives, tape units, or remote systems.

difference. Given two sets A and B, the set of all elements contained in A but not in B (A-B).

digraph. A combination of two keystrokes used to represent unavailable characters in a C or C++ source program. Digraphs are read as tokens during the preprocessor phase.

directive. A control statement that directs the operation of a feature and is recognized by a preprocessor or other tool.

directory. (1) In a hierarchical file system, a grouping of related files and directories. A directory can contain zero or more entries, which refer to other directories and files. (2) The part of a partitioned data set that describes the members in the data set.

display. To direct the output to the user's terminal. If the output is not directed to the terminal, the results are undefined.

DLL. See dynamic link library.

do statement. For the C and C++ compilers, a looping statement that contains the keyword do, followed by a statement (the action), the keyword while, and an expression in parentheses (the condition).

dot. A symbol (.) that indicates the current directory in a relative path name.

double quote. See quotation mark.

double-byte character set (DBCS). A set of characters in which each character is represented by two bytes. These character sets are commonly used by national languages, such as Japanese and Chinese, that have more symbols than can be represented by a single byte.

double-precision. Pertaining to the use of two computer words to represent a number in accordance with the required precision.

doubleword. A contiguous sequence of bits or characters that comprises two computer words and is capable of being addressed as a unit.See also halfword.

DSA. See dynamic storage area.

DWS. See Data Window Services.

dynamic. Pertaining to an operation that occurs at the time it is needed rather than at a predetermined or fixed time.

dynamic access. A process where records can be accessed sequentially or randomly, depending on the form of the input/output request.

dynamic allocation. Assignment of system resources to a program when the program is executed rather than when it is loaded into main storage.

dynamic binding. The act of resolving references to external variables and functions at run time. In C++, dynamic binding is supported by using virtual functions.

dynamic link library (DLL). A file containing executable code and data bound to a program at load time or run time, rather than during linking. The code and data in a DLL can be shared by several applications simultaneously.

dynamic storage. An area of storage that is explicitly allocated by a program or procedure while it is running.See also automatic storage.

dynamic storage area (DSA). A type of storage allocation in which storage is assigned to a program or application at run time.

E

EBCDIC. See Extended Binary Coded Decimal Interchange Code.

effective group ID. An attribute of a process that is used in determining various permissions, including file access permissions.This value is subject to change during the process lifetime.

elaborated type specifier. Typically used in C++ in an incomplete class declaration or to qualify types that are otherwise hidden.

element. The smallest unit in a table, array, list, set, or other structure. Examples of an element are a value in a list of values and a data field in an array.

element equality. A relation that determines if two elements are equal.

element occurrence. A single instance of an element in a collection. In a unique collection, element occurrence is synonymous with element value.

element value. All the instances of an element with a particular value in a collection. In a non-unique collection, an element value may have more than one occurrence. In a unique collection, element value is synonymous with element occurrence.

else clause. The part of an if statement that contains the keyword 'else' followed by a statement. The else clause provides an action that is started when the if condition evaluates to a value of 0 (false).

empty line. A line consisting of only a newline character. X/Open.

empty string. A character array whose first element is a null character.

encapsulation. In object-oriented programming, the technique that is used to hide the inherent details of an object, function, or class from client programs.

entry point. The address or label of the first instruction processed or entered in a program, routine, or subroutine.There might be a number of different entry points, each corresponding to a different function or purpose.

enum constant. See enumeration constant.

enumeration constant (enum constant). In the C or C++ language, an identifier, with an associated integer value, defined in an enumerator. An enumeration constant may be used anywhere an integer constant is allowed.

enumeration data type. In the Fortran, C, and C++ language, a data type that represents a set of values that a user defines.

enumeration tag. The identifier that names an enumeration data type.

enumeration type. A data type that defines a set of enumeration constants. In the C++ language, an enumeration type is a distinct data type that is not an integral type.

enumerator. An enumeration constant and its associated value.

equivalence class. A grouping of characters or character strings that are considered equal for purposes of collation. For example, many languages place an uppercase character in the same equivalence class as its lowercase form, but some languages distinguish between accented and unaccented character forms for the purpose of collation.

escape sequence. A sequence of characters that contains the \ symbol followed by one of the characters: a ,b ,f ,n ,r ,t ,v ,' , ,x ,\ , or by one or more octal or hexadecimal digits.

exception. A condition or event that cannot be handled by a normal process.

exception handler. (1) A set of routines that responds to an abnormal condition. An exception handler is able to interrupt and to resume the normal running of processes. (2) In C++, the catch block that catches exceptions when they are thrown from a function enclosed in a try block.

executable file. A file that contains programs or commands that perform operations on actions to be taken.

executable program. A program in a form suitable for execution by a computer. The program can be an application or a shell script.

Extended Binary Coded Decimal Interchange Code (EBCDIC). A coded character set of 256 8-bit characters developed for the representation of textual data.

extended-precision. Pertains to the use of more than two computer words to represent a floating point number in accordance with the required precision. For example, in z/OS, four computer words are used for an extended-precision number.

extension. An element or function not included in the standard language.

Extra Performance Linkage (XPLINK). A type of call linkage that can improve performance in an environment of frequent calls between small functions.

extrapartition destination. In CICS, a type of transient data queue. Extrapartition destinations can be accessed either within the CICS environment or outside of CICS; they can be defined as either input or output.

F

FIFO special file. A type of file with the property that data written to such a file is read on a first-in-first-out (FIFO) basis.

file descriptor. A positive integer or a data structure that uniquely identifies an open file for the purpose of file access.

file mode. An object containing the file permission bits and other characteristics of a file.

file permission bit. Information about a file that is used, along with other information, to determine whether a process has read, write, or execute permission to a file. The use of file permission bits is described in file access permissions.

file scope. A property of a file name that is declared outside all blocks, classes, and function declarations and that can be used after the point of declaration in a source file.

filter. A command that reads standard input data, modifies the data, and sends it to standard output. A pipeline usually has several filters.

flat collection. A collection that has no hierarchical structure.

float constant. (1) A constant representing a non-integral number. (2) A number containing a decimal point, an exponent, or both a decimal point and an exponent. The exponent contains an ″e″ or ″E,″ an optional sign (+ or -), and one or more digits (0 through 9).

footprint. The amount of computer storage that is occupied by a computer program. For example, if a program occupies a large amount of storage, it has a large footprint.

for statement. A looping statement that contains the word for followed by a list of expressions enclosed in parentheses (the condition) and a statement (the action). Each expression in the parenthesized list is separated by a semicolon, which cannot be omitted.

foreground process. A process that must be completed before another command is issued.See also background process.

foreground process group. A group whose member processes have privileges that are denied to background processes when the controlling terminal is being accessed. Each controlling terminal can have only one foreground process group.

form-feed character. A character in the output stream that indicates that printing should start on the next page of an output device. The form-feed character is designated by '\f' in the C and C++ language. If the form-feed character is not the first character of an output line, the result is unspecified. X/Open.

forward declaration. A declaration of a class or function made earlier in a compilation unit, so that the declared class or function can be used before it has been defined.

free store. Dynamically allocated memory. New and delete are used to allocate and deallocate free store.

freestanding application. (1) A z/OS C/C++ application that does not use the services of the dynamic z/OS C/C++ run-time library or of the Language Environment. Under z/OS C support, this ability is a feature of the System Programming C support. (2) An application that is created to run without the run-time environment or library with which it was developed.

friend class. A class in which all member functions are granted access to the private and protected members of another class. It is named in the declaration of another class and uses the keyword friend as a prefix to the class.

function. A named group of statements that can be called and evaluated and can return a value to the calling statement.

function call. An expression that transfers the path of execution from the current function to a specified function (the called function). A function call contains the name of the function to which control is transferred and a parenthesized list of values

function declarator. The part of a function definition that names the function, provides additional information about the return value of the function, and lists the function parameters.

function definition. The complete description of a function. A function definition contains an optional storage class specifier, an optional type specifier, a function declarator, optional parameter declarations, and a block statement (the function body).

function prototype. A function declaration that provides type information for each parameter. It is the first line of the function (header) followed by a semicolon (;). The declaration is required by the compiler at the time that the function is declared, so that the compiler can check the type.

function scope. Labels that are declared in a function have function scope and can be used anywhere in that function after their declaration.

function template. A detailed plan that describes the construction of a set of related individual C++ functions.

G

GCC. See GNU Compiler Collection.

GDDM. See Graphical Data Display Manager.

generalization. The derivation of the definition of a class, function, or static data member from a template. An instantiation of a template function is a generalization.

Generalized Object File Format (GOFF). This object module format extends the capabilities of object modules so that they can contain more information. It is required for XPLINK.

generic class. See class template.

global. Pertaining to information available to more than one program or subroutine.

global variable. A symbol defined in one program module that is used in other program modules that are independently compiled.

GMT. See Greenwich mean time.

GNU Compiler Collection (GCC). An open source collection of compilers supporting C, C++, Objective-C, Ada, Java, and Fortran.

GOFF. See Generalized Object File Format.

graphic character. A visual representation of a character, other than a control character, that is normally produced by writing, printing, or displaying.

Graphical Data Display Manager (GDDM). An IBM computer-graphics system that defines and displays text and graphics for output on a display or printer.

Greenwich mean time (GMT). The mean solar time at the meridian of Greenwich, England.

H

halfword. A contiguous sequence of bits or characters that constitutes half a computer word and can be addressed as a unit.See also doubleword.

hash function. A function that determines which category, or bucket, to put an element in. A hash function is needed when implementing a hash table.

hash table. (1) A data structure that divides all elements into (preferably) equal-sized categories, or buckets, to allow quick access to the elements. The hash function determines which bucket an element belongs in. (2) A table of information that is accessed by way of a shortened search key (the hash value). The use of a hash table minimizes average search time.

header file. See include file.

heap storage. An area of storage used for allocation of storage that has a lifetime that is not related to the execution of the current routine. The heap consists of the initial heap segment and zero or more increments.

hexadecimal constant. A constant, usually starting with special characters, that contains only hexadecimal digits.

High Level Assembler. An IBM licensed program that translates symbolic assembler language into binary machine language.

hiperspace memory file. A type of file that is stored in a single buffer in an address space, with the rest of the data being kept in a hiperspace. In contrast, for regular files, all the file data is stored in a single address space.

hook. A location in a compiled program where the compiler has inserted an instruction that allows programmers to interrupt the program (by setting breakpoints) for debugging purposes.

hybrid code. Program statements that have not been internationalized with respect to code page, especially where data constants contain variant characters. Such statements can be found in applications written in older implementations of MVS, which required syntax statements to be written using code page IBM-1047 exclusively. Such applications cannot be converted from one code page to another using iconv().

I

I/O Stream Library. A class library that provides the facilities to deal with many varieties of input and output.

ID. See identifier.

identifier (ID). One or more characters used to identify or name a data element and possibly to indicate certain properties of that data element.

if statement. A conditional statement that specifies a condition to be tested and the action to be taken if the condition is satisfied.

ILC. (1) See interlanguage call. (2) See interlanguage communication.

implementation-defined. Pertaining to behavior that is defined by the compiler rather than by a language standard. Programs that rely on implementation-defined behavior may behave differently when compiled with different compilers.

IMS. See Information Management System.

include directive. A preprocessor directive that causes the preprocessor to replace the statement with the contents of a specified file.

include file. A text file that contains declarations that are used by a group of functions, programs, or users.

incomplete class declaration. A C++ class declaration that does not define any members of a class. Until a class is fully declared or defined, you can use the class name only where the size of the class is not required.

incomplete type. A type that has no value or meaning when it is first declared. There are three incomplete types: void, arrays of unknown size and structures, and unions of unspecified content.

indirect destination. In CICS, a type of transient data destination that points to another destination within the destination control table, rather than directly to a queue.

indirection. (1) A mechanism for connecting objects by storing, in one object, a reference to another object. (2) In the C and C++ languages, the application of the unary operator * to a pointer to access the object to which the pointer points.

indirection class. See reference class.

induction variable. A controlling variable of a loop.

Information Management System (IMS). Any of several system environments available with a database manager and transaction processing that are capable of managing complex databases and terminal networks.

inheritance. An object-oriented programming technique in which existing classes are used as a basis for creating other classes. Through inheritance, more specific elements incorporate the structure and behavior of more general elements.

initial heap. A heap that is controlled by the HEAP run-time option and designated by a heap_id of 0.

initializer. An expression used to initialize data objects. In the C++ language, there are three types of initializers: an expression followed by an assignment operator initializes fundamental data type objects or class objects that have copy constructors; an expression enclosed in braces ({ }) initializes aggregates; and a parenthesized expression list initializes base classes and members using constructors

inline. To replace a function call with a copy of the function's code during compilation.

inline function. A function whose actual code replaces a function call. A function that is both declared and defined in a class definition is an example of an inline function. Another example is one which you explicitly declared inline by using the keyword inline. Both member and non-member functions can be inlined.

input stream. A sequence of control statements and data submitted to an operating system by an input device.

instance. (1) A specific occurrence of an object that belongs to a class. (2) In object-oriented programming, a region of storage that contains a value or group of values.

instantiate. To create or generate a particular instance or object of a data type. For example, an instance box1 of class box could be instantiated with the declaration: box box1

instruction. A program statement that specifies an operation to be performed by the computer, along with the values or locations of operands. This statement represents the programmer's request to the processor to perform a specific operation. [OSF]

instruction scheduling. An optimization technique that reorders instructions in code to minimize execution time.

integer constant. A decimal, octal, or hexadecimal constant.

integral object. A character object, an object having an enumeration type, an object having variations of the type int, or an object that is a bit field.

Interactive System Productivity Facility (ISPF). An IBM licensed program that serves as a full-screen editor and dialog manager. Used for writing application programs, it provides a means of generating standard screen panels and interactive dialogs between the application programmer and terminal user.

interlanguage call (ILC). A call to a procedure or function made by a program written in one language to a procedure or function coded in a different language.

interlanguage communication (ILC). The ability of routines written in different programming languages to communicate. ILC support enables the application writer to readily build applications from component routines written in a variety of languages.

interoperability. The ability of a computer or program to work with other computers or programs.

interprocess communication (IPC). The process by which programs send messages to each other. Sockets, semaphores, signals, and internal message queues are common methods of interprocess communication.

intrapartition destination. In CICS, a type of transient data queue used subsequently as input data to another task within CICS.

IPC. See interprocess communication.

ISPF. See Interactive System Productivity Facility.

iteration. The repetition of a set of computer instructions until a condition is satisfied.

J

JCL. See job control language.

job control language (JCL). A command language that identifies a job to an operating system and describes the job's requirements.

K

kernel. The part of an operating system that contains programs for such tasks as input/output, management and control of hardware, and the scheduling of user tasks.

keyword. (1) A symbol that identifies a parameter in job control language (JCL). (2) One of the predefined words of a programming language, artificial language, application, or command.

L

label. An identifier within or attached to a set of data elements.

Language Environment. An element of z/OS that provides a common runtime environment and common runtime services for C/C++, COBOL, PL/I, and Fortran applications.

last element. The element visited last in an iteration over a collection. Each collection has its own definition for last element. For example, the last element of a sorted set is the element with the largest value.

leaf. In a tree, an entry or node that has no children.

library. (1) A collection of model elements, including business items, processes, tasks, resources, and organizations. (2) A set of object modules that can be specified in a link command.

link pack area (LPA). The portion of virtual storage below 16MB that contains frequently used modules.

linkage. Refers to the binding between a reference and a definition. A function has internal linkage if the function is defined inline as part of the class, is declared with the inline keyword, or is a non-member function declared with the static keyword. All other functions have external linkage.

linkage editor. A computer program for creating load modules from one or more object modules or load modules by resolving cross-references among the modules and, if necessary, adjusting addresses.

linker. A program that resolves cross-references among separately compiled object modules and then assigns final addresses to create a single executable program.

literal. A symbol or a quantity in a source program that is itself data, rather than a reference to data.

load module. A program in a form suitable for loading into main storage for execution.

loader. A program that copies an executable file into main storage so that the file can be run.

local. (1) In programming languages, pertaining to the relationship between a language object and a block such that the language object has a scope contained in that block. (2) Pertaining to information that is defined and used only in one subdivision of a computer program.

local custom. A convention of a geographical area or territory for such things as date, time, and currency formats. X/Open.

local scope. A name declared in a block that has local scope and can only be used in that block.

locale. A setting that identifies language or geography and determines formatting conventions such as collation, case conversion, character classification, the language of messages, date and time representation, and numeric representation.

loop unrolling. An optimization that increases the step of a loop, and duplicates the expressions within a loop to reflect the increase in the step. This can improve instruction scheduling and memory access time.

LPA. See link pack area.

lvalue. An expression that represents a data object that can be viewed, tested, and changed. An lvalue is usually the left operand in an assignment expression.

M

macro. An instruction that causes the execution of a predefined sequence of instructions.

macro call. See macro.

main function. A function that has the identifier main. Each program must have exactly one function named main. The main function is the first user function that receives control when a program starts to run.

make utility. A utility that maintains all of the parts and dependencies for an application. The make utility uses a makefile to keep the parts of a program synchronized. If one part of an application changes, the make utility updates all other files that depend on the changed part.

makefile. In UNIX, a text file containing a list of an application's parts. The make utility uses makefiles to maintain application parts and dependencies.

mangled name. An external name, such as a function or variable name, which has been encoded during compilation to include type and scope information.

mangling. The encoding, during compilation, of C++ identifiers such as function and variable names to include type and scoping information. The linker uses these mangled names for type-safe linkage.

manipulator. A value that can be inserted into streams or extracted from streams to affect or query the behavior of the stream.

member. A C++ data object or function in a structure, union or class. Members can also be classes, enumerations, bit fields and type names.

member function. A C++ operator or function that is declared as a member of a class. A member function has access to the private and protected data members and member functions of an object of its class. Member functions are also called methods.

method. See member function.

method file. (1) A file that allows users to indicate to the localedef utility where to look for user-provided methods for processing user-designed codepages. (2) For ASCII locales, a file that defines the method functions to be used by C runtime locale-sensitive interfaces. A method file also identifies where the method functions can be found. IBM supplies several method files used to create its standard set of ASCII locales. Other method files can be created to support customized or user-created codepages. Such customized method files replace IBM-supplied charmap method functions with user-written functions.

migrate. To install a new version or release of a program to replace an earlier version or release.

module. A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading.

multibyte character. A mixture of single-byte characters from a single-byte character set and double-byte characters from a double-byte character set.

multibyte control. See escape sequence.

multicharacter collating element. A sequence of two or more characters that collate as an entity. For example, in some coded character sets, an accented character is represented by a non-spacing accent, followed by the letter. Other examples are the Spanish elements ch and ll. X/Open.

multiple inheritance. An object-oriented programming technique implemented in C++ through derivation, in which the derived class inherits members from more than one base class.

multiprocessor. A processor complex that has more than one central processor.

multitasking. A mode of operation in which two or more tasks can be performed at the same time.

mutex. See mutual exclusion.

mutex attribute object. A type of attribute object with which a user can manage mutual exclusion (mutex) characteristics by defining a set of variables to be used during its creation. A mutex attribute object eliminates the need to redefine the same set of characteristics for each mutex object created.

mutex object. An identifier for a mutual exclusion (mutex).

mutual exclusion (mutex). A flag used by a semaphore to protect shared resources. The mutex is locked and unlocked by threads in a program.

N

namespace. A category used to group similar types of identifiers.

natural reentrancy. The attribute of applications that contain no static external data and do not require additional processing to make them reentrant.

nested class. A C++ class defined within the scope of another class.

nested enclave. A new enclave created by an existing enclave. The nested enclave that is created must be a new main routine within the process.

newline character (NL). A control character that causes the print or display position to move down one line.

nickname. See alias.

NL. See newline character.

nonprinting character. See control character.

NUL. See null character.

null character (NUL). A control character with the value of X'00' that represents the absence of a displayed or printed character.

null pointer. The value that is obtained by converting the number 0 into a pointer; for example, (void *) 0 . The C and C++ languages guarantee that this value will not match that of any legitimate pointer, so it is used by many functions that return pointers to indicate an error.

null statement. A statement that consists of a semicolon.

null string. A character or bit string with a length of zero.

null value. A parameter position for which no value is specified.

null wide-character code. A wide-character code with all bits set to zero.

number sign. The character #, which is also referred to as the hash sign.

O

object. (1) A region of storage. An object is created when a variable is defined. An object is destroyed when it goes out of scope. (2) In object-oriented design or programming, a concrete realization (instance) of a class that consists of data and the operations associated with that data. An object contains the instance data that is defined by the class, but the class owns the operations that are associated with the data.

object module. A set of instructions in machine language that is produced by a compiler or assembler from a subroutine or source module and can be input to the linking program. The object module consists of object code.

object-oriented programming. A programming approach based on the concepts of data abstraction and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates not on how something is accomplished but instead on what data objects comprise the problem and how they are manipulated.

octal constant. The digit 0 (zero) followed by any digits 0 through 7.

open file. A file that is currently associated with a file descriptor.

operand. An entity on which an operation is performed.

operating system (OS). A collection of system programs that control the overall operation of a computer system.

operator function. An overloaded C++ operator that is either a member of a class or takes at least one argument that is a class type or a reference to a class type.

operator precedence. In programming languages, an order relationship that defines the sequence of the application of operators with an expression.

orientation. The orientation of a stream refers to the type of data which may pass through the stream. A stream without orientation is one on which no stream I/O has been performed.

OS. See operating system.

overflow. The condition that occurs when data cannot fit in the designated field.

overlay. The technique of repeatedly using the same areas of internal storage during different stages of a program. Unions are used to accomplish this in C and C++.

overloading. The capability of an identifier or method to have different meanings depending on the context. For example, in C++, a user can redefine functions and most standard operators when the functions and operators are used with class types

P

parameter (parm). A value or reference passed to a function, command, or program that serves as input or controls actions. The value is supplied by a user or by another program or process.

parameter declaration. The description of a value that a function receives. A parameter declaration determines the storage class and the data type of the value.

parent enclave. The enclave that issues a call to system services or language constructs to create a nested (or child) enclave.

parent process. A process that is created to carry out a request or set of requests. The parent process, in turn, can create child processes to process requests for the parent.

parent process ID (PPID). An attribute of a new process identifying the parent of the process. The parent process ID of a process is the process ID of its creator for the lifetime of the creator. After the creator's lifetime has ended, the parent process ID is the process ID of an implementation-dependent system process.

parm. See parameter.

partitioned concatenation. The allocation of partitioned data sets (PDSs), partitioned data sets extended (PDSEs), UNIX file directories, or any combination of these such that the basic partitioned access method (BPAM) retrieves them as a single data set.

partitioned data set (PDS). A data set on direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data.See also sequential data set.

partitioned data set extended (PDSE). A data set that contains an indexed directory and members that are similar to the directory and members of partitioned data sets (PDSs).

path name. A name that specifies all directories leading to a file plus the file name itself.

path name resolution. The process of resolving a path name to a particular file in a file hierarchy. There may be multiple path names that resolve to the same file. X/Open.

pattern. A sequence of characters used either with regular expression notation or for path name expansion, as a means of selecting various characters strings or path names, respectively. The syntaxes of the two patterns are similar, but not identical.

PDS. See partitioned data set.

PDSE. See partitioned data set extended.

period. The symbol ″.″. The term dot is used for the same symbol when referring to a Web address or file extension. This character is named <period> in the portable character set.

permission. The ability to access a protected object, such as a file or directory. The number and meaning of permissions for an object are defined by the access control list.

persistent environment. An environment that once created by the user may be used repeatedly without incurring the overhead of initialization and termination for each call. The environment remains available until explicitly terminated by the user.

PGID. See process group ID.

PID. See process ID.

platform. The combination of an operating system and hardware that makes up the operating environment in which a program runs.

pointer. A data element or variable that holds the address of a data object or a function.

pointer class. A class that implements pointers.

pointer to member. An identifier that is used to access the address of nonstatic members of a C++ class.

polymorphism. An object-oriented programming characteristic that allows a method to perform differently, depending on the class that implements it. Polymorphism allows a subclass to override an inherited method without affecting the parent class's method. Polymorphism also enables a client to access two or more implementations of an object from a single interface.

portability. (1) The ability of a program to run on more than one type of computer system without modification. (2) The ability of a programming language to compile successfully on different operating systems without requiring changes to the source code.

portable character set. A set of characters, specified in POSIX 1003.2, section 4, that must be supported by conforming implementations.

portable file name character set. The set of characters from which portable file names must be constructed to be portable across implementations conforming to the ISO POSIX-1 standard and to ISO/IEC 9945.

positional parameter. A parameter that must appear in a specified location, relative to other parameters.

PPID. See parent process ID.

pragma. A standardized form of comment which has meaning to a compiler. A pragma usually conveys non-essential information, often intended to help the compiler to optimize the program.

precedence. The priority system for grouping different types of operators with their operands.

predefined macro. In C/C++, an identifier predefined by the compiler, which will be expanded by the preprocessor during compilation.

preinitialization. A process by which an environment or library is initialized once and can then be used repeatedly to avoid the inefficiency of initializing the environment or library each time it is needed.

prelinker. A utility that preprocesses an object for certain programs.

preprocessor. A routine that performs initial processing and translation of source code or data prior to compiling the source code or processing the data in another program such as an emulator.

preprocessor directive. In the C and C++ languages, a statement that begins with the symbol # and is interpreted by the preprocessor during compilation.

preprocessor statement. In the C and C++ languages, a statement that begins with the symbol # and contains instructions that the preprocessor can interpret.

primary expression. (1) Any of the following types of expressions: a) identifiers, b) parenthesized expressions, c) function calls, d) array element specifications, e) structure member specifications, or f) union member specifications. (2) Literals, names, and names qualified by the :: (scope resolution) operator.

private. Pertaining to a member of a class that is accessible only to member functions and friends of that class.

process. (1) An address space and single thread of control that executes within that address space, and its required system resources. A process is created by another process issuing the fork() function. The process that issues the fork() function is known as the parent process, and the new process created by the fork() function is known as the child process. (2) An instance of a program running on a system and the resources that it uses.

process group. A collection of processes in a system that is identified by a process group ID.

process group ID (PGID). The unique identifier representing a process group during its lifetime. A process group ID is a positive integer that is not reused by the system until the process group lifetime ends.

process group lifetime. A period of time that begins when a process group is created and ends when the last remaining process in the group leaves the group because either it is the end of the last process' lifetime or the last remaining process is calling the setsid() or setpgid() functions. X/Open. ISO.1.

process ID (PID). The unique identifier that represents a process. A process ID is a positive integer and is not reused until the process lifetime ends.

process lifetime. The period of time that begins when a process is created and ends when the process ID is returned to the system. X/Open. ISO.1 . After a process is created with a fork() function, it is considered active. Its thread of control and address space exist until it terminates. It then enters an inactive state where certain resources may be returned to the system, although some resources, such as the process ID, are still in use. When another process executes a wait() or waitpid() function for an inactive process, the remaining resources are returned to the system. The last resource to be returned to the system is the process ID. At this time, the lifetime of the process ends.

profile-directed feedback. A two-stage compilation process that first compiles and runs a program to analyze its behavior and then recompiles the program to optimize its execution. The results of the analysis stage are saved in a profile data file that is input to the second, optimization stage.

profiling. A performance analysis process that is based on statistics for the resources that are used by a program or application.

program object. All or part of a computer program in a form suitable for loading into virtual storage for execution. Program objects are stored in partitioned data set extended (PDSE) program libraries and have fewer restrictions than load modules. Program objects are produced by the binder.

program unit. See compilation unit.

protected. Pertaining to a class member that is accessible to the class itself, subclasses, and all classes in the same package.

prototype. A function declaration or definition that includes both the return type of the function and the types of its parameters.

public. In object-oriented programming, pertaining to a class member that is accessible to all classes.

pure virtual function. A virtual function is declared pure by replacing the function definition with '=0;'.

Q

QMF. See Query Management Facility.

qualified class name. Any class name or class name qualified with one or more :: (scope) operators.

qualified name. (1) A data set name consisting of a string of names separated by periods; for example, TREE.FRUIT.APPLE is a qualified name. (2) In C++, a name that is used to qualify a nonclass type name, such as a member, by its class name.

qualified type name. A name used to reduce complex class name syntax by using typedefs to represent qualified class names.

Query Management Facility (QMF). An IBM query and report writing facility that supports a variety of tasks such as data entry, query building, administration, and report analysis.

queue. A data structure for processing work in which the first element added to the queue is the first element processed. This order is referred to as first-in first-out (FIFO).

quotation mark. The characters ″ and ’.

R

radix character. The character that separates the integer part of a number from the fractional part. X/Open .

random access. A process where individual records can be referred to in a nonsequential manner.

real group ID. The attribute of a process that, at the time of process creation, identifies the group of the user who created the process. This value is subject to change during the process lifetime.

real user ID. The attribute of a process that, at the time a process is created, identifies the user who created the process.

reason code. A value used to indicate the specific reason for an event or condition.

reassociation. An optimization technique that rearranges the sequence of calculations in a subscript expression producing more candidates for common expression elimination.

redirection. In a shell, a method of associating files with the input or output of commands.

reentrant. The attribute of a program or routine that allows the same copy of the program or routine to be used concurrently by two or more tasks.

reference class. A class that links a concrete class to an abstract class. Reference classes make polymorphism possible with the collection classes.

refresh. To ensure that the information on the user’s terminal screen is up-to-date.

register variable. A variable defined with the register storage class specifier. Register variables have automatic storage.

regular expression. (1) A mechanism for selecting specific strings from a set of character strings. (2) A set of characters, meta characters, and operators that define a string or group of strings in a search pattern. (3) A string containing wildcard characters and operations that define a set of one or more possible strings.

regular file. A file that is a randomly accessible sequence of bytes, with no further structure imposed by the system. [POSIX.1]

relation. An unordered flat collection class that uses keys, allows for duplicate elements, and has element equality.

relative path name. A string of characters that is used to refer to an object and that starts at some point in the directory hierarchy other than the root. The starting point is frequently a user’s current directory.

reserved word. A word that is defined by a programming language and that cannot be used as an identifier or changed by the user.

residency mode (RMODE). In z/OS, a program attribute that refers to where a module is prepared to run. RMODE can be 24 or ANY. ANY refers to the fact that the module can be loaded either above or below the 16M line. RMODE 24 means the module expects to be loaded below the 16M line.

reverse solidus. See backslash.

RMODE. See residency mode.

runtime environment. A set of resources that are used to run a program or process.

runtime library. A compiled collection of functions whose members can be referred to by an application program at run time.

S

SBCS. See single-byte character set.

scalar. An arithmetic object, an enumerated object, or a pointer to an object.

scope. A part of a source program in which an object is defined and recognized.

scope operator. In C++, an operator that defines the scope for the argument on the right: if the left argument is blank, the scope is global; if the left argument is a class name or namespace name, then the scope is within that class or namespace respectively.

SDK. See software development kit.

semaphore. An object used by multi-threaded applications for signaling purposes and for controlling access to serially reusable resources. Processes can be locked to a resource with semaphores if the processes follow certain programming conventions.

sequence. A sequentially ordered flat collection.

sequential access. The process of referring to records one after another in the order in which they appear on the file.

sequential concatenation. The allocation of sequential data sets, partitioned data set (PDS) members, partitioned data set extended (PDSE) members, UNIX files, or any combination of these such that the system retrieves them as a single, sequential, data set.

sequential data set. A data set whose records are organized on the basis of their successive physical positions, such as on magnetic tape.

session. A collection of process groups established for job control purposes.

shell. A software interface between users and an operating system. Shells generally fall into one of two categories: a command line shell, which provides a command line interface to the operating system; and a graphical shell, which provides a graphical user interface (GUI).

signal. (1) A condition that might or might not be reported during program execution. For example, a signal can represent erroneous arithmetic operations, such as division by zero. (2) A mechanism by which a process can be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term signal is also used to refer to the event itself. (3) In operating system operations, a method of inter-process communication that simulates software interrupts.

signal handler. A subroutine or function that is called when a signal occurs.

single precision. The use of one computer word to represent a number, in accordance with the required precision.

single quote. See quotation mark.

single-byte character set (SBCS). A coded character set in which each character is represented by a 1-byte code. A 1-byte code point allows representation of up to 256 characters.

slash. The character /, also known as forward slash. This character is named <slash> in the portable character set.

socket. In the Network Computing System (NCS), a port on a specific host; a communications end point that is accessible through a protocol family's addressing mechanism. A socket is identified by a socket address.

software development kit (SDK). A set of tools, APIs, and documentation to assist with the development of software in a specific computer language or for a particular operating environment.

sorted map. A sorted flat collection with key and element equality.

sorted relation. A sorted flat collection that uses keys, has element equality, and allows duplicate elements.

sorted set. A sorted flat collection with element equality.

source module. See source program.

source program. A set of instructions that are written in a programming language and must be translated into machine language before the program can be run.

space character. In the portable character set, the <space> character.

spanned record. A logical record stored in more than one block on a storage medium.

specialization. A user-supplied definition which replaces a corresponding template instantiation.

spill area. A storage area that is used to save the contents of registers.

SQL. See Structured Query Language.

square bracket. See bracket.

stack frame. See dynamic storage area.

standard error (STDERR). The output stream to which error messages or diagnostic messages are sent.

standard input (STDIN). An input stream from which data is retrieved. Standard input is normally associated with the keyboard, but if redirection or piping is used, the standard input can be a file or the output from a command.

standard output (STDOUT). The output stream to which data is directed. Standard output is normally associated with the console, but if redirection or piping is used, the standard output can be a file or the input to a command.

stanza. A grouping of options in a configuration file to control various aspects of compilation by default.

statement. In programming languages, a language construct that represents a step in a sequence of actions or a set of declarations.

static binding. The act of resolving references to external variables and functions before run time.

STDERR. See standard error.

STDIN. See standard input.

STDOUT. See standard output.

storage class specifier. A storage class keyword that determines storage duration, scope, and linkage.

stream. A file access object that allows access to an ordered sequence of characters, as described by the ISO C standard. Such objects can be created by the fdopen() or fopen() functions, and are associated with a file descriptor. A stream provides the additional services of user-selectable buffering and formatted input and output.

string. A contiguous sequence of bytes terminated by and including the first null byte.

string constant. Zero or more characters enclosed in double quotation marks.

string literal. Zero or more characters enclosed in double quotation marks.

striped data set. An extended-format data set that occupies multiple volumes. A striped data set is a software implementation of sequential data striping.

struct. See structure.

struct tag. See structure tag.

structure. A class data type that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types.

structure tag. The identifier that names a structure data type.

Structured Query Language (SQL). A standardized language for defining and manipulating data in a relational database.

stub routine. Within a runtime library, a routine that contains the minimum lines of code needed to locate a given routine.

subprogram. In the IPA Link version of the Inline Report listing section, an equivalent term for 'function'.

subscript. One or more expressions, each enclosed in brackets, that follow an array name. A subscript refers to an element in an array.

subtree. A tree structure created by arbitrarily denoting a node to be the root node in a tree. A subtree is always part of a whole tree.

superset. Given two sets A and B, A is a superset of B if and only if all elements of B are also elements of A. That is, A is a superset of B if B is a subset of A.

support. In system development, to provide the necessary resources for the correct operation of a functional unit.

switch expression. The controlling expression of a switch statement.

switch statement. A C or C++ language statement that causes control to be transferred to one of several statements depending on the value of an expression.

system default. A default value defined in the system profile.

system process. An implementation-dependent object, other than a process executing an application, that has a process ID. X/Open.

T

tab character. A character that indicates that printing or displaying should start at the next horizontal position on the current line. The tab is designated by '\t' in the C language and is named in the portable character set.

task library. A class library that provides the facilities to write programs that consist of tasks.

template. A family of C++ classes or functions with variable types.

template class. A C++ class instance generated by a class template.

template function. A C++ function generated by a function template.

template instantiation. The act of creating a new definition of a function, class, or member of a class from a template declaration and one or more template arguments.

text file. A file that contains only printable characters.

thread. A stream of computer instructions that is in control of a process. In some operating systems, a thread is the smallest unit of operation in a process. Several threads can run concurrently, performing different jobs.

throw. In programming languages, to pass an error or exception to a handling routine.

tilde. One of the accent marks in Latin script (~).

token. The basic syntactic unit of a computing language. A token consists of one or more characters, excluding the blank character and excluding characters within a string constant or delimited identifier.

toolchain. A collection of programs or tools used to develop a product.

traceback. A section of a dump that provides information about the stack frame, the program unit address, the entry point of the routine, the statement number, and status of the routines on the call-chain at the time the traceback was produced.

trigraph. A sequence of three graphic characters that represent another graphic character. For example, in the C programming language, the trigraph ??= is used to denote the # character.

truncate. To shorten a field, value, statement, or string.

try block. A C++ block in which a known exception is passed to an exception handler.

type definition. A definition of a name for a data type.

type specifier. In programming languages, a keyword used to indicate the data type of an object or function being declared.

U

ultimate consumer. The target for data in an input and output operation. An ultimate consumer can be a file, a device, or an array of bytes in memory.

ultimate producer. The source for data in an input and output operation. An ultimate producer can be a file, a device, or an array of bytes in memory.

unary expression. An expression that contains one operand.

undefined behavior. Referring to a program or function that might produce erroneous results without warning because of its use of an indeterminate value, or because of erroneous program constructs or erroneous data.

union tag. An identifier that names a union data type.

UNIX System Services. An element of z/OS that creates a UNIX environment that conforms to XPG4 UNIX 1995 specifications and that provides two open-system interfaces on the z/OS operating system: an application programming interface (API) and an interactive shell interface.

UTC. See Coordinated Universal Time.

V

volatile attribute. An attribute of a data object that indicates the object is changeable. Any expression referring to a volatile object is evaluated immediately (for example, assignments).

W

while statement. A looping statement that executes one or more instructions repeatedly during the time that a condition is true.

white space. A sequence of one or more characters, such as the blank character, the newline character, or the tab character, that belong to the space character class.

wide character. A character whose range of values can represent distinct codes for all members of the largest extended character set specified among the supporting locales.

wide-character code. An integral value that corresponds to a single graphic symbol or control code.

wide-character string. A contiguous sequence of wide characters terminated by and including the first instance of a null wide character.

wide-oriented stream. A wide-oriented stream refers to a stream which only wide character input/output is allowed.

word. A fundamental unit of storage that refers to the amount of data that can be processed at a time. Word size is a characteristic of the computer architecture.See also doubleword, halfword.

working directory. The active directory. When a file name is specified without a directory, the current directory is searched.

writable static area (WSA). An area of memory in a program that is modifiable during the running of a program. Typically, this area contains global variables and function and variable descriptors for dynamic link libraries (DLLs).

write. (1) To make a permanent or transient record of data in a storage device or on a data medium. (2) To output characters to a file, such as standard output or standard error. Unless otherwise stated, standard output is the default output destination for all uses of the term write. [POSIX.2]

WSA. See writable static area.

X

XPLINK. See Extra Performance Linkage.

Bibliography

This bibliography lists the publications for IBM products that are related to z/OS XL C/C++. It includes publications covering the application programming task. The bibliography is not a comprehensive list of the publications for these products, however, it should be adequate for most z/OS XL C/C++ users. Refer to *z/OS Information Roadmap*, SA22-7500, for a complete list of publications belonging to the z/OS product.

Related publications not listed in this section can be found on the *IBM Online Library Omnibus Edition MVS Collection*, SK2T-0710, the *z/OS Collection*, SK3T-4269, or on a tape available with z/OS.

z/OS

- *z/OS Introduction and Release Guide*, GA22-7502
- *z/OS Planning for Installation*, GA22-7504
- *z/OS Summary of Message and Interface Changes*, SA22-7505
- *z/OS Information Roadmap*, SA22-7500
- *z/OS Licensed Program Specifications*, GA22-7503
- *z/OS Migration*, GA22-7499
- *z/OS Program Directory*, GI10-0670

z/OS XL C/C++

- *z/OS XL C/C++ Programming Guide*, SC09-4765
- *z/OS XL C/C++ User's Guide*, SC09-4767
- *z/OS XL C/C++ Language Reference*, SC09-4815
- *z/OS XL C/C++ Messages*, GC09-4819
- *z/OS XL C/C++ Run-Time Library Reference*, SA22-7821
- *z/OS C Curses*, SA22-7820
- *z/OS XL C/C++ Compiler and Run-Time Migration Guide for the Application Programmer*, GC09-4913
- *Standard C++ Library Reference*, SC09-4949

z/OS Metal C Runtime Library

- *z/OS Metal C Programming Guide and Reference*, SA23-2225

z/OS Run-Time Library Extensions

- *z/OS Common Debug Architecture User's Guide*, SC09-7653
- *z/OS Common Debug Architecture Library Reference*, SC09-7654
- *DWARF/ELF Extensions Library Reference*, SC09-7655

Debug Tool

- Debug Tool documentation, which is available at: www.ibm.com/software/awdtools/debugtool/library/

© Copyright IBM Corp. 1998, 2009

z/OS Language Environment

- *z/OS Language Environment Concepts Guide*, SA22-7567
- *z/OS Language Environment Customization*, SA22-7564
- *z/OS Language Environment Debugging Guide*, GA22-7560
- *z/OS Language Environment Programming Guide*, SA22-7561
- *z/OS Language Environment Programming Reference*, SA22-7562
- *z/OS Language Environment Run-Time Application Migration Guide*, GA22-7565
- *z/OS Language Environment Writing Interlanguage Communication Applications*, SA22-7563
- *z/OS Language Environment Run-Time Messages*, SA22-7566

Assembler

- *HLASM Language Reference*, SC26-4940
- *HLASM Programmer's Guide*, SC26-4941

COBOL

- COBOL documentation, which is available at: www.ibm.com/software/awdtools/cobol/zos/library/

PL/I

- PL/I documentation, which is available at: www.ibm.com/software/awdtools/pli/plizos/library/

VS FORTRAN

- VS FORTRAN documentation, which is available at: www.ibm.com/software/awdtools/fortran/vsfortran/library.html

CICS Transaction Server for z/OS

- CICS Transaction Server 2.3 documentation, which is available at: www.ibm.com/software/htp/cics/tserver/v23/library/
- CICS Transaction Server 3.1 documentation, which is available at: www.ibm.com/software/htp/cics/tserver/v31/library/
- CICS Transaction Server 3.2 documentation, which is available at: www.ibm.com/software/htp/cics/tserver/v32/library/

DB2

- DB2 for z/OS documentation, which is available at: www.ibm.com/software/data/db2/zos/library.html

IMS/ESA®

- IMS documentation, which is available at: www.ibm.com/software/data/ims/library.html

MVS

- *z/OS MVS Program Management: User's Guide and Reference*, SA22-7643
- *z/OS MVS Program Management: Advanced Facilities*, SA22-7644

QMF

- QMF documentation, which is available at: www.ibm.com/software/data/qmf/library.html

DFSMS

- *z/OS DFSMS Introduction*, SC26-7397
- *z/OS DFSMS Managing Catalogs*, SC26-7409
- *z/OS DFSMS Using Data Sets*, SC26-7410
- *z/OS DFSMS Macro Instructions for Data Sets*, SC26-7408
- *z/OS DFSMS Access Method Services for Catalogs*, SC26-7394

INDEX

© Copyright IBM Corp. 1998, 2009

Program Number: 5694-A01

Printed in the United States of America

SC09-4767-08