z/OS

C/C++ User's Guide

SC09-4767-01

z/OS

C/C++ User's Guide

SC09-4767-01

Note!
Before using this information and the product it supports, be sure to read the information in “Notices” on page 619.

Second Edition (October 2001)

This edition applies to Version 1 Release 2 Modification 0 of z/OS C/C++ (5694-A01) and to all subsequent releases and modifications until otherwise indicated in new editions. This edition replaces SC09-4767-00. 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.

Order publications through your IBM® representative or the IBM branch office serving your location. Publications are not stocked at the address below. You can also browse the books on the World Wide Web by clicking on ″The Library″ link on the z/OS home page. The web address for this page is
`http://www.ibm.com/servers/eserver/zseries/zos/bkserv`

IBM welcomes your comments. You can send your comments by mail to the following address:

IBM Canada Ltd. Laboratory
Information Development
B3/KB7/8200/MKM
8200 Warden Avenue
Markham, Ontario Canada
L67 1C7

If you send comments, 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.

Contents

Part 1. Introduction

This part discusses the z/OS C/C++ information library, and presents introductory concepts on the z/OS C/C++ product. Specifically, it discusses the following:

Chapter 1. About This Book

This edition of *z/OS C/C++ User's Guide* is intended for users of the IBM z/OS C/C++ compiler with the z/OS Language Environment® product. 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.

To use this, or any other z/OS C/C++ book, you must have a working knowledge of the C and C++ programming languages. You should also know the operating system, and the related products as appropriate. This includes the z/OS Language Environment product and z/OS UNIX® System Services (z/OS UNIX).

This book 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 book; 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 books.

z/OS C/C++ and Related Publications

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

Table 1. z/OS C/C++ Publications

Book Title and Number	Key Sections/Chapters in the Book
z/OS C/C++ Programming Guide, SC09-4765	Guidance information for: • C/C++ input and output • Debugging z/OS 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 applications • Reentrancy • Using the decimal data type in C and C++ • Handling exceptions, error conditions, and signals • Optimizing code • Optimizing your C/C++ code with Interprocedural Analysis • Network communications under z/OS UNIX • Interprocess communications using z/OS UNIX • 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 C multitasking facility • Using other IBM products with z/OS C/C++ (CICS, CSP, DWS, DB2®, GDDM®, IMS™, ISPF, QMF) • Internationalization: locales and character sets, code set conversion utilities, mapping variant characters • POSIX character set • Code point mappings • Locales supplied with z/OS C/C++ • Charmap files supplied with z/OS 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 C/C++
z/OS C/C++ User's Guide, SC09-4767	Guidance information for: • z/OS C/C++ examples • Compiler options • Binder options and control statements • Specifying z/OS Language Environment run-time options • Compiling, IPA Linking, binding, and running z/OS C/C++ programs • Utilities (Object Library, DLL Rename, CXXFILT, DSECT Conversion, Code Set and Locale, ar and make, BPXBATCH) • Diagnosing problems • Cataloged procedures and REXX EXECs supplied by IBM

Table 1. z/OS C/C++ Publications (continued)

Book Title and Number	Key Sections/Chapters in the Book
C/C++ Language Reference, SC09-4815	Reference information for: • The C and C++ languages • Lexical elements of z/OS C and z/OS 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 C and z/OS C++ compatibility
z/OS C/C++ Messages, GC09-4819	Provides error messages and return codes for the compiler, utilities, and IBM Open Class® Library. For the C/C++ run-time library messages, refer to *z/OS Language Environment Run-Time Messages*, SA22-7566.
z/OS 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 C/C++ Compiler and Run-Time Migration Guide, 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 • C/370™ to current compiler migration • Other migration considerations
IBM Open Class Library User's Guide, SC09-4811	Guidance information for: • Using the Complex Mathematics Class Library: Review of complex numbers, header files, constructing complex objects, mathematical operators for complex, friend functions for complex, handling complex mathematics errors • Using the I/O Stream Class Library: Introduction, getting started, advanced topics, and manipulators • Using the Collection Class Library: Overview, instantiating and using, element and key functions, tailoring a collection implementation, polymorphic use of collections, support for notifications, exception handling, problem solving, compatibility with previous releases, thread safety • Using the Application Support Class Library: Introduction, String classes, Exception and Trace classes, Date and Time classes, controlling threads and protecting data, the IBM Open Class notification framework, Binary Coded (Packed) Decimal classes, text and internationalization framework, testing

Table 1. z/OS C/C++ Publications (continued)

Book Title and Number	Key Sections/Chapters in the Book
IBM Open Class Library Reference, SC09-4812	Reference information for: • Complex Mathematics Class Library • I/O Stream Class Library • Collection Class Library • Application Support Class Library
Debug Tool User's Guide and Reference, SC09-2137	Guidance and reference information for: • Preparing to debug programs • Debugging programs • Using Debug Tool in different environments • Language-specific information • Debug Tool reference
Standard C++ Library Reference, available on the z/OS C/C++ library page on the World Wide Web	The documentation, which is available at `http://www.ibm.com/software/ad/c390/czos/czosdocs.html` covers using 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.
APAR and BOOKS files (Shipped with Program materials)	Partitioned data set CBC.SCCNDOC on the product tape contains the members, APAR and BOOKS, which provide additional information for using the z/OS C/C++ licensed program, including: • Isolating reportable problems • Keywords • Preparing an Authorized Program Analysis Report (APAR) • Problem identification worksheet • Maintenance on z/OS • Late changes to z/OS C/C++ publications

Note: For complete and detailed information on linking and running with z/OS Language Environment and using the z/OS 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 Applications*, SA22-7563.

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

Table 2. Publications by Task

Tasks	Books
Planning, preparing, and migrating to z/OS C/C++	• *z/OS C/C++ Compiler and Run-Time Migration Guide*, GC09-4913 • *z/OS Language Environment Customization*, SA22-7564 • *z/OS Language Environment Run-Time 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

Table 2. Publications by Task (continued)

Tasks	Books
Coding programs	• *z/OS C/C++ Run-Time Library Reference*, SA22-7821 • *C/C++ Language Reference*, SC09-4815 • *z/OS C/C++ Programming Guide*, SC09-4765 • *z/OS Language Environment Concepts Guide*, SA22-7567 • *z/OS Language Environment Programming Guide*, SA22-7561 • *z/OS Language Environment Programming Reference*, SA22-7562 • *IBM Open Class Library User's Guide*, SC09-4811 • *IBM Open Class Library Reference*, SC09-4812
Coding and binding programs with interlanguage calls	• *z/OS C/C++ Programming Guide*, SC09-4765 • *C/C++ Language Reference*, SC09-4815 • *z/OS Language Environment Programming Guide*, SA22-7561 • *z/OS Language Environment Writing Interlanguage Applications*, SA22-7563 • *z/OS DFSMS Program Management*, SC27-1130
Compiling, binding, and running programs	• *z/OS 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 DFSMS Program Management*, SC27-1130
Compiling and binding applications in the z/OS UNIX environment	• *z/OS 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 DFSMS Program Management*, SC27-1130
Debugging programs	• README file • *Debug Tool User's Guide and Reference*, SC09-2137 • *z/OS C/C++ User's Guide*, SC09-4767 • *z/OS C/C++ Messages*, GC09-4819 • *z/OS 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 • z/OS Messages Database, available on the z/OS Library page on the World Wide Web (`http://www.ibm.com/servers/eserver/zseries/zos/bkserv`)
Using shells and utilities in the z/OS UNIX environment	• *z/OS 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 environment	• *z/OS C/C++ Run-Time Library Reference*, SA22-7821
Using the ISO Standard C++ Library to write portable C/C++ code that complies with ISO standards	• Standard C++ Library Reference, available on the z/OS C/C++ library page on the World Wide Web (`http://www.ibm.com/software/ad/c390/czos/czosdocs.html`)

Table 2. Publications by Task (continued)

Tasks	Books
Porting a UNIX 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, 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: `http://www-1.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 C/C++ User's Guide*, SC09-4767 • CBC.SCCNDOC(APAR) on z/OS C/C++ product tape
Tuning Large C/C++ Applications on z/OS UNIX System Services	• IBM Redbook called *Tuning Large C/C++ Applications on z/OS UNIX System Services*, which is available at: `http://www.redbooks.ibm.com/abstracts/sg245606.html`
C/C++ Applications on OS/390 UNIX	• IBM Redbook called *C/C++ Applications on OS/390 UNIX*, which is available at: `http://www.redbooks.ibm.com/abstracts/sg245992.html`
Performance considerations for XPLINK	• IBM Redbook called *XPLink: OS/390® Extra Performance Linkage*, which is available at: `http://www.redbooks.ibm.com/abstracts/sg245991.html`

Note: For information on using the prelinker, see "Appendix A. Prelinking and Linking z/OS C/C++ Programs" on page 485. As of Release 4, this appendix contains information that was previously in the chapter on prelinking and linking z/OS C/C++ programs in *z/OS C/C++ User's Guide*. It also contains prelinker information that was previously in *z/OS C/C++ Programming Guide*.

Hardcopy Books

The following z/OS C/C++ books are available in hardcopy:

- *z/OS C/C++ Run-Time Library Reference*, SA22-7821
- *z/OS C/C++ User's Guide*, SC09-4767
- *z/OS C/C++ Messages*, GC09-4819
- *z/OS C/C++ Programming Guide*, SC09-4765
- *z/OS C Curses*, SA22-7820
- *z/OS C/C++ Compiler and Run-Time Migration Guide*, GC09-4913
- *Debug Tool User's Guide and Reference*, SC09-2137

You can purchase these books on their own, or as part of a set. You receive *z/OS C/C++ Compiler and Run-Time Migration Guide*, GC09-4913 at no charge. Feature code 8009 includes the remaining books.

Softcopy Books

The z/OS C/C++ publications are supplied in PDF and BookMaster® formats on the following CD: *z/OS Collection*, SK3T-4269. They are also available at the following Web site:

`http://www.ibm.com/software/ad/c390/czos/czosdocs.html`

To read a PDF file, use the Adobe Acrobat Reader. If you do not have the Adobe Acrobat Reader, you can download it for free from the Adobe Web site:

`http://www.adobe.com`

To read a file in BookManager® format, use BookManager READ/MVS Version 1 Release 3 (5695-046) or the Library Reader™ for DOS, OS/2® or Windows® supplied on the CD-ROMs containing BookManager books.

If your system has BookManager Read installed, you can enter the command BOOKMGR to start BookManager and display a list of books available to you. If you know the name of the book that you want to view, you can use the OPEN command to open the book directly.

Note: If your workstation does not have graphics capability, BookManager Read cannot correctly display some characters, such as arrows and brackets.

You can also browse the books on the World Wide Web by clicking on "The Library" link on the z/OS home page. The web address for this page is:

`http://www.ibm.com/servers/eserver/zseries/zos/bkserv`

Softcopy Examples

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

- *C/C++ Language Reference*, SC09-4815
- *z/OS C/C++ User's Guide*, SC09-4767
- *z/OS C/C++ Programming Guide*, SC09-4765
- *IBM Open Class Library User's Guide*, SC09-4811

In the following books, a label on an example indicates that the example is distributed in softcopy. The label is the name of a member in the data sets `CBC.SCCNSAM` or the directory `/usr/lpp/ioclib/sample`. The labels have the form CCN*xyyy* or CLB*xyyy*, where *x* refers to a publication:

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

Examples labelled as CCN*xyyy* appear in *C/C++ Language Reference*, *z/OS C/C++ Programming Guide*, and *z/OS C/C++ User's Guide*. Examples labelled as CLB*xyyy* appear in the *z/OS C/C++ User's Guide*. Additional IBM Open Class Samples are provided as softcopy only. They can be found in the `/usr/lpp/ioclib/sample` directory.

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

Additional information on z/OS C/C++ is available on the World Wide Web on the z/OS C/C++ home page at:

`http://www.ibm.com/software/ad/c390/czos`

This page contains late-breaking information about the z/OS C/C++ product, including the compiler, the class libraries, and utilities. It also contains a tutorial on the source level interactive debugger. There are links to other useful information, such as the z/OS C/C++ information library and the libraries of other z/OS elements that are available on the Web. The z/OS C/C++ home page also contains samples that you can download, and 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.

Accessing licensed books on the Web

z/OS licensed documentation in PDF format is available on the Internet at the IBM Resource Link Web site at:

`http://www.ibm.com/servers/resourcelink`

Licensed books are available only to customers with a z/OS license. Access to these books requires an IBM Resource Link Web userid and password, and a key code. With your z/OS order you received a memo that includes this key code.

To obtain your IBM Resource Link Web userid and password log on to:

`http://www.ibm.com/servers/resourcelink`

To register for access to the z/OS licensed books:

1. Log on to Resource Link using your Resource Link userid and password.
2. Click on **User Profiles** located on the left-hand navigation bar.
3. Click on **Access Profile.**
4. Click on **Request Access to Licensed books.**
5. Supply your key code where requested and click on the **Submit** button.

If you supplied the correct key code you will receive confirmation that your request is being processed. After your request is processed you will receive an e-mail confirmation.

Note: You cannot access the z/OS licensed books unless you have registered for access to them and received an e-mail confirmation informing you that your request has been processed.

To access the licensed books:

1. Log on to Resource Link using your Resource Link userid and password.
2. Click on **Library**.
3. Click on **zSeries**.
4. Click on **Software**.
5. Click on **z/OS**.
6. Access the licensed book by selecting the appropriate element.

Using LookAt to look up message explanations

LookAt is an online facility that allows you to look up explanations for z/OS messages and system abends.

Using LookAt to find information is faster than a conventional search because LookAt goes directly to the explanation.

LookAt can be accessed from the Internet or from a TSO command line.

You can use LookAt on the Internet at:

`http://www.ibm.com/servers/eserver/zseries/zos/bkserv/lookat/lookat.html`

To use LookAt as a TSO command, LookAt must be installed on your host system. You can obtain the LookAt code for TSO from the LookAt Web site by clicking on **News and Help** or from the *z/OS Collection*, SK3T-4269.

To find a message explanation from a TSO command line, simply enter: **lookat** *message-id* as in the following example:

```
lookat iec192i
```

This results in direct access to the message explanation for message IEC192I.

To find a message explanation from the LookAt Web site, simply enter the message ID. You can select the release if needed.

Note: Some messages have information in more than one book. For example, IEC192I has routing and descriptor codes listed in *z/OS MVS Routing and Descriptor Codes*. For such messages, LookAt prompts you to choose which book to open.

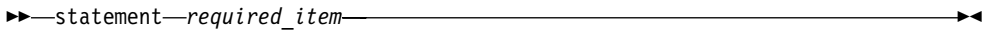
How to Read the Syntax Diagrams

This book describes the syntax for commands, directives, and statements, using the following structure:

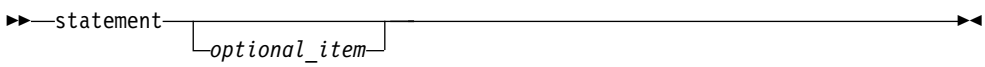
- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  A double right arrowhead indicates the beginning of a command, directive, or statement. A single right arrowhead indicates that it is continued on the next line. In the following diagrams, "statement" represents a command, directive, or statement.

- Required items are on the horizontal line (the main path).



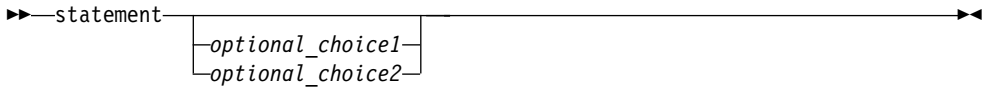
- Optional items are below the main path.



- If you can choose from two or more items, they are vertical in a stack.

  If you *must* choose one of the items, one item of the stack is on the main path.

  If choosing one of the items is optional, the entire stack is below the main path.

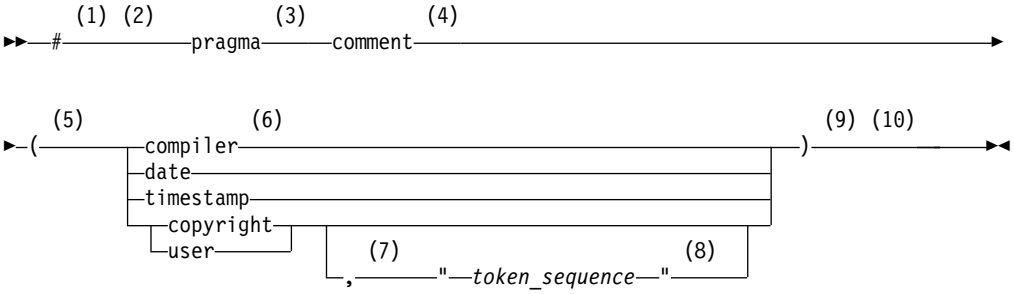


- An arrow that returns to the left above the main line indicates an item that you can repeat.

  A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.
- Keywords are not italicized, and should be entered exactly as shown (for example, `pragma`). You must spell keywords exactly as shown in the syntax diagram. Variables are in lowercase italics (in hardcopy), for example, *identifier*. They represent user-supplied names or values.
- If the syntax diagram shows punctuation marks, parentheses, arithmetic operators, or other nonalphanumeric characters, you must enter them as part of the syntax.

Note: You do not always require the white space between tokens. You should, however, include at least one blank space between tokens unless otherwise specified.

The following syntax diagram example shows the syntax for the `#pragma comment` directive.



Notes:

1 This is the start of the syntax diagram.

2 The symbol `#` must appear first.

3 The keyword `pragma` must follow the `#` symbol.

4 The keyword `comment` must follow the keyword `pragma`.

5 An opening parenthesis must follow the keyword `comment`.

6 The comment type must be entered only as one of the following: `compiler`, `date`, `timestamp`, `copyright`, or `user`.

7 If the comment type is `copyright` or `user`, and an optional character string is following, a comma must be present after the comment type.

8 A character string must follow the comma. The character string must be enclosed in double quotation marks.

9 A closing parenthesis is required.

10 This is the end of the syntax diagram.

The following examples of the `#pragma comment` directive are syntactically correct according to the diagram above:

```
#pragma comment(date)
#pragma comment(user)
#pragma comment(copyright,"This text will appear in the module")
```

Chapter 2. About IBM z/OS C/C++

The C/C++ feature of the IBM z/OS licensed program provides support for C and C++ application development on the z/OS platform. The C/C++ feature is based on the C/C++ for MVS/ESA™ product.

z/OS C/C++ includes:
- A C compiler (referred to as the z/OS C compiler)
- A C++ compiler (referred to as the z/OS C++ compiler)
- Support for a set of C++ class libraries that are available with the base z/OS operating system
- Application Support Class and Collection Class Library source
- A mainframe interactive Debug Tool (optional)
- Performance Analyzer host component, which supports the IBM C/C++ Productivity Tools for OS/390 product
- A set of utilities for C/C++ application development

IBM offers the C language on other platforms, such as the AIX®, OS/400®, VM/ESA®, and VSE/ESA™ operating systems. The AIX and OS/400 operating systems also offer the C++ language.

Changes for z/OS V1R2

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

C++ Compiler Compliant with ISO C++ 1998 Standard

z/OS V1R2 C/C++ includes a C++ compiler which is fully compliant with the ISO C++ 1998 Standard. This includes support for:
- namespaces and associated keywords namespace and using
- new type bool and associated keywords bool, true, and false
- new class member modifying keywords mutable and explicit
- new casts and associated keywords static_cast, dynamic_cast, reinterpret_cast, and const_cast
- new template model and its associated keyword typename
- Run Time Type Identification (RTTI) and its associated keyword typeid. The C++ compiler does not support exported template definitions, nor does it allow overloading functions in ways that differ only in the linkage type of function pointer parameters.
- The C++ Standard Library, including the Standard Template Library (STL) and other library features of ISO C++ 1998

The associated run-time library DLLs use the XPLINK convention and require an XPLINK-capable run-time environment. Environments such as CICS will require the continued use of the previous environment and be compiled with the `NOXPLINK` and `TARGET(OSV2R10)` options.

Note: The OS/390 V2R10 C/C++ compiler is being shipped along with the new z/OS V1R2 C/C++ compiler. Existing C++ programs may need source code changes in order to conform to the ISO C++ 1998 Standard. If you do not require the 1998 Standard, you can use the OS/390 V2R10 compiler

to avoid these source changes, but you will not get the benefits of the new features introduced in the new compiler.

IBM Open Class Library
: The IBM Open Class (IOC) is a library of C++ classes. z/OS V1R2 includes a new level of IOC, which is consistent with that shipped in VisualAge® C++ for AIX Version 5.0. This is intended to ease porting from AIX, but is not intended for use in new development. Support will be withdrawn in a future release. New application development involving C++ classes should make use of the C++ Standard Library instead of the IBM Open Class Library.

Large File Support in Standard I/O Stream Class Library
: Large file support enables access to hierarchical file system (HFS) files that are over 2 GB in size, using the C++ Standard Library.

IPA Support for XPLINK
: This feature combines the highest optimization level (IPA) for z/OS C/C++ with its high performance linkage (XPLINK).

  - XPLINK is a function call linkage introduced in OS/390 V2R10 which offers significant performance increments 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. When all functions are compiled with the XPLINK option, function pointers can be used without restriction.
  - InterProcedural Analysis (IPA) was introduced in OS/390 V1R2 C. IPA performs optimizations across compilation units, which exposes more optimization opportunities. This complements the traditional approach of optimizing within compilation units.

Enhanced ASCII Support
: z/OS V1R2 C/C++ provides enhanced ASCII support that simplifies porting of applications from ASCII platforms. It provides the ability to:

  - Build ASCII-based applications by producing object code with ASCII string literals and character constants, and a flag that identifies applications as ASCII or EBCDIC
  - Use Unicode-based wide characters (wchar_t) in ASCII-based applications
  - Transparently call native ASCII run-time library functions from ASCII-based applications
  - Process user-defined ASCII multi-byte code pages with user-supplied code set related methods
  - Create ASCII-based locale objects, which allow processing of ASCII data natively at run time

  The ability to produce code that contains ASCII string literals and character constants allows ASCII-dependent logic to continue working as on ASCII platforms thus eliminating the need to find all such places in the code and convert them to EBCDIC.

New Compiler Options
: z/OS V1R2 C/C++ introduces the following new compiler options:

  - ASCII
  - BITFIELD

- CHARS
- ENUM
- HALTONMSG (C++ only)
- KEYWORD (C++ only)
- LANGLVL (added new suboptions)
- OBJECTMODEL (C++ only)
- RTTI (C++ only)
- STATICINLINE (C++ only)
- SUPPRESS (C++ only)
- TEMPLATERECOMPILE (C++ only)
- TEMPLATEREGISTRY (C++ only)
- TMPLPARSE (C++ only)

Compiler Option Whose Syntax Has Been Changed for C++ to Match C

- INLINE: The default inline behavior has changed. In previous versions of C++, the threshold and limit values were 100 and 2000, respectively. These are now 100 and 1000.

Compiler Options That Are No Longer Supported

In z/OS V1R2 C/C++ the following compiler options are no longer supported:

- DECK: The replacement for DECK functionality that routes output to DD: SYSPUNCH is to use OBJECT(DD:SYSPUNCH). Alternatively, you can replace all references to DD:SYSPUNCH in your JCL with DD:SYSLIN, and use the OBJECT option.
- GENPCH
- HWOPTS: use ARCHITECTURE instead
- LANGLVL(COMPAT)
- OMVS: use OE instead
- SRCMSG
- SYSLIB: use SEARCH instead
- SYSPATH: use SEARCH instead
- TARGET(OSV1R2 | OSV1R3 | OSV2R4 | OSV2R5)
- USEPCH
- USERLIB: use LSEARCH instead
- USERPATH: use LSEARCH instead

Compiler Option Whose Default Value Has Changed

In z/OS V1R2 the default setting for the following compiler options has changed:

- Default is now DIGRAPH, both for C and C++
- Default is now INFO(LAN) for C++
- Default is now ROSTRING, both for C and C++

Built-in Functions for Floating-Point and Other Hardware Instructions

z/OS V1R2 has new built-in functions for floating-point and other hardware instructions, making these accessible to C/C++ programs. For information on using these built-in functions, see the appendix on built-in functions in *z/OS C/C++ Programming Guide*.

Limitations of Enhanced ASCII

This section explains under what conditions you can use Enhanced ASCII.

- A subset of C headers and functions is provided in ASCII. For more information, see *z/OS C/C++ Run-Time Library Reference*.
- The only way to get to the ASCII version of functions and the external variables `environ` and `tzname` is to use the appropriate IBM header files.
- ASCII applications may read, but not update, environment variables using the external variable. Updates to the environment variables using `environ` in an ASCII application cause unpredictable results and may result in an abend. Language Environment maintains two equivalent arrays of environment variables when running an ASCII application, one with EBCDIC encodings and the other with ASCII encodings. All ASCII compile units that use the `environ` external variable must include `<stdlib.h>` so that `environ` can be mapped to access the ASCII encoded environment strings. If `<stdlib.h>` is not included, `environ` will refer to the EBCDIC representation of the environment variable strings.

Enhanced ASCII provides limited conversion of ASCII to EBCDIC, and EBCDIC to ASCII. The character set or alphabet that is associated with any locale consists of the following:

- A common, XPG4-defined subset of characters such as POSIX portable characters
- A unique, locale-specific subset of characters such as NLS characters

The conversion only applies to the portable subset of characters that are associated with a locale. Only the EBCDIC IBM-1047 encoding of portable characters is supported.

You might encounter unexpected results in the following situations:

- If Enhanced ASCII applications are run in locales that contain non-Latin Alphabet Number 1 NLS characters, C-RTL functions might copy some of the locale's non-Latin 1 NLS characters into buffers that the application is writing to stdout or another HFS files. The non-Latin Alphabet Number 1 NLS characters would then cause problems during automatic conversion.
- Language Environment applications must select non-English message files. If your NATLANG run-time option is not UEN or ENU, messages directed to the Language Environment message file are converted to ASCII. These messages would cause problems during automatic conversion to EBCDIC.

z/OS Language Environment Downward Compatibility

z/OS Language Environment provides downward compatibility support. Assuming that you have met the required programming guidelines and restrictions, described in the *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 or OS/390. In C and C++, downward compatibility support is provided through the C/C++ `TARGET` compiler option. See "TARGET" on page 199 for details on this compiler option.

For example, a company may use z/OS V1R2 with Language Environment on a development system where applications are coded, link-edited, and tested, while using any supported lower release of OS/390 or z/OS Language Environment on their production systems where the finished application modules are used.

Downward compatibility support is not the roll-back of new function to prior releases of the operating system. Applications developed that exploit the downward

compatibility support must not use any Language Environment function that is unavailable on the lower release of OS/390 or z/OS where the application will be used.

The downward compatibility support includes toleration PTFs for lower releases of OS/390 or z/OS to assist in diagnosing applications that do not meet the programming requirements for this support. (Specific PTF numbers can be found in the PSP buckets.)

The downward compatibility support provided by z/OS Language Environment and by the toleration PTFs does not change Language Environment's upward compatibility. That is, applications coded and link-edited with one release of OS/390 or z/OS Language Environment will continue to run on later releases of OS/390 or z/OS Language Environment without the need to recompile or re-link edit the application, independent of the downward compatibility support.

Downward compatibility is supported in earlier releases of OS/390 C/C++ (from Version 2 Release 6), but in OS/390 V2R6, the user is required to copy header files and link-edit `SYSLIB` data sets from the deployment release of OS/390. Starting with OS/390 Version 2 Release 10, the current level header files and `SYSLIB` can be used (the user no longer has to copy header files and `SYSLIB` data sets from the deployment release).

The C/C++ Compilers

The following sections describe the C and C++ languages and the z/OS 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, type genericity or templates, and an extensive library. For a detailed description of the differences between z/OS C++ and z/OS C, refer to the *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 C and C++ Compilers

The C and C++ compilers, when used with z/OS Language Environment, offer many features to help your work:

- Optimization support:
  - Algorithms to take advantage of the S/390® architecture to get better optimization for speed and use of computer resources through the `OPTIMIZE` and `IPA` compiler options.
  - The `OPTIMIZE` compiler option, which instructs the compiler to optimize the machine instructions it generates to produce faster-running object code, which improves application performance at run time.
  - Interprocedural Analysis (IPA), to perform optimizations across compilation units, thereby optimizing application performance at run time.
- 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 both load-on-reference and load-on-demand DLLs. When your program refers to a function or variable which resides in a DLL, z/OS C/C++ generates code to load the DLL and access the functions and variables within it. This is called *load-on-reference*. Alternatively, your program can use z/OS C library functions to load a DLL and look up the address of functions and variables within it. This is called *load-on-demand*. Your application code explicitly controls load-on-demand DLLs at the source level.

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

  With reentrancy, many users can simultaneously run a program. A reentrant program uses less storage if it is stored in the LPA (link pack area) or ELPA (extended link pack area) 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 C programmers can design programs that are naturally reentrant. For those programs that are not naturally reentrant, 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 z/OS Language Environment prelinker and program management binder. The z/OS C++ compiler always ensures that C++ programs are reentrant.
- `INLINE` compiler option

  Additional optimization capabilities are available with the `INLINE` compiler option.
- Locale-based internationalization support 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 programmers 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 programmers to integrate z/OS C/C++ code with existing applications.
- Exploitation of z/OS and z/OS UNIX technology.

  z/OS UNIX is an IBM implementation of the open operating system environment, as defined in the XPG4 and POSIX standards.
- Support for the following standards at the system level:

  - A subset of the extended multibyte and wide character functions as defined by *Programming Language C Amendment 1*. This is *ISO/IEC 9899:1990/Amendment 1:1994(E)*
  - *ISO/IEC 9945-1:1990(E)/IEEE POSIX 1003.1-1990*
  - A subset of *IEEE POSIX 1003.1a, Draft 6, July 1991*
  - *IEEE Portable Operating System Interface (POSIX) Part 2, P1003.2*
  - A subset 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*
  - A subset of *IEEE 754-1985 (R1990) IEEE Standard for Binary Floating-Point Arithmetic (ANSI)*, as applicable to the S/390 environment.
  - *X/Open CAE Specification, Network Services, Issue 4*
- Year 2000 support
- Support for the Euro currency

z/OS C Compiler Specific Features

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

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

z/OS C++ Compiler Specific Features

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

Class Libraries

z/OS V1R2 C/C++ provides the following class libraries, which are all thread-safe:

- C++ Standard Library, including the Standard Template Library (STL) and other library features of ISO C++ 1998
- IBM Open Class Library for z/OS V1R2
- IBM Open Class Library for OS/390 V2R10

Refer to *z/OS C/C++ Compiler and Run-Time Migration Guide* and *IBM Open Class Library User's Guide* for more details on the components of these libraries.

For new code and the most portable code you will want to use the new C++ Standard Library, which includes the following:

- The C++ Standard I/O Stream Library for performing input and output (I/O) operations

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

The IBM Open Class (IOC) is a comprehensive library of C++ classes that you can use to develop applications. z/OS V1R2 includes a new level of IOC which is consistent with that shipped in VisualAge C++ for AIX V5.0. This is intended to ease porting from AIX, but is not intended for use in new development. Support will be withdrawn in a future release.

The z/OS V1R2 IBM Open Class Library includes:

- The Application Support Class Library which provides the basic abstractions that are needed during the creation of most C++ applications, including String, Date, Time, and Decimal. The Application Support Class Library corresponds to the IOC member in the data sets.
- The Collection Class Library implements a wide variety of classical data structures such as stack, tree, list, hash table, and so on. The Collection Class Library provides developers with a consistent set of building blocks from which they can derive application objects. The library design exploits features of the C++ language such as exception handling and template support. The Collection Class Library corresponds to the COLL member in the data sets.

The z/OS V1R2 IBM Open Class enables you to choose between the C++ Standard I/O Stream and Complex Mathematics libraries, and the UNIX Systems Laboratories C++ Language System Release (USL) I/O Stream and Complex Mathematics libraries.

The OS/390 V2R10 IBM Open Class Library and USL class libraries include the following:

- The USL I/O Stream Class Library (corresponds to the IOSTREAM member in the data sets)
- The USL Complex Mathematics Class Library (corresponds to the COMPLEX member in the data sets)
- The Application Support Class Library (corresponds to the APPSUPP member in the data sets)
- The Collection Class Library (corresponds to the COLLECT member in the data sets)

Note: Retroactive to OS/390 Version 1 Release 3, the IBM Open Class Library is licensed with the base operating system. This enables applications to use this library at run time without having to license the z/OS C/C++ compiler features or to use the DLL Rename Utility.

IBM Open Class Library Source

The IBM Open Class Library Source consists of the following:

- Application Support Class Library source code
- Collection Class Library source code

Utilities

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

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

- The DSECT Conversion Utility to convert descriptive assembler DSECTs into z/OS C/C++ data structures.
- The localedef utility to read the locale definition file and produce a locale object that the locale-specific library functions can use.
- The makedepend utility to derive all dependencies in the source code and write these into the makefile for the make command to determine which source files to recompile, whenever a dependency has changed. This frees the user from manually monitoring such changes in the source code.

z/OS Language Environment provides the following utilities:

- The Object Library Utility (C370LIB) to update partitioned data set (PDS and PDSE) libraries of object modules and Interprocedural Analysis (IPA) object modules.
- The DLL Rename Utility to make selected DLLs a unique component of the applications with which they are packaged. The DLL Rename Utility does not support XPLINK.
- The prelinker which combines object modules that comprise a z/OS 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.

The Debug Tool

z/OS C/C++ supports program development by using the Debug Tool. This optionally available tool allows you to debug applications in their native host environment, such as CICS/ESA®, IMS/ESA®, DB2, and so on. The 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 the 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 hierarchical file system (HFS) files as source files.

Note: You can also use the `dbx` shell command to debug programs, as described in *z/OS UNIX System Services Command Reference*.

For further information, see “IBM C/C++ Productivity Tools for OS/390”.

IBM C/C++ Productivity Tools for OS/390

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:

- A Performance Analyzer to help you analyze, understand, and tune your C and C++ applications for improved performance

- A Distributed Debugger that allows you to debug C or C++ programs from the convenience of the workstation
- A 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

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 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 C or C++ application remotely from your workstation. Set a break point 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.

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

z/OS Language Environment

z/OS C/C++ exploits the C/C++ run-time environment and library of run-time services available with z/OS Language Environment (formerly OS/390 Language Environment, Language Environment for MVS™ & VM, Language Environment/370 and LE/370).

z/OS Language Environment consists of four language-specific run-time libraries, and Base Routines and Common Services, as shown below. z/OS Language Environment establishes a common run-time environment and common run-time services for language products, user programs, and other products.

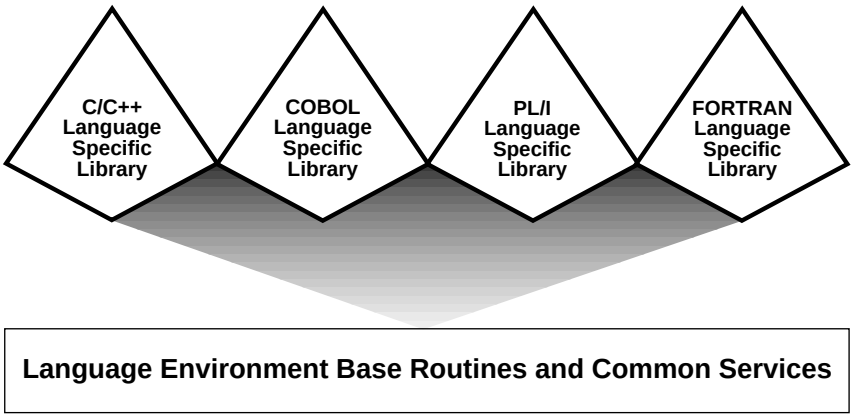


Figure 1. Libraries in z/OS Language Environment

The common execution environment is composed of data items and services that are included in library routines available to an application that runs in the environment. The z/OS Language Environment provides a variety of services:

- Services that satisfy basic requirements common to most applications. These include support for the initialization and termination of applications, allocation of storage, interlanguage communication (ILC), and condition handling.
- Extended services that are often needed by applications. z/OS C/C++ contains these functions within a library of callable routines, and include interfaces to operating system functions and a variety of other commonly used functions.
- Run-time options that help in the execution, performance, and diagnosis of your application.
- Access to operating system services; z/OS UNIX services are available to an application programmer or program through the z/OS C/C++ language bindings.
- Access to language-specific library routines, such as the z/OS C/C++ library functions.

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 *C/C++ Language Reference* for more information on linkage specification.

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

Notes on the Prelinking Process

Note that you cannot use the prelinker if you are using the `XPLINK` or `GOFF` compiler options. Also, IBM recommends using the binder without the prelinker whenever possible.

Prior to OS/390 V2R4 C/C++, the *z/OS C/C++ User's Guide* showed how to use the prelinker and linkage editor. Sections throughout the book discussed concepts of *prelinking* and *linking*. The prelinker was designed to process long names and support constructed reentrancy in earlier versions of the C complier on the MVS and OS/390 operating systems. The prelinker, shipped with the z/OS C/C++ run-time library, provides output that is compatible with the linkage editor, that is shipped with the binder.

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

Updates to the prelinking, linkage-editing, and loading functions that are performed by the binder are delivered through the binder. If you use the prelinker shipped with the z/OS C/C++ run-time library and the linkage editor (supplied through the binder) you have to apply the latest maintenance for the run-time library 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 10. Binding z/OS C/C++ Programs" on page 365.

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 (Partitioned Data Set Extended) member or an HFS file.

If you cannot use a PDSE member or HFS 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

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

- Long name maximum symbol length:
  - Long names currently processed by the binder are limited to 1024 characters. The prelinker supports up to (32 K - 1) characters. IBM intends to bring the binder limit in line with the prelinker in a future release.

The prelinker provided with z/OS Language Environment combines the object modules comprising a z/OS 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 HFS file).

Note: For further information on the binder, refer to the DFSMS home page at http://www.ibm.com/storage/software/sms/smshome.htm.

z/OS UNIX System Services (z/OS UNIX)

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

Together, the z/OS UNIX System Services, z/OS Language Environment, and z/OS 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

    BPXBATCH Allows you to submit batch jobs that run shell commands, scripts, or z/OS C/C++ executable files in HFS files from a shell session

    c89 Compiles, assembles, and binds z/OS UNIX C/C++ and assembler applications

    dbx Provides an environment to debug and run programs

    gencat Merges the message text source files message file (usually *.msg) into a formatted message catalog file (usually *.cat)

    iconv Converts characters from one code set to another

    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

    yacc Allows you to write compilers and other programs that parse input according to strict grammar rules
  - Support for other utilities such as:

    c++ Compiles, assembles, and binds z/OS UNIX C++ applications

    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

    dspcat Displays all or part of a message catalog

    dspmsg Displays a selected message from a message catalog
- The z/OS UNIX Debugger feature, which provides the dbx interactive symbolic debugger for z/OS UNIX applications
- Access to a hierarchical file system (HFS), with support for the POSIX.1 and XPG4 standards
- z/OS C/C++ I/O routines, which support using HFS 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 C/C++ DLLs

z/OS UNIX 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 are a familiar environment for C/C++ application

development. If you are familiar with existing MVS development environments, you may find that the z/OS UNIX environment can enhance your productivity. Refer to *z/OS UNIX System Services User's Guide* for more information on the Shell and Utilities.

z/OS C/C++ Applications with z/OS UNIX C/C++ Functions

All z/OS UNIX 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, and the system() and signal handling functions where the behavior is different between POSIX/XPG4 and ISO. Refer to *z/OS C/C++ Run-Time Library Reference* for more information about requirements for each function.

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

I/O Interfaces

The 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 C/C++ extension to the ISO standard.

TCP/IP Sockets I/O
z/OS UNIX 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 OE sockets). The slightly different interface of the X/Open CAE Specification, Networking Services, Issue 4, is supplied as an additional choice. This interface is known as X/Open Sockets.

The 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 C++ I/O Stream libraries support formatted I/O in C++. You can code sophisticated I/O statements easily and clearly, and define input and output for your own data types. This helps improve the maintainability of programs that use input and output.

File Types

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

Virtual Storage Access Method (VSAM) Data Sets

z/OS C/C++ has native support for three types of VSAM data organization:

- 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 *z/OS C/C++ Programming Guide*.

Hierarchical File System Files

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

Memory Files

Memory files are temporary files that reside in memory. For improved performance, you can direct input and output to memory files rather than to devices. 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.

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).

Additional I/O Features

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

- Large file support, which enables I/O to and from hierarchical file system (HFS) files that are larger than 2 GB
- User error handling for serious I/O failures (SIGIOERR)

- Improved sequential data access performance through enablement of the DFSMS/MVS® 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 require no dynamic loading of z/OS Language Environment libraries. It also allows you to tailor your application for better utilization of the 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 z/OS Language Environment services. Note that if you do not use z/OS Language Environment services, only some built-in functions and a limited set of C/C++ run-time library functions are available to you.
- You can substitute the z/OS C language in place of assembler 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 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 C/C++, you can write programs that utilize the power of other IBM products and subsystems:

- Cross System Product (CSP)

  Cross System Product/Application Development (CSP/AD) is an application generator that provides ways to interactively define, test, and generate application programs to improve productivity in application development. Cross System Product/Application Execution (CSP/AE) takes the generated program and executes it in a production environment.

  Note: You cannot compile CSP applications with the z/OS C++ compiler. However, your z/OS C++ program can use interlanguage calls (ILC) to call z/OS C programs that access CSP.
- Customer Information Control System (CICS)

  You can use the CICS/ESA 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.

Note: Code preprocessed with CICS/ESA versions prior to V4R1 is not supported for z/OS C++ applications. z/OS C++ code preprocessed on CICS/ESA V4R1 cannot run under CICS/ESA V3R3.

- DB2 Universal Database™ (UDB) 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 C/C++ compilers. The DB2 program processes requests, then returns control to the application program.

- Data Window Services (DWS)

  The Data Window Services (DWS) part of the Callable Services Library allows your C or C++ program to manipulate temporary data objects that are known as TEMPSPACE and VSAM linear data sets.

- Information Management System (IMS)

  The Information Management System/Enterprise Systems Architecture (IMS/ESA) product provides support for hierarchical databases.

- Interactive System Productivity Facility (ISPF)

  z/OS C/C++ provides access to the Interactive System Productivity Facility (ISPF) Dialog Management Services. A dialog is the interaction between a user and a computer. The dialog interface contains display, variable, message, and dialog services as well as other facilities that are used to write interactive applications.

- Graphical Data Display Manager (GDDM)

  GDDM provides 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

- Query Management Facility (QMF)

  z/OS 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.

- 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 interface JNI. Java code, which has been compiled using the High Performance Compiler for Java (HPCJ), will support the JNI interface. Calls to C or C++ do not distinguish between compiled Java and interpreted Java.

Additional Features of z/OS C/C++

Feature	Description
long long Data Type	The z/OS C/C++ compiler supports long long as a native data type when the compiler option `LANGLVL(LONGLONG)` is turned on. This option is turned on by default by the compiler option `LANGLVL(EXTENDED)`.
Multibyte Character Support	z/OS 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 C library functions and encoded in units of one length. These normalized characters are called wide characters. Conversions between multibyte and wide characters can be performed by string conversion functions such as `wcstombs()`, `mbstowcs()`, `wcsrtombs()`, and `mbsrtowcs()`, as well as the family of wide-character I/O functions. Wide-character data can be represented by the `wchar_t` data type.
Extended Precision Floating-Point Numbers	z/OS C/C++ provides three S/390 floating-point number data types: single precision (32 bits), declared as `float`; double precision (64 bits), declared as `double`; and extended precision (128 bits), declared as `long double`. Extended precision floating-point numbers give greater accuracy to mathematical calculations. As of Release 6, z/OS C/C++ also supports IEEE 754 floating-point representation. By default, `float`, `double`, and `long double` values are represented in IBM S/390 floating point format. However, the IEEE 754 floating-point representation is used if you specify the `FLOAT(IEEE754)` compile option. For details on this support, see “FLOAT” on page 116.
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 C/C++ provides message text in either American English or Japanese. You can dynamically switch between these two languages.
Locale Definition Support	z/OS C/C++ provides a locale definition utility that supports the creation of separate files of internationalization data, or locales. Locales can be used at run time to customize the behavior of an application to national language, culture, and coded character set (code page) requirements. Locale-sensitive library functions, such as `isdigit()`, use this information.
Coded Character Set (Code Page) Support	The z/OS 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	Selected library functions, such as string and character functions, are built into the compiler to improve performance execution. Built-in functions are compiled into the executable, and no calls to the library are generated.
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 the *z/OS C/C++ Programming Guide*
Multitasking Facility (MTF)	Multitasking is a mode of operation where your program performs two or more tasks at the same time. z/OS C provides a set of library functions that perform multitasking. These functions are known as the Multitasking Facility (MTF). MTF uses the multitasking capabilities of z/OS to allow a single z/OS C application program to use more than one processor of a multiprocessing system simultaneously. **Note:** `XPLINK` is not supported in an MTF environment. You can also use threads to perform multitasking with or without `XPLINK`, as described in the *z/OS C/C++ Programming Guide*.

Feature	Description
Packed Structures and Unions	z/OS C provides support for packed structures and unions. Structures and unions may be packed to reduce the storage requirements of an z/OS C program or to define structures that are laid out according to COBOL or PL/I structure layout rules.
Fixed-point (Packed) Decimal Data	z/OS 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. The Application Support Class Library provides the Binary Coded Decimal Class for C++ programs.
Long Name Support	For portability, external names can be mixed case and up to 1024 characters in length. For C++, the limit applies to the mangled version of the name.
System Calls	You can call commands or executable modules using the `system()` function under z/OS, z/OS UNIX, and TSO. You can also use the `system()` function to call EXECs on z/OS and TSO, or Shell scripts using z/OS UNIX.
Exploitation of ESA	Support for z/OS, IMS/ESA, Hiperspace expanded storage, and CICS/ESA allows you to exploit the features of the ESA.
Exploitation of hardware	Use the `ARCHITECTURE` compiler option to select the minimum level of machine architecture on which your program will run. `ARCH(2)` instructs the compiler to generate faster instruction sequences that are available only on newer machines. `ARCH(3)` also generates these faster instruction sequences and enables support for IEEE 754 Binary Floating-Point instructions. Code compiled with `ARCH(2)` runs on G2, G3, G4, and 2003 processors and code compiled with `ARCH(3)` runs on a G5 or G6 processor, and follow-on models. For more information, refer to “ARCHITECTURE” on page 86. Use the `TUNE` compiler option to optimize your application for a specific machine architecture. `TUNE` impacts performance only; it does not impact the processor model on which you will be able to run your application. `TUNE(3)` optimizes your application for the newer G4, G5, and G6 processors. `TUNE(2)` optimizes your application for other architectures. For more information, refer to “TUNE” on page 213.
Built-in Functions for Floating-Point and Other Hardware Instructions	Use built-in functions for floating-point and other hardware instructions that are otherwise inaccessible to C/C++ programs. See the appendix on built-in functions in *z/OS C/C++ Programming Guide*.

Part 2. User's Reference

This part reviews the basic steps for compiling, binding, and running z/OS C/C++ programs under the z/OS operating system. It also describes the options available to you at compile, IPA link, bind, and run time.

Chapter 3. z/OS C Example

This chapter outlines the basic steps for compiling, binding, and running 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 chapter may be unfamiliar. Refer to “Chapter 8. Compiling” on page 301, “Chapter 10. Binding z/OS C/C++ Programs” on page 365, and “Chapter 12. Running a C or C++ Application” on page 413 for a detailed description on compiling, binding, and running a C program.

This chapter 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 C/C++ Programs” on page 485.

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

Example of a C Program

The following example shows a simple z/OS 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 2. 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 2. Celsius-to-Fahrenheit Conversion (Part 2 of 2)

CCNUAAN

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

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

Figure 3. 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 334.

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 C language, see the *C/C++ Language Reference* and the *z/OS C/C++ Run-Time Library Reference*.

Compiling, Binding, and Running the z/OS C Example

In general, you can compile, bind, and run z/OS C programs under z/OS batch, TSO, or the z/OS shell. You cannot run the IPA Link step under TSO, or under z/OS batch by using the ISPF interface. For more information, see “Chapter 8. Compiling” on page 301, “Chapter 10. Binding z/OS C/C++ Programs” on page 365, and “Chapter 12. Running a C or C++ Application” on page 413.

This book 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.

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 4. JCL to Compile, Bind, and Run the Example Program Using the EDCCBG Procedure

In Figure 4, 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 5 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 5. 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 C program (CCNUAAM) in PETE.TEST.C(CTOF) and the header file in PETE.TESTHDR.H(CCNUAAN).

Use the following set of TSO commands to compile, bind, and run the example program:

1. Ensure that the z/OS Language Environment run-time libraries SCEERUN and SCEERUN2, and the z/OS C compiler are in the STEPLIB, LPALST, or LNKLST concatenation.
2. Compile the z/OS C source. You can use the REXX EXEC CC to invoke the z/OS 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 311.
3. 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 10. Binding z/OS C/C++ Programs” on page 365 for more information.
4. Run the program:

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

   CALL runs CTOF from PETE.TEST.C.LOAD with the default run-time options in effect. See “Chapter 12. Running a C or C++ Application” on page 413 for more information.

Non-XPLINK and XPLINK Under the z/OS UNIX Shell

Put the source in the HFS.

1. Ensure that the z/OS Language Environment run-time libraries SCEERUN and SCEERUN2, and the z/OS C compiler are in the STEPLIB, LPALST, or LNKLST concatenation.
2. From the z/OS shell type:

   ```
   cp "//'cbc.sccnsam(ccnuaam)'" ccnuaam.c
   cp "//'cbc.sccnsam(ccnuaan)'" ccnuaan.h
   ```
3. Compile and bind:

```
c89 -o ctof ccnuaam.c

-- or, for XPLINK --

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

4. Run the program:

```
./ctof
```

Chapter 4. z/OS C++ Examples

This chapter outlines the basic steps for compiling, binding, and running z/OS C++ example programs under z/OS batch, TSO, or the z/OS shell.

If you have not yet compiled a C++ program, some concepts in this chapter may be unfamiliar. Refer to “Chapter 8. Compiling” on page 301, “Chapter 10. Binding z/OS C/C++ Programs” on page 365, and “Chapter 12. Running a C or C++ Application” on page 413 for a detailed description on compiling, binding, and running an C++ program.

The example programs that this chapter describes are shipped with the z/OS 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 HFS in `/usr/lpp/ioclib/sample`.

Example of a C++ Program

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

The program is written in object-oriented fashion. 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 2 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 C++ compiler in data sets CBC.SCCNSAM(CCNUBRC) and CBC.SCCNSAM(CCNUBRH).

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

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

using namespace std;

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

  protected:
    int curYear, curDay;
    static const int dateLen;
    static const int numMonths;
    static const int numDays[];
};
const int Date::dateLen = 10;
const int Date::numMonths = 12;
const int Date::numDays[Date::numMonths] = {
  31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};
class BirthDate : public Date {
  public:
    BirthDate();
    BirthDate(const char *birthText);
    int DaysOld() { return(DaysSince(text)); }

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

Figure 6. Header File for the Biorhythm Example (Part 1 of 2)

```
class BioRhythm {
  friend ostream& operator<<(ostream&, 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;
    static const int eCycle;
    static const int iCycle;
};

const int BioRhythm::pCycle=23;     // Physical cycle - 23 days
const int BioRhythm::eCycle=28;     // Emotional cycle - 28 days
const int BioRhythm:iCycle=33;     // Intellectual cycle - 33 days

ostream& operator<<(ostream&,BioRhythm&);
```

Figure 6. 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 "ccn3ubrh.h"  //BioRhythm class and Date class

using namespace std;

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);
}

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);
}

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);
}
```

Figure 7. z/OS C++ Biorhythm Example Program (Part 1 of 2)

```
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);
  }

  for (i=0;i<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 7. z/OS C++ Biorhythm Example Program (Part 2 of 2)

Compiling, Binding, and Running the z/OS C++ Example

In general, you can compile, bind, and run z/OS C++ programs under z/OS batch, TSO, or the z/OS shell. You cannot run the IPA Link step under TSO, or under z/OS batch by using the ISPF interface. For more information, see “Chapter 8. Compiling” on page 301, “Chapter 10. Binding z/OS C/C++ Programs” on page 365, and “Chapter 12. Running a C or C++ Application” on page 413.

This book 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`.

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 TODAY'S DATE IN THE FORM YYYY/MM/DD
//GO.SYSIN DD *
  1997/10/19
/*
```

Figure 8. JCL to Compile, Bind, and Run the Example Program Using the CBCCBG Procedure

In Figure 8, 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 TODAY'S DATE IN THE FORM YYYY/MM/DD
//GO.SYSIN DD *
  1997/10/19
/*
```

Figure 9. 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 8. Compiling” on page 301, “Chapter 10. Binding z/OS C/C++ Programs” on page 365, and “Chapter 12. Running a C or C++ Application” on page 413.

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).

Use the following set of TSO commands to compile, bind, and run the example program:

1. Ensure that the z/OS Language Environment run-time libraries SCEERUN and SCEERUN2, the z/OS class library DLLs, and the z/OS C++ compiler are in the STEPLIB, dynamic LPA, or Link List concatenation.
2. Compile the z/OS C++ source. You can use the REXX EXEC CXX to invoke the z/OS 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 311.
3. 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 10. Binding z/OS C/C++ Programs” on page 365.
4. Run the program:

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

   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 416.

Non-XPLINK and XPLINK Under the z/OS UNIX Shell

Put the source in the HFS. From the z/OS shell type:

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

In this example, the current working directory is used, 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.

1. Ensure that the z/OS Language Environment run-time libraries `SCEERUN` and `SCEERUN2`, the z/OS class library DLLs, and the z/OS C++ compiler are in the `STEPLIB`, dynamic LPA, or Link List concatenation.
2. 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.
3. Run the program:

   ```
   ./bio
   ```

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 C++ program that uses templates to perform simple operations on linked lists. It is shipped with IBM Open Class and it resides in the HFS in the directory /usr/lpp/ioclib/sample/clb3atmp. This program consists of ten files that are described and illustrated below.

The main program, CLB3ATMP.CPP (see “CLB3ATMP.CPP” on page 55) has two class templates: `List` (in the file CLB3ALST.C that uses CLB3ALST.H) and `Iterator` (in the file CLB3AITR.C that uses CLB3AITR.H). `List` is a template of a linked list, and `Iterator` is a template that walks a `List` class.

CLB3ALST.C

```
#include "clb3alst.h"
template <class Item> void List<Item>::append(Item item) {
  GetNode();
  cur—>node = item;
}

template <class Item> void List<Item>::GetNode() {
  if (cur) {
    cur—>next = new Node<Item>;
    cur = cur—>next;
  }
  else {
    cur = new Node<Item>;
    head = cur;
  }
  cur—>next = 0;
  return;
}
```

Figure 10. Template of a Linked List

CLB3ALST.H

```
                    ??=ifndef _CBCLIST_H_
                    ??=ifdef __COMPILER_VER__
                    ??=pragma filetag ("IBM-1047")
                    ??=endif
                    ??=define _CBCLIST_H_  1
                    #pragma nomargins nosequence
                    #pragma checkout (suspend)

  template <class Item> struct Node {
    Item node;
    struct Node<Item> *next;
  };

  template <class Item> class List {
     public:

      List() :cur(0), head(0) {}

        ~List() {}
      void append(Item item);
      Node<Item> *cur, *head;

     private:
      void GetNode();
  };

                    #pragma checkout (resume)
                    #endif
```

Figure 11. Header file for CLB3ALST.C

CLB3AITR.C

```
#include "clb3aitr.h"
template <class Item> Item& Iterator<Item>::operator++() {
  node = cur->node;
  cur  = cur->next;
  return(node);
}

template <class Item> int Iterator<Item>::eol() {
  return(cur == 0);
}

template <class Item> void Iterator<Item>::reset() {
  cur = head;
}
```

Figure 12. Template of an Iterator

CLB3AITR.H

```
                    ??=ifndef _CBCITER_H_
                    ??=ifdef __COMPILER_VER__
                    ??=pragma filetag ("IBM-1047")
                    ??=endif
                    ??=define _CBCITER_H_    1
                    #pragma nomargins nosequence
                    #pragma checkout (suspend)
#include "clb3alst.h"
template <class Item> class Iterator {
  public:
    Iterator(List<Item>& list)
             :cur(list.head), head(list.head) {}

    Item& operator++();
    int eol();
    void reset();

  private:
    Node<Item> *cur;
    Node<Item> *head;
    Item node;
};

                     #pragma checkout (resume)
                     #endif
```

Figure 13. Header file for CLB3AITR.C

There are two template functions, `max(T,T)` (in the file CLB3AMAX.C which uses CLB3AMAX.H), and `min(T,T)` (in the file CLB3AMIN.C which uses CLB3AMIN.H). `max(T,T)` returns the maximum object of two objects, and `min(T,T)` returns the minimum object of two objects.

CLB3AMAX.H

```
template <class T> T& max(T a, T b);
```

CLB3AMAX.C

```
template <class T> T& max(T a, T b) {
  if (a > b) return(a);
  else return(b);
}
```

CLB3AMIN.H

```
template <class T> T& min(T a, T b);
```

CLB3AMIN.C

```
template <class T> T& min(T a, T b) {
  if (a < b) return(a);
  else return(b);
}
```

There is one simple class, `String`, defined in the file CLB3ASTR.H.

CLB3ASTR.H

```
                    ??=ifndef _CBCSTR_H_
                    ??=ifdef __COMPILER_VER__
                    ??=pragma filetag ("IBM-1047")
                    ??=endif
                    ??=define _CBCSTR_H_    1
                    #pragma nomargins nosequence
                    #pragma checkout (suspend)
#include <iostream.h>
#include <iomanip.h>
class String {
  friend ostream& operator<<(ostream&, String&);
  friend istream& operator>>(istream&, String&);
  public:
    String() {
      str = new char[1];
      str??(0??)= '\0';
    }
    String(const char *s) {
      const int len = strlen(s);
      str = new char[len+1];
      memcpy(str, s, len+1);
    }

      ~String() {
      delete str;
    }
    void replace(const char *s) {
      const int len = strlen(s);
      char *newStr = new char[len+1];
      delete str;
      str = newStr;
      memcpy(str, s, len+1);
    }
    int operator >(String& rhs) {
      if (strcmp(str, rhs.string()) >= 0)

             return 1;
      else
             return 0;
    }
    int operator <(String& rhs) {
      if (strcmp(str, rhs.string()) >= 0)
             return 0;
      else
             return 1;
    }
    const char *string() {
      return(str);
    }
  private:
    char *str;
};

                    #pragma checkout (resume)
                    #endif
```

Figure 14. Definition of the String Class

CLB3ATMP.CPP

```
#include "clb3amax.h"
#include "clb3amin.h"
#include "clb3alst.h"
#include "clb3aitr.h"
#include "clb3astr.h"
#include <string.h>
#include <iostream.h>
#include <iomanip.h>

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

template <class Item> void IOList<Item>::write() {
  Iterator<Item> iter(list);
  while (!iter.eol()) {
    cout << ' ' << ++iter;
  }
  cout << endl;
}

template <class Item> void IOList<Item>::read(const char *msg) {
  Item item;
  cout << msg << endl;
  while (cin >> item) {
    list.append(item);
  }
}

ostream& operator<<(ostream& os, String& str) {
  os << str.string() << endl;
  return(os);
}

istream& operator>>(istream& is, String& str) {
  char tmpStr[80];
  cin.width(79);
  is >> tmpStr;
  str.replace(tmpStr);
  return(is);
}
```

Figure 15. z/OS C++ Template Program (Part 1 of 2)

```
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";

  cout << min(name1, name2) << " comes before "
       << max(name1, name2) << endl;

  int num1 = 23;
  int num2 = 28;

  cout << min(num1, num2) << " comes before "
       << max(num1, num2) << endl;

  return(0);
}
```

Figure 15. z/OS C++ Template Program (Part 2 of 2)

Compiling, Binding, and Running the C++ Template Example

This section 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

To compile, bind, and run the template example program under z/OS batch use the following JCL, changing <userhlq> to your own user prefix:

1. Ensure that z/OS Language Environment run-time library and the z/OS C++ compiler are in `STEPLIB`, `LPALST`, or the `LNKLST` concatenation.
2. Use the following JCL to compile, bind, and run the template example. In the example JCL, change `<userhlq>` to your own user prefix.

CCNUNCL

```
//Jobcard info
//PROC JCLLIB ORDER=(CBC.SCCNPRC,
//   CEE.SCEEPROC)
//*
//* Compile MAIN program, creating an object deck and a TEMPLATE PDS
//* of the source code.  The TEMPLATE PDS of source code will be
//* written to the default TEMPLATE PDS '<userhlq>.TEMPINC'
//*
//MAINCC  EXEC CBCC,          * Compile main program
//        OUTFILE='<userhlq>.SAMPLE.OBJ(CLB3ATMP),DISP=SHR ',
//        CPARM='OPTF(DD:COPTS)'
//SYSIN DD PATH='/usr/lpp/ioclib/sample/clb3atmp/clb3atmp.cpp'
//*
//COPTS   DD *
  SEARCH('CEE.SCEEH.+','CBC.SCLBH.+')
  LSEARCH(/usr/lpp/ioclib/sample/clb3atmp/)
  TEMPINC(//TEMPINC)
/*
//*
//* Compile PDS of TEMPLATE source code.  Direct template source file
//* creation to this PDS with the TEMPINC option.  Then, if any
//* TEMPLATE compilation creates new members, they will be created
//* in this PDS.  The compiler will detect this and automatically
//* compile the newly created members as part of this step.
//*
//TMPCC   EXEC CBCC,          * Compile PDS of templates
//        INFILE='<userhlq>.TEMPINC',
//        OUTFILE='<userhlq>.TEMPINC.OBJ,DISP=SHR ',
//        CPARM='OPTF(DD:COPTS)'
//COPTS   DD *
  SEARCH('CEE.SCEEH.+','CBC.SCLBH.+')
  LSEARCH(/usr/lpp/ioclib/sample/clb3atmp/)
  TEMPINC
/*
//*
//* Make the PDS of template objects have long named aliases used
//* for autocall by using the EDCLIB utility with the DIR command.
//*
//GENLIB  EXEC EDCLIB,        * Create Template Library
//        OPARM='DIR',
//        LIBRARY='<userhlq>.TEMPINC.OBJ'
//*
//* Bind the program --- specify the template library on the
//* bind autocall library.
//*
//BIND    EXEC CBCB,          * Bind main program
//        INFILE='<userhlq>.SAMPLE.OBJ(CLB3ATMP)',
//        OUTFILE='<userhlq>.SAMPLE.LOAD(CLB3ATMP),DISP=SHR'
//BIND.SYSLIB DD
//             DD
//             DD
//             DD DSN=<userhlq>.TEMPINC.OBJ,DISP=SHR
//GO      EXEC CBCG,
//        INFILE='<userhlq>.SAMPLE.LOAD',
//        GOPGM=CLB3ATMP
//GO.SYSIN DD *
 1 2 5 3 7 8 3 2 10 11
/*
```

Figure 16. JCL to Compile, Bind and Run the Template Example

Under TSO

To compile, bind, and run the example program under TSO, follow these steps:

1. Ensure that thez/OS Language Environment run-time library, the z/OS Class Library DLLs, and the z/OS C++ compiler are in STEPLIB, LPALST, or the LNKLST concatenation.
2. Compile the source files:
   a. `cxx /usr/lpp/ioclib/sample/clb3atmp/clb3atmp.cpp (lsearch(/usr/lpp/ioclib/sample/clb3atmp/) 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/ioclib/sample/clb3atmp/) 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 311 for more information.
3. 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 426.
4. 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 382.
5. 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 416.

Under the z/OS UNIX Shell

To compile, bind, and run the template example program under the z/OS shell, follow these steps:

1. Ensure that the z/OS Language Environment run-time library and the z/OS C++ compiler are in STEPLIB, LPALST, or the LNKLST concatenation.
2. Copy sample files to your own directory, as follows:

   ```
   cp /usr/lpp/ioclib/sample/clb3atmp/* your_dir/.
   ```
3. 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.

4. Run the program:

   ```
   ./clb3atmp
   ```

Chapter 5. Compiler Options

This chapter 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 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 8. Compiling” on page 301, and “Appendix D. Cataloged Procedures and REXX EXECs” on page 551 for details.
- In your own JCL procedure, by passing a parameter string to the compiler.
- In an options file. See “OPTFILE | NOOPTFILE” on page 171 for details.
- For z/OS C, in a `#pragma options` preprocessor directive within your source file. See “Specifying z/OS C Compiler Options Using #pragma Options” on page 64 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.
- In the utilities `c89`, `cc`, or `c++`, by using the `-Wc` option to pass options to the compiler.
- In the ISPF panels that are used to invoke the z/OS C/C++ compiler in foreground and background.

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

AGGRCOPY	
ALIAS	(C compile only)
ANSIALIAS	(C compile and C++ compile only)
ARCHITECTURE	
ARGPARSE	
ASCII	
BITFIELD	(C and C++ compile only)
CHARS	(C and C++ compile only)
COMPACT	
COMPRESS	
CONVLIT	
CSECT	
CVFT	(C++ compile only)
DLL	
EXECOPS	
EXPORTALL	(C compile and C++ compile only)
FLOAT	
GOFF	
GONUMBER	
IGNERRNO	
INITAUTO	

INLINE	
IPA	
KEYWORD	(C++ compile only)
LANGLVL	
LIBANSI	
LOCALE	
LONGNAME	
MAXMEM	
OBJECTMODEL	
OPTIMIZE	
PLIST	
REDIR	
RENT	(C compile and IPA Link step only)
ROCONST	(C and C++ compile only)
ROSTRING	(C and C++ compile only)
ROUND	
RTTI	(C++ compile only)
SERVICE	
SPILL	
START	
STRICT	
STRICT_INDUCTION	
TARGET	
TEMPLATERECOMPILE	(C++ compile only)
TEMPLATEREGISTRY	(C++ compile only)
TMPLPARSE	(C++ compile only)
TEST	
TUNE	
UPCONV	(C compile only)
XPLINK	

IPA Considerations

The following sections explain what you should be aware of if you request Interprocedural Analysis (IPA) through the `IPA` option. Refer to the *z/OS C/C++ Programming Guide* for an overview of IPA before you use the `IPA` compiler option.

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 that affect the way the compiler generates a regular object module have the same effect on how the IPA compile step generates an object module with `IPA (OBJECT)`.
- Compiler options for `IPA(OBJONLY)` compiles are the same as for `NOIPA` compiles.
- In some situations, you must specify a compiler option on the IPA Compile step if you want the benefit of the option on the IPA Link step. In some situations, you must specify the option again on the IPA Link step.
- Some compiler options have special behavior or restrictions other than what is described above.

- `#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 chapter, 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 3 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 3. 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.

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, IPA link-time optimizations cannot be performed for this compilation unit because no IPA information is available.

Using Special Characters

Under TSO

When HFS file names contain the special characters blank, backslash, and double quote, 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 quote, less than <, and greater than >

For example:

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

Under the z/OS Shell

The z/OS 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. See “Appendix F. c89 — Compile, link-edit and assemble a z/OS C program and create an executable 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 quote (') within a string as two single quotes (''), 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 quotes (''''), as follows:

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

Special characters in HFS 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 HFS file names that are referenced in the `PARM` statement. The special characters are: backslash, blank, and double quote. 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 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 the *C/C++ Language Reference*.

The #pragma options preprocessor directive must appear before the first z/OS 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
ASCII	CHECKOUT
ENUMSIZE	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

The c89 and c++ utilities specify most compiler options when they call the z/OS C/C++ compiler. Therefore, #pragma options and other #pragma directives that are overridden by command line options should not be used. For example, if you compile using c89, and have #pragma langlvl (EXTENDED) in your source, c89 uses LANGLVL(ANSI). This is because c89 specifies ANSI explicitly when it calls the compiler.

To change compiler options, use the corresponding c89 or c++ option. For example, use -I to set the search option that specifies where to search for #include files. If there is no corresponding c89 or c++ option, use -Wc. For example, specify -Wc,expo to export all functions and variables.

For a complete description of c89, c++ and related utilities, refer to “Appendix F. c89 — Compile, link-edit and assemble a z/OS C program and create an executable file” on page 577 or to the *z/OS UNIX System Services Command Reference*.

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 preceeded with two slashes are HFS 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 quote is needed for specifying a full file name with a high-level qualifier.

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 C/C++ is installed. 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`, `cc`, and `c++` utilities that run in the z/OS UNIX shell specify certain compiler options in order to support POSIX standards. For a complete description of these utilities, refer to "Appendix F. c89 — Compile, link-edit and assemble a z/OS C program and create an executable 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 the above 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 the above z/OS UNIX 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`, `cc`, and `c++` 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`, `cc`, or `c++` 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 C Compiler Listing" on page 221 and "Using the z/OS C++ Compiler Listing" on page 255 for more information.

The `c89` utilities remap 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 "Appendix F. c89 — Compile, link-edit and assemble a z/OS C program and create an executable 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 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 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)
```

Note that the locale option is set according to the environment where the cc, c89, and c++ utilities 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 "Appendix F. c89 — Compile, link-edit and assemble a z/OS C program and create an executable file" on page 577 for more information.
- z/OS UNIX environment variable settings: {_INCDIRS}, {_INCLIBS}, and {_CSYSLIB}. They are normally set during compiler installation to reflect the compiler and run-time include libraries. Refer to "Environment Variables" on page 589 for more information.

Refer to "SEARCH | NOSEARCH" on page 187 for more information on SEARCH.

For the remainder of the compiler options, the c89, cc,or c++ utilities 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 {_OPTIONS}

(_C89_OPTIONS, _CC_OPTIONS, _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. The table that follows lists the compiler options in alphabetical order, their abbreviations, and the defaults that are supplied by IBM. Suboptions inside square brackets are optional.

The C, C++, and IPA Link columns, which are shown in the table below, 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. ″C″ refers to a C language compile step. ″C++″ refers to a C++ language compile step. These options are accepted regardless of whether these are for NOIPA, IPA (OBJONLY), or IPA(NOLINK).

Table 4. 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 80
AGGREGATE | NOAGGREGATE	NOAGG	✓		✓	See 81
ALIAS[(name)] | NOALIAS	NOALI	✓			See 82
ANSIALIAS | NOANSIALIAS	ANS	✓	✓	✓	See 83
ARCHITECTURE(*n*)	TARGET(OSV2R10 and above): ARCH(2) TARGET(OSV2R9 and below): ARCH(0)	✓	✓	✓	See 86
ARGPARSE | NOARGPARSE	ARG	✓	✓	✓	See 88
ASCII | NOASCII	NOASCII	✓	✓	✓	See 89
ATTRIBUTE[(FULL)] | NOATTRIBUTE	NOATT		✓	✓	See 90
BITFIELD(SIGNED|UNSIGNED)	BITF(UNSIGNED)	✓	✓	✓	See 91
CHARS(SIGNED | UNSIGNED)	CHARS(UNSIGNED)	✓	✓	✓	See 91
CHECKOUT(subopts) | NOCHECKOUT	NOCHE	✓		✓	See 92
COMPACT | NOCOMPACT	NOCOMPACT	✓	✓	✓	See 94
COMPRESS | NOCOMPRESS	NOCOMPRESS	✓	✓	✓	See 96
CONVLIT[(subopts)] | NOCONVLIT[(subopts)]	NOCONV (, WCHAR)	✓	✓	✓	See 97
CSECT | NOCSECT	NOCSE for NOGOFF or CSE() for GOFF	✓	✓	✓	See 99
CVFT | NOCVFT	CVFT		✓		See 102
DEFINE(name1[= | =def1], name2[= | =def2],...)	no default user definitions	✓	✓	✓	See 103
DIGRAPH | NODIGRAPH	DIGR	✓	✓	✓	See 104
DLL(CBA | NOCBA) | NODLL(CBA | NOCBA)	NODLL(NOCBA)	✓		✓	See 106
DLL(CBA | NOCBA)	DLL(NOCBA)		✓	✓	See 106
ENUMSIZE(subopts)	ENUM(SMALL)	✓	✓	✓	See 108
EVENTS[(filename)] | NOEVENTS	NOEVENT	✓	✓	✓	See 110

Table 4. Compiler Options, Abbreviations, and IBM Supplied Defaults (continued)

Compiler Option (Abbreviated Names are underlined)	IBM Supplied Default	C	C++	IPA Link	More Information
EXECOPS | NOEXECOPS	EXEC	✓	✓	✓	See 111
EXH | NOEXH	EXH		✓		See 111
EXPMAC | NOEXPMAC	NOEXP	✓	✓	✓	See 112
EXPORTALL | NOEXPORTALL	NOEXPO	✓	✓	✓	See 113
FASTTEMPINC | NOFASTTEMPINC	NOFASTT		✓		See 114
FLAG(severity) | NOFLAG	FL(I)	✓	✓	✓	See 115
FLOAT(subopts)	FLOAT(HEX, FOLD, NOMAF, NORRM, NOAFP or AFP). For ARCH(2) the default is NOAFP. For ARCH(3) or higher, the default is AFP.	✓	✓	✓	See 116
GOFF | NOGOFF	NOGOFF	✓	✓	✓	See 120
GONUMBER | NOGONUMBER	NOGONUM	✓	✓	✓	See 121
HALT(num)	HALT(16)	✓	✓	✓	See 123
HALTONMSG(msgno) | NOHALTONMSG	NOHALTON	✓	✓	✓	See 123
IGNERRNO | NOIGNERRNO	NOIGNER	✓	✓	✓	See 124
INFO[(subopts)] | NOINFO	IN(LAN)		✓		See 125
INITAUTO(number [,word]) | NOINITAUTO	NOINITA	✓	✓	✓	See 127
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 128
INLRPT[(filename)] | NOINLRPT[(filename)]	NOINLR	✓	✓	✓	See 132
IPA[(subopts)] | NOIPA[(subopts)]	NOIPA(NOLINK, OBJECT, OPT, NOLIST, NOGONUMBER, NOATTRIBUTE, NOXREF, LEVEL(1),NOMAP, DUP, ER, NONCAL, NOUPCASE, NOCONTROL)	✓	✓	✓	See 133
KEYWORD(name)|NOKEYWORD(name)	Recognizes all C++ keywords		✓	✓	See 138
LANGLVL(subopts)	LANG(EXTENDED)	✓	✓	✓	See 139
LIBANSI | NOLIBANSI	NOLIB	✓	✓	✓	See 149
LIST[(filename)] | NOLIST [(filename)]	NOLIS	✓	✓	✓	See 150
LOCALE[(name)] | NOLOCALE	NOLOC	✓	✓	✓	See 152
LONGNAME | NOLONGNAME	C:NOLO C++: LO	✓	✓	✓	See 154

Table 4. Compiler Options, Abbreviations, and IBM Supplied Defaults (continued)

Compiler Option (Abbreviated Names are underlined)	IBM Supplied Default	C	C++	IPA Link	More Information
LSEARCH(subopts) | NOLSEARCH	NOLSE	✓	✓	✓	See 155
MARGINS | NOMARGINS	NOMAR		✓		See 160
MARGINS(m,n) | NOMARGINS	V-format: NOMAR F-format: MAR(1,72)	✓		✓	See 160
MAXMEM(size) | NOMAXMEM	MAXM(2097152)	✓	✓	✓	See 162
MEMORY | NOMEMORY	MEM	✓	✓	✓	See 163
NAMEMANGLING	NAMEMANGLING(ANSI)		✓	✓	See 164
NESTINC(num) | NONESTINC	NONEST	✓	✓	✓	See 165
OBJECT[(filename)] | NOOBJECT [(filename)]	OBJ	✓	✓	✓	See 166
OBJECTMODEL(subopt)	OBJECTMODEL(COMPAT)		✓	✓	See 167
OE[(filename)] | NOOE[(filename)]	NOOE	✓	✓	✓	See 169
OFFSET | NOOFFSET	NOOF	✓	✓	✓	See 170
OPTFILE[(filename)] | NOOPTFILE[(filename)]	NOOPTF	✓	✓	✓	See 171
OPTIMIZE[(level)] | NOOPTIMIZE	NOOPT	✓	✓	✓	See 173
PHASEID | NOPHASEID	NOPHASEID	✓	✓	✓	See 176
PLIST(HOST | OS)	PLIST(HOST)	✓	✓	✓	See 176
PORT(PPS | NOPPS) | NOPORT(PPS | NOPPS)	NOPORT(NOPPS)		✓		See 177
PPONLY[(subopts)] | NOPPONLY [(subopts)]	NOPP	✓	✓	✓	See 179
REDIR | NOREDIR	RED	✓	✓	✓	See 181
RENT | NORENT	NORENT	✓		✓	See 182
ROCONST | NOROCONST	C: NOROC C++: ROC	✓	✓	✓	See 183
ROSTRING | NOROSTRING	ROS	✓	✓	✓	See 184
ROUND(opt)	For IEE: ROUND(N) For HEX: ROUND(Z)	✓	✓	✓	See 185
RTTI|NORTTI	NORTTI		✓	✓	See 186
SEARCH(opt1,opt2,...) | NOSEARCH	For C++, SE(//'CEE.SCEEH.+, //'CBC.SCLBH.+') For C, SE(//'CEE.SCEEH.+')	✓	✓	✓	See 187
SEQUENCE | NOSEQUENCE	NOSEQ		✓		See 189
SEQUENCE(m,n) | NOSEQUENCE	V-format: NOSEQ F-format: SEQ(73,80)	✓		✓	See 189
SERVICE(string) | NOSERVICE	NOSERV	✓	✓	✓	See 188
SHOWINC | NOSHOWINC	NOSHOW	✓	✓	✓	See 190
SOURCE[(filename)] | NOSOURCE[(filename)]	NOSO	✓	✓	✓	See 191
SPILL(size) | NOSPILL[(size)]	SP(128)	✓	✓	✓	See 192
SSCOMM | NOSSCOMM	NOSS	✓		✓	See 194
START | NOSTART	STA	✓	✓	✓	See 195
STATICINLINE | NOSTATICINLINE	NOSTATICI		✓	✓	See 196
STRICT | NOSTRICT	STRICT	✓	✓	✓	See 196

Table 4. Compiler Options, Abbreviations, and IBM Supplied Defaults (continued)

Compiler Option (Abbreviated Names are underlined)	IBM Supplied Default	C	C++	IPA Link	More Information
STRICT_INDUCTION | NOSTRICT_INDUCTION	NOSTRICT_INDUC	✓	✓	✓	See 197
SUPPRESS(msg-no) | NOSUPPRESS	NOSUPP	✓	✓	✓	See 198
TARGET(suboption)	TARG(LE, CURRENT)	✓	✓	✓	See 199
TEMPINC[(filename)] | NOTEMPINC[(filename)]	PDS: TEMP(TEMPINC) HFS Directory: TEMP(./tempinc)		✓		See 205
TEMPLATERECOMPILE | NOTEMPLATERECOMPILE	TEMPLATEREC		✓	✓	See 206
TEMPLATEREGISTRY | NOTEMPLATEREGISTRY	NOTEMPL		✓	✓	See 207
TERMINAL | NOTERMINAL	TERM	✓	✓	✓	See 209
TEST[(subopts)] | NOTEST[(subopts)]	C: NOTEST (HOOK, SYM, BLOCK, LINE, PATH) C++: NOTEST(HOOK)	✓	✓	✓	See 209
TMPLPARSE(subopts)	TMPLPARSE(NO)		✓		See 208
TUNE(n)	TUN(3)	✓	✓	✓	See 213
UNDEFINE(name)	no default	✓	✓	✓	See 215
UPCONV | NOUPCONV	NOUPC	✓		✓	See 216
WSIZEOF| NOWSIZEOF	NOWSIZEOF	✓	✓	✓	See 216
XPLINK[(subopts)] | NOXPLINK[(subopts)]	NOXPL	✓	✓	✓	See 217
XREF | NOXREF	NOXR	✓	✓	✓	See 220

Compiler Options for File Management

These options specify the data set or HFS directory where the compiler stores output files, and direct the search for include files by the compiler.

Table 5. Compiler Options for File Management

Option	Description	C Compile	C++ Compile	IPA Link	More Information
FASTTEMPINC	Defers generating object code until the final version of all template definitions have been determined. Then, a single compilation pass generates the final object code, resulting in improved compilation time when recursive templates are used in an application.		✓		See 114
IPA(CONTROL)	Indicates the name of the control file that contains additional directives for the IPA Link step. This option only affects the IPA Link step.	✓	✓	✓	See 137
LSEARCH	Specifies the libraries or disks to be scanned for user include files.	✓	✓	✓	See 155
MEMORY	Improves compile-time performance by using a MEMORY file in place of a work file, if possible.	✓	✓	✓	See 163

Table 5. Compiler Options for File Management (continued)

Option	Description	C Compile	C++ Compile	IPA Link	More Information
OBJECT	Produces an object module, and stores it in the file that you specify, or in the data set associated with `SYSLIN`.	✓	✓	✓	See 166
OE	Specifies that file names used in compiler options and include directives should be interpreted as HFS file names when the file name provided is ambiguous. Also specifies that POSIX.2 standard rules for include file searching should be used.	✓	✓	✓	See 169
OPTFILE	Directs the compiler to look for compiler options in the file specified.	✓	✓	✓	See 171
SEARCH	Specifies the libraries or disks to be scanned for system include files.	✓	✓	✓	See 187
TEMPINC	Places template instantiation files in the PDS or HFS directory specified.		✓		See 205
TEMPLATERECOMPILE	Helps to manage dependencies between compilation units that have been compiled using the `TEMPLATEREGISTRY` option.		✓	✓	See 206
TEMPLATEREGISTRY	Maintains records of all templates as they are encountered in the source and ensures that only one instantiation of each template is made.		✓	✓	See 207

Options That Control the Preprocessor

These options specify how the preprocessor runs.

Table 6. Summary of Compiler Options for Preprocessor

Option	Description	C Compile	C++ Compile	IPA Link	More Information
CONVLIT	Turns on string literal codepage conversion.	✓	✓	✓	See 97
DEFINE	Defines preprocessor macro names.	✓	✓	✓	See 103
DIGRAPH	Allows you to use additional digraphs in both C and C++ applications.	✓	✓	✓	See 104
LOCALE	Specifies the locale to be used at compile time.	✓	✓	✓	See 152
PPONLY	Specifies that only the preprocessor is to be run and not the compiler.	✓	✓	✓	See 179
UNDEFINE	Removes any value its argument may have.	✓	✓	✓	See 215

Options That Control the Processing of an Input Source File

These options allow you to control your z/OS C/C++ processing of an input source file.

Table 7. Summary of Compiler Options Used for Input Source File Processing Control

Option	Description	C Compile	C++ Compile	IPA Link	More Information
HALT	Specifies that the compiler stop processing files when it returns an error severity level of *n* or above.	✓	✓	✓	See 123
HALTONMSG	Instructs the C++ front-end to stop after the compilation phase when it encounters the specified msg_number.		✓	✓	See 123
MARGINS	Identifies position of source to be scanned by the compiler.	✓	✓	✓	See 160
NESTINC	Specifies the number of nested include files to be allowed.	✓	✓	✓	See 165
SEQUENCE	Specifies the columns used for sequence numbers.	✓	✓	✓	See 189

Options That Control the Compiler Listing

These options control the generation of a compiler listing, and the information that goes into the listing.

Table 8. Compiler Options That Control Listings

Option	Description	C Compile	C++ Compile	IPA Link	More Information
AGGREGATE	Lists structures and unions, and their sizes. The IPA Link step accepts but ignores this option.	✓		✓	See 81
ATTRIBUTE	For C++ compile, generates a cross reference section showing attributes for each symbol and External Symbol Cross Reference section. For IPA Link, it also generates the Storage Offset Listing if IPA objects were created using the C compiler with `XREF`, `IPA(ATTR)`, or `IPA(XREF)` options and the symbols for the current partition were not coalesced.		✓	✓	See 90
EXPMAC	Lists all expanded macros. You must use the `SOURCE` option with `EXPMAC`.	✓	✓	✓	See 112
INLINE(,REPORT,,)	Generates a report on the status of inlined functions.	✓	✓	✓	See 128
INLRPT	Generates a report on the status of inlined functions.	✓	✓	✓	See 132
IPA(MAP)	Generates the following listing sections for the IPA Link step: Object File Map, Source File Map, Compiler Options Map, Global Symbols Map, Partition Map. This option only affects the IPA Link step.	✓	✓	✓	See 133
LIST	Includes the object module in the compiler listing, in assembler-like code.	✓	✓	✓	See 150
OFFSET	Lists offset addresses relative to entry points of functions. The `LIST` option must be used with `OFFSET`.	✓	✓	✓	See 170
SHOWINC	Lists include files if `SOURCE` option is specified.	✓	✓	✓	See 190
SOURCE	Lists source file.	✓	✓	✓	See 191

Table 8. Compiler Options That Control Listings (continued)

Option	Description	C Compile	C++ Compile	IPA Link	More Information
XREF	For C/C++, generates a cross reference listing showing file/line definition, reference and modification information for each symbol. Also generates the External Symbol Cross Reference and Static Map. If you specify the `XREF` option for the IPA Link step, it generates an External Symbol Cross Reference listing section for each partition and Static Map. The IPA Link step creates a Storage Offset listing section if you created your IPA objects with the C compiler and the `XREF`, `IPA(ATTR)`, or `IPA(XREF)` option, and if IPA did not coalesce the symbols for the current partition.	✓	✓	✓	See 220

Options That Control the IPA Object

These options control the content of the IPA object that is produced by the IPA Compile step.

Table 9. Compiler Options for IPA Object Control

Option	Description	C Compile	C++ Compile	IPA Link	More Information
IPA(ATTRIBUTE)	Saves information about symbol storage offsets in the IPA object file.	✓	✓	✓	See 134
IPA(GONUMBER)	Saves source line numbers in the IPA object file without generating line number tables. This option can only be specified for the IPA Compile step, if a combined conventional/IPA object file is requested.	✓	✓	✓	See 134
IPA(LIST)	Saves source line numbers in the IPA object file without generating a Pseudo Assembly listing. This option can only be specified for the IPA Compile step, if a combined conventional/IPA object file is requested.	✓	✓	✓	See 134
IPA(OBJECT)	Indicates whether a conventional (non-IPA)/IPA object is to be produced during the IPA Compile step.	✓	✓	✓	See 134
IPA(OPTIMIZE)	Generates information in the IPA object file that the compiler option `OPT` needs during IPA Link processing. `IPA(OPTIMIZE)` is the default setting. If you specify `IPA(NOOPTIMIZE)`, IPA will change the option to `IPA(OPTIMIZE)` and issue an informational message.	✓	✓	✓	See 134
IPA(XREF)	Saves information about symbol storage offsets in the IPA object file.	✓	✓	✓	See 134

Options That Control the IPA Link Step

These options control the IPA Link step.

Table 10. Compiler Options for IPA Link Control

Option	Description	C Compile	C++ Compile	IPA Link	More Information
IPA(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.	✓	✓	✓	See 137
IPA(LINK)	Instructs the compiler to perform IPA Link processing.	✓	✓	✓	See 137
IPA(NCAL)	Indicates whether library searches are performed during the IPA Link step 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 step input processing.	✓	✓	✓	See 137
IPA(UPCASE)	Determines whether an additional automatic library call pass is made for SYSLIB if unresolved references remain at the end of standard IPA Link step processing. Symbol matching is not case-sensitive in this pass.	✓	✓	✓	See 137

Options for Debugging and Diagnosing Errors

These options help you to detect and correct errors in your z/OS C/C++ program.

Table 11. Compiler Options for Debugging and Diagnostics

Option	Description	C Compile	C++ Compile	IPA Link	More Information
CHECKOUT	Gives informational messages for possible programming errors. The IPA Link step accepts but ignores this option.	✓		✓	See 92
EVENTS	Produces an events file that contains error information and source file statistics. The IPA Link step accepts but ignores this option.	✓	✓	✓	See 110
FLAG	Specifies the lowest severity level to be listed.	✓	✓	✓	See 115
GONUMBER	Generates line number tables for Debug Tool and error trace backs. The `TEST` option turns on `GONUMBER`.	✓	✓	✓	See 121
INFO	Generates informational messages.		✓		See 125
IPA(DUP)	Indicates whether a message and a list of duplicate symbols are written to the console during the IPA Link step. This option only affects the IPA Link step.	✓	✓	✓	See 133
IPA(ER)	Indicates whether a message and a list of unresolved symbols are written to the console during the IPA Link step. This option only affects the IPA Link step.	✓	✓	✓	See 133
PHASEID	Causes each compiler module (phase) to issue an informational message which identifies the compiler phase module name, product identifier, and build level.	✓	✓	✓	See 176

Table 11. Compiler Options for Debugging and Diagnostics (continued)

Option	Description	C Compile	C++ Compile	IPA Link	More Information
SERVICE	Places a string in the object module, which is displayed in the traceback if the application fails abnormally.	✓	✓	✓	See 188
SUPPRESS	Prevents the batch compiler, or driver informational, or warning messages from being displayed or added to the listings.	✓	✓	✓	See 198
TERMINAL	Directs diagnostic messages to be displayed on the terminal.	✓	✓	✓	See 209
TEST	Generates information that the Debug Tool needs to debug your program.	✓	✓	✓	See 209
XPLINK (BACKCHAIN)	Generates a prolog that saves information about the calling function in the called function stack frame. This facilitates debugging using storage dumps, at a cost in execution time.	✓	✓	✓	See 217
XPLINK (STOREARGS)	Generates code to store arguments that are normally passed in registers, into the argument area. This facilitates debugging using storage dumps, at a cost in execution time.	✓	✓	✓	See 217

Options That Control the Programming Language Characteristics

These options allow you to control your z/OS C/C++ programming language characteristics.

Table 12. Summary of Compiler Options Used for Programming Language Characteristics Control

Option	Description	C Compile	C++ Compile	IPA Link	More Information
BITFIELD(UNSIGNED)	Specifies the default sign for bitfields.	✓	✓	✓	See 91
CHARS(UNSIGNED)	Instructs the compiler to treat all variables of type char as either signed or unsigned.	✓	✓	✓	See 91
KEYWORD	Controls whether the specified name is created as a keyword or an identifier whenever it appears in your C++ source.		✓	✓	See 138
LANGLVL	Specifies the language standard to be used.	✓	✓	✓	See 139
SSCOMM	Allows comments to be specified by two slashes (//). The IPA Link step accepts but ignores this option.	✓		✓	See 194
STATICINLINE	Treats an inline function as static instead of extern.		✓	✓	See 196

Table 12. Summary of Compiler Options Used for Programming Language Characteristics Control (continued)

Option	Description	C Compile	C++ Compile	IPA Link	More Information
TMPLPARSE	Controls whether parsing and semantic checking are applied to template definition implementations (function bodies and static data member initializers) or only to template instantiations.		✓		See 208
UPCONV	Preserves *unsignedness* during z/OS C/C++ type conversions. The IPA Link step accepts but ignores this option.	✓		✓	See 216

Options That Control Object Code Generation

These options are used to control how your z/OS C/C++ object code is produced.

Table 13. Summary of Compiler Options Used for Object Code Control

Option	Description	C Compile	C++ Compile	IPA Link	More Information
AGGRCOPY	Instructs the compiler whether or not the source and destination in structure assignments can overlap. (They cannot overlap according to ISO Standard C rules.)	✓	✓	✓	See 80
ALIAS	Generates `ALIAS` binder control statements for each required entry point.	✓			See 82
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 83
ARCHITECTURE	Specifies the architecture for which the executable program instructions are to be generated.	✓	✓	✓	See 86
ASCII	Provides native ASCII/NLS support.	✓	✓	✓	See 89
COMPACT	Controls choices made between those optimizations which tend to result in faster but larger code and those which tend to result in smaller but slower code.	✓	✓	✓	See 94
COMPRESS	Suppresses the generation of function names in the function control block, thereby reducing the size of your application's load module.	✓	✓	✓	See 96
CSECT	Instructs the compiler to generate `csect` names in the output object module.	✓	✓	✓	See 99
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.		✓		See 102
DLL	Generates object code for DLLs or DLL applications.	✓	✓	✓	See 106

Table 13. Summary of Compiler Options Used for Object Code Control (continued)

Option	Description	C Compile	C++ Compile	IPA Link	More Information
ENUMSIZE	Specifies the amount of storage occupied by enumerations	✓	✓	✓	See 108
EXH	Controls the generation of C++ exception handling code.		✓		See 111
EXPORTALL	Exports all externally defined functions and variables.	✓	✓	✓	See 113
FLOAT	Switches floating-point representation between IEEE and hexadecimal.	✓	✓	✓	See 116
GOFF	Instructs the compiler to produce an object file in the Generalized Object File Format.	✓	✓	✓	See 120
IGNERRNO	Informs the compiler that your application is not using `errno`, allowing the compiler to explore additional optimization opportunities for library functions in `LIBANSI`.	✓	✓	✓	See 124
INITAUTO	Directs the compiler to generate code to initialize automatic variables. Automatic variables require storage only while the function in which they are declared are active.	✓	✓	✓	See 127
INLINE	Inlines user functions into source and helps maximize optimization.	✓	✓	✓	See 128
IPA	Instructs the compiler to perform Interprocedural Analysis (IPA) processing.	✓	✓	✓	See 133
IPA(LEVEL)	Indicates the level of IPA optimization that the IPA Link step should perform.	✓	✓	✓	See 133
LIBANSI	Indicates whether or not functions with the name of an ANSI C library function are in fact ANSI C library functions.	✓	✓	✓	See 149
LONGNAME	Provides support for external names of mixed case and up to 1024 characters long.	✓	✓	✓	See 154
MAXMEM	Limits the amount of memory used for local tables of specific, memory intensive optimization.	✓	✓	✓	See 162
NAMEMANGLING	Enables the encoding of variable names into unique names so that linkers can separate common names in the language.		✓	✓	See 164
OBJECT	Produces an object module, and stores it in the file that you specify, or in the data set associated with `SYSLIN`.	✓	✓	✓	See 166
OBJECTMODEL	Sets the type of object model.	✓	✓	✓	See 167
OPTIMIZE	Improves run-time performance by introducing optimizations during code generation.	✓	✓	✓	See 173
RENT	Generates reentrant code. The IPA Link step accepts but ignores this option.	✓		✓	See 182
ROCONST	Informs the compiler that the `const` qualifier is respected by the program so that variables defined with the `const` keyword are not be overridden (for example, by a casting operation).	✓	✓	✓	See 183

Table 13. Summary of Compiler Options Used for Object Code Control (continued)

Option	Description	C Compile	C++ Compile	IPA Link	More Information
ROSTRING	Informs the compiler that string literals are read-only.	✓	✓	✓	See 184
ROUND	Sets the rounding mode for binary floating point numbers.	✓	✓	✓	See 185
SPILL	Specifies the size of the spill area to be used for compilation.	✓	✓	✓	See 192
START	Generates a `CEESTART` whenever necessary.	✓	✓	✓	See 195
STRICT	Affects the precision of floating point calculations.	✓	✓	✓	See 196
STRICT_INDUCTION	Instructs the compiler to disable loop induction variable optimizations.	✓	✓	✓	See 197
TARGET	Generates an object module for the targeted operating system or run-time library.	✓	✓	✓	See 199
TUNE	Specifies the architecture for which the executable program will be optimized.	✓	✓	✓	See 213
WSIZEOF	Causes the size of operator to return the widened size for function return types	✓	✓	✓	See 216
XPLINK	Instructs the compiler to generate extra performance linkage for function calls.	✓	✓	✓	See 217

Options That Control Program Execution

These options control the execution of your program

Table 14. Summary of Compiler Options for Program Execution

Option	Description	C Compile	C++ Compile	IPA Link	More Information
ARGPARSE	Parses arguments provided on the invocation line.	✓	✓	✓	See 88
ASCII	Provides ASCII/NLS support.	✓	✓	✓	See 89
EXECOPS	Allows you to specify run-time options on the invocation line.	✓	✓	✓	See 111
PLIST	Specifies that the original operating system parameter list should be available.	✓	✓	✓	See 176
REDIR	Allows redirection of `stderr`, `stdin`, and `stdout` from the invocation line.	✓	✓	✓	See 181
RTTI	Generates run-time type identification (RTTI) information for the typeid operator and the dynamic_cast operator.		✓	✓	See 186
TARGET	Generates an object module for the targeted operating system or run-time library.	✓	✓	✓	See 199

Portability Options

These options allow you to port your C++ code to the z/OS C++ compiler.

Table 15. Summary of Compiler Options for Portability

Option	Description	C Compile	C++ Compile	IPA Link	More Information
PORT	Adjusts the error recovery action that the compiler takes when it encounters an ill-formed `#pragma pack` directive.		✓		See 177

Description of Compiler Options

The following sections describe the compiler options and their usage. Compiler options are listed alphabetically. Syntax diagrams show the abbreviated forms of the compiler options.

AGGRCOPY

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
`AGGRCOPY(NOOVERLAP)`						

CATEGORY: Object Code Control

The `AGGRCOPY` option instructs the compiler whether or not the source and destination assignments for structures can overlap. The 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.

In the case of `structure` assignments, the compiler can generate faster code if no overlap is assumed. The `OVERLAP` suboption specifies that the source and destination in a structure assignment might overlap. The `NOOVERLAP` option specifies that they do not, and that the compiler can assume this when generating code.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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.

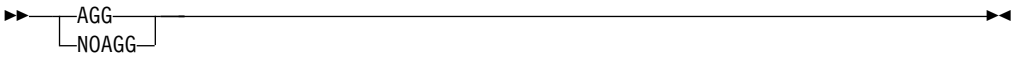
The value of the AGGRCOPY option for a partition is set to the value of the first subprogram that is placed in the partition. During IPA inlining, subprograms with different AGGRCOPY settings may be combined in the same partition. When this occurs, the resulting partition is always set to AGGRCOPY(OVERLAP).

AGGREGATE | NOAGGREGATE

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓		✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOAGGREGATE	NOAGGREGATE -V sets AGGREGATE	NOAGGREGATE -V sets AGGREGATE				

CATEGORY: Listing



The AGGREGATE option instructs the compiler to include 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.

You can specify this option using the #pragma option directive for C.

In the z/OS UNIX System Services environment, this option is turned on by specifying -V when using the c89 or cc commands.

Effect on IPA Compile Step
The AGGREGATE option has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step
The IPA Link step accepts the AGGREGATE option, but ignores it.

ALIAS | NOALIAS

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓		✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOALIAS	NOALIAS	NOALIAS				

CATEGORY: Object Code Control

The ALIAS option generates ALIAS control statements that help the binder locate modules in a load library. The suboption *name* is assigned to the NAME control statement.

ALIAS*(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.

ALIAS If you specify ALIAS with no suboption, the compiler selects an existing CSECT name from the program, and nominates it on the NAME statement.

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

NOALIAS If you specify NOALIAS, the compiler does not generate a NAME control statement. NOALIAS has the same effect as ALIAS().

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

For complete details on ALIAS and NAME control statements, see *z/OS DFSMS Program Management*.

You can specify this option using the #pragma option directive for C.

Effect on IPA Compile Step

If you specify the ALIAS option on the IPA Compile step, the ALIAS option is ignored.

Effect on IPA Link Step

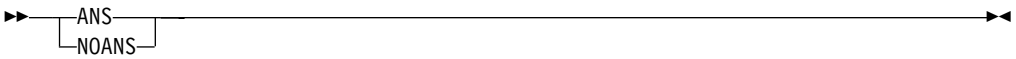
If you specify the ALIAS option on the IPA Link step, the IPA Link step generates an unrecoverable error.

ANSIALIAS | NOANSIALIAS

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
ANSIALIAS	ANSIALIAS	NOANSIALIAS	ANSIALIAS			

CATEGORY: Object Code Control



The ANSIALIAS option 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 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.

IBM C and 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 *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.
- Positively affects the long-term maintainability of a program by enforcing ISO-compliant code at an earlier phase of the development process.

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.

If you specify `NOANSIALIAS`, 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.

The `CHECKOUT(CAST)` compiler option can help you locate some but not all suspicious casts and `ANSIALIAS` violations.

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.

```
1  extern int y = 7.;
2
3  void main() {
4       float x;
5       int i;
```

```
6       x = y;
7       i = *(int *) &x;
8       printf("i=%d. x=%f.\n", i, x);
9  }
```

In the example above, 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 it in the ISO standard, so the program violates the standard.

Using ANSIALIAS (the default) is a *guarantee* that the program 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 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 believes this guarantee. 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:

```
1  extern int y = 7.;
2
3  void main() {
4       float x;
5       int i;
6       int temp;
7       temp = *(int *) &x;  /* uninitialized */
8       x = y;
9       i = temp;
10      printf("i=%d. x=%f.\n", i, x);
9  }
```

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 compiler so correctness is more important.

Notes:

1. This option only takes effect if the OPTIMIZE option is in effect.
2. 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 139 for more information on LANGLVL(COMMONC).
3. A comment that indicates the ANSIALIAS option setting is generated in your object module to aid you in diagnosing your program.
4. 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.

See “CHECKOUT | NOCHECKOUT” on page 92 to see how to obtain more diagnostic information.

You can specify this option using the #pragma option directive for C.

Effect on IPA Compile Step

The ANSIALIAS option has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

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

ARCHITECTURE

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
TARGET(OSV2R10 and above): ARCH(2) TARGET(OSV2R9 and below): ARCH(0)						

CATEGORY: Object Code Control

The ARCH option 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. A subparameter specifies the group to which a model number belongs. Note that 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.

Use the ARCH option in conjunction with the TUNE option. For more information on the interaction between ARCH and TUNE, see “TUNE” on page 213.

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

Current groups of models that are supported include the following:

0 Is the default value. It 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 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 OS/390 V2R6 or higher operating system, and simulated by software. There will be a significant performance degradation for the simulation.

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 above system machine models, x indicates any value. For example, 9672-Rx4 means 9672-RA4 through to 9672-RX4, not just 9672-RX4.

You can specify this option using the #pragma option directive for C.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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 213.

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.

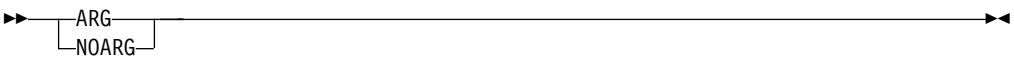
The Compiler Options Map section of the IPA Link step listing displays the option value that you specified for each IPA object file during the IPA Compile step.

ARGPARSE | NOARGPARSE

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
ARGPARSE	ARGPARSE	ARGPARSE	ARGPARSE			

CATEGORY: Program Execution



The ARGPARSE option specifies that the arguments supplied 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.

If you specify NOARGPARSE, arguments on the invocation line are not parsed. argc has a value of 2, and argv contains a pointer to the string.

Note: 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 .

This option has no effect under CICS.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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.

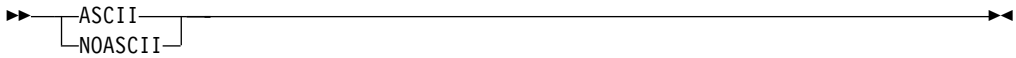
If you specified ARGPARSE on the IPA Compile step, you do not need to specify it again on the IPA Link step to affect that step. The IPA Link step uses the information generated for the compilation unit that contains the main() function. If it cannot find a compilation unit that contains main(), it uses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

ASCII | NOASCII

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOASCII						

CATEGORY: Object Code Control and Program Execution



The ASCII option instructs the compiler to perform the following:

- Use XPLink linkage unless explicitly overwritten by the NOXPLINK option. Note that the ASCII run-time functions require XPLINK. The system headers checks the __XPLINK__ macro (which is predefined when the XPLINK option is turned on). The prototypes for the ASCII runtime functions will not be exposed under NOXPLINK. Specifying the NOXPLINK option explicitly will prevent you from using the ASCII run-time functions. ASCII NOXPLINK will be accepted, and will generate an error (CCN8136) if there is a ″main″ in the code (an executable to be generated).
- Use ISO8895-1 for its default codepage rather than IBM-1047 for character constants and string literals.
- Set a flag in the program control block to indicate that the compile unit is ASCII.
- Pre-define the macro__CHARSET_LIB for use in header files.

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

Effect on IPA Compile Step

The ASCII option has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

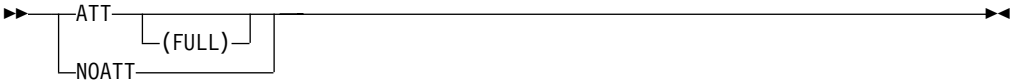
The IPA Link step accepts the ASCII option, but ignores it.

ATTRIBUTE | NOATTRIBUTE

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
	✓	✓		✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOATTRIBUTE			NOATTRIBUTE	NOATTRIBUTE	NOATTRIBUTE	NOATTRIBUTE

CATEGORY: Listing



The ATTRIBUTE option produces a Cross Reference listing that shows the attributes for each symbol, an External Symbol Cross Reference section, and Static Map section.

The ATTRIBUTE(FULL) option produces a listing of all identifiers that are found in your code, even those that are not referenced. The compiler writes the listing produced by ATTRIBUTE or ATTRIBUTE(FULL) to a listing file.

The NOATTRIBUTE option suppresses the attribute listing.

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

Effect on IPA Compile Step

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 option (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.

Effect on IPA Link Step

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.

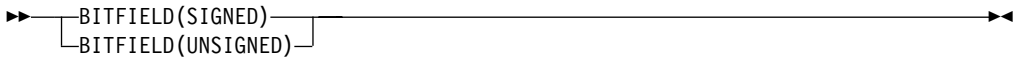
The IPA Link step creates a Storage Offset listing section if during the IPA Compile step you requested the additional symbol storage offset information for your IPA objects.

BITFIELD(SIGNED) | BITFIELD(UNSIGNED)

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
BITFIELD(UNSIGNED)						

CATEGORY: Programming Language Characteristics Control



The BITFIELD compiler option specifies the default sign for bitfields.

Effect on IPA Compile Step

The BITFIELD option has the same effect on IPA Compile step processing as it does on a regular compilation.

Effect on IPA Link Step

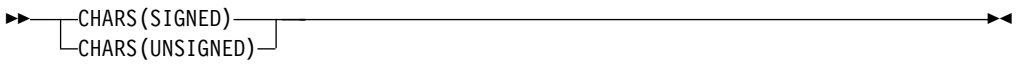
The IPA Link step accepts the BITFIELD option, but ignores it.

CHARS(SIGNED) | CHARS(UNSIGNED)

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities Note that this changes if the -v flag is set.					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
CHARS(UNSIGNED)						

CATEGORY: Programming Language Characteristics Control



The CHARS compiler option instructs the compiler to treat all variables of type char as either signed or unsigned. This option has the same effect as the #pragma chars directive, which takes precedence over the compiler option. See the *C/C++ Language Reference* for more information on this directive.

Effect on IPA Compile Step

The CHARS option has the same effect on IPA Compile step processing as it does on a regular compilation.

Effect on IPA Link Step

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

CHECKOUT | NOCHECKOUT

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓		✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities Note that this changes if the -v flag is set.					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOCHECKOUT	NOCHECKOUT	NOCHECKOUT				

CATEGORY: Debug/Diagnostic

where:

subopts is one of the suboptions that are shown in Table 16 on page 93.

The CHECKOUT option instructs the compiler to produce informational messages that can indicate possible programming errors. The messages can help z/OS C programmers to debug their programs.

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.

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

Note: Default CHECKOUT suboptions are underlined.

Table 16. CHECKOUT Suboptions, Abbreviations, and Descriptions

CHECKOUT Suboption	Abbreviated Name	Description
ACCURACY | NOACCURACY	AC | NOAC	Assignments of long values to variables that are not long
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 83 for more information on ANSI type-based aliasing. Also see "DLL | NODLL" on page 106 for DLL function pointer casting restrictions.
ENUMSIZE	ENUM	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	Nonportable usage of the z/OS 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

You can specify the CHECKOUT option on the invocation line and on the #pragma options preprocessor directive. 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)
```

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).

The NOCHECKOUT option specifies that the compiler should not generate informational error messages. 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 the *C/C++ Language Reference* for more information regarding this pragma directive.

You can specify this option using the #pragma option directive for C.

In the z/OS UNIX System Services environment, this option is turned on by specifying -V when using the c89 or cc commands. The suboptions, when -V is specified, become ALL, NOEXTERN, NOPPCHECK, NOPPTRACE.

Note: See the "INFO | NOINFO" on page 125 compiler option section for information on C++ support for similar functionality.

Effect on IPA Compile Step

The CHECKOUT option is used for source code analysis, and has the same effect on IPA Compile step processing as it does on a regular compilation.

Effect on IPA Link Step

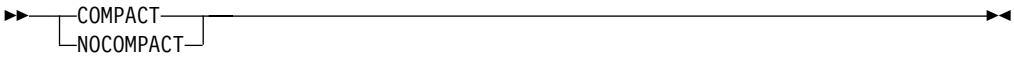
The IPA Link step accepts the CHECKOUT option, but ignores it.

COMPACT | NOCOMPACT

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOCOMPACT						

CATEGORY: Object Code Control



During optimizations performed during 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. When the COMPACT option is used, the compiler favours those optimizations which tend to limit the growth of the code. 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 since subprograms may not be inlined or inline other subprograms 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 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 the user changes the application code.

You can specify this option for a specific subprogram using the #pragma option_override(subprogram_name, "OPT(COMPACT)") directive.

Effect on IPA(OBJONLY) Compilation

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.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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.

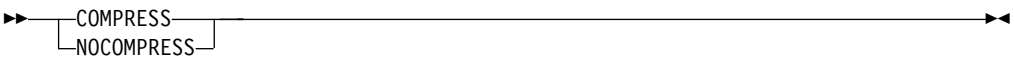
The Partition Map sections of the IPA Link step listing and the object module END information section display the value of the COMPACT option. The Partition Map also displays any subprogram-specific COMPACT values.

COMPRESS | NOCOMPRESS

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOCOMPRESS						

CATEGORY: Object Code Control



Use the COMPRESS option to suppress the generation of function names in the function control block thereby reducing the size of your application's load module. Note that the 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 the Debug Tool and Performance Analyzer. Without these function names, the reports generated by these services and tools may not be complete.

Note that if COMPRESS and TEST are in effect at the same time, the compiler issues a warning message and ignores the COMPRESS option.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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.

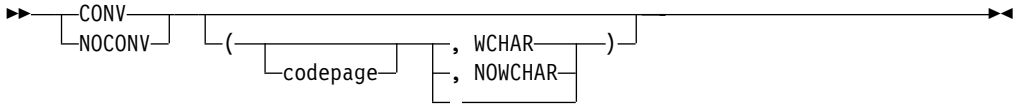
The Partition Map sections of the IPA Link step listing and the object module END information section display the value of the COMPRESS option.

CONVLIT | NOCONVLIT

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOCONVLIT (, WCHAR)						

CATEGORY: Preprocessor



The CONVLIT option changes the assumed codepage for character and string literals within the compilation unit. You can use an optional suboption to specify the codepage that you want to use for string literals. If you specify NOCONV or CONV without a suboption, the default codepage, or the codepage specified by the LOCALE option is used.

You can also specify a suboption with the NOCONV option. The result of the following specifications is the same:

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

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 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 the *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"

The CONVLIT(**,** WCHAR) suboption 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. The default is WCHAR. 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.

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.

Note: 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 uses the option regardless of the order in which you specify them.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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

CSECT | NOCSECT

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
For NOGOFF, NOCSECT						
For GOFF, CSECT()						

CATEGORY: Object Code Control

The CSECT option ensures that the code, static data, and test sections of your object module are named. Use this option, or the #pragma CSECT directive, if you will be using SMP/E to service your product, and to aid in debugging your program. See the *C/C++ Language Reference* for further information on the #pragma CSECT directive.

The NOCSECT option does not name the code, static, or test data sections of your object module.

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

For NOGOFF, if the LONGNAME compiler option is not in effect when you specify CSECT(qualifier), the compiler turns it on, and issues an informational message. For GOFF, both 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.

The CSECT option with no 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 an HFS file.
- For NOGOFF, 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.

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 eight characters long, the rightmost character is dropped and the compiler issues an informational message except in the case of GOFF. The test CSECT name is always truncated to eight 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

The CSECT option with the qualifier suboption

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 an HFS file

suffix Is one of the following:

C For code CSECT

S For static CSECT

T For test CSECT

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. If the compiler option LONGNAME is not in effect when you specify the CSECT(qualifier), the compiler turns it on, and issues an informational message.

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

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 z/OS 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.

Effect on IPA Compile Step

The `CSECT` option has the same effect on the IPA Compile step (if you specify the `OBJECT` suboption of the `IPA` option) as it does on a regular compilation.

Effect on IPA Link Step

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. See “IPA Link Step Control File” on page 350 for details.
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.

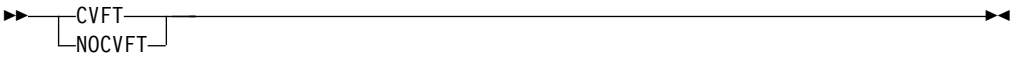
The IPA Link step ignores the information that is generated by `#pragma csect` on the IPA Compile step.

CVFT | NOCVFT

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
	✓			

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
CVFT						

CATEGORY: Object Code Control



The NOCVFT option benefits your application's performance by shrinking the size of the writeable static area (WSA). It reduces the size of construction virtual function tables (CVFT), which in turn reduces the load module size. Use NOCVFT if none of the constructors in your application calls virtual functions from within the class hierarchy, either directly or indirectly.

The NOCVFT option relieves certain constructors from tracking which virtual function to call at different stages of the construction process. This tracking by the constructor would require that the constructor maintain its own CVFT. Only constructors that call virtual functions within a class hierarchy that uses virtual inheritance are affected.

Consider the following example:

```
struct  A {
            virtual int f() { return 0; }    // line a
};

struct B : virtual A {
            virtual int f() { return 1; }    // line b
            B() { cout << f() << endl; }
};

struct C : virtual B {
            virtual int f() { return 2; }    // line c
};

...
```

In the above example, if an instance of C is constructed, the ISO C++ standard requires that 1 (the number one) be printed. That is, the function B::f() at line b should be called during the construction of C. After C is constructed, a call to f() should invoke C::f() at line c. To support the ANSI behavior and call the right function, the z/OS C++ compiler needs to keep extra information during object construction. This extra information can require a lot of memory if an application uses a lot of virtual inheritance.

The NOCVFT option breaks the above ANSI C++ behavior. In that case, the virtual function called by the application is always the same one that would be called if the object is fully constructed. In the above example, this is C::f(), and 2 is printed during the construction of an instance of C (the function at line c). The CVFT option preserves the ANSI C++ behavior.

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

Effect on IPA Compile Step

The CVFT option has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

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

DEFINE

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities Note you can override with appropriate -D or -U options.					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
no default user definitions	DEFINE(errno=(*__errno())) c89 also specifies DEFINE(_OPEN_DEFAULT=1)	DEFINE(errno=(*__errno()+)) cc also specifies DEFINE(_OPEN_DEFAULT=0) and DEFINE(_NO_PROTO=1)	DEFINE(errno=(*__errno())) c++ also specifies DEFINE(_OPEN_DEFAULT=1)			

CATEGORY: Preprocessor

The DEFINE option defines preprocessor macros that take effect before the compiler processes the file. You can use this option more than once.

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*.

If the suboptions that you specify contain special characters, see “Using Special Characters” on page 64 for information on how to escape special characters.

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))
```

In the z/OS UNIX System Services environment, variables can be set by specifying `-D` when using the `c89`, `cc`, or `c++` commands.

Effect on IPA Compile Step
The `DEFINE` option is used for source code analysis, and has the same effect on an IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step
The IPA Link step accepts but ignores the `DEFINE` option.

DIGRAPH | NODIGRAPH

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
DIGRAPH			DIGRAPH			

CATEGORY: Preprocessor

The `DIGRAPH` option allows you to use additional digraphs in both C and C++ applications. In addition, it allows you to use additional keywords in C++ applications only. A digraph is a combination of keys that produces a character that is not available on some keyboards. Table 17 on page 105 shows the digraphs that z/OS C/C++ supports:

Table 17. Digraphs

Key Combination	Character Produced
<%	{
%>	}
<:	[
:>	]
%:	#
%%[1]	#
%:%:	##
%%%%[1]	##

Table 18 shows additional keywords that z/OS C++ supports:

Table 18. Additional Keywords

Keyword	Characters produced
bitand	&
and	&&
bitor	|
or	||
xor	^
compl	~
and_eq	&=
or_eq	|=
xor_eq	^=
not	!
not_eq	!=

Note: Digraphs are not replaced in string literals, comments, or character literals. For example:

```
char * s = "<%%>";   // stays "<%%>"

switch (c) {
  case '<%' : ...    // stays '<%'
  case '%>' : ...    // stays '%>'
}
```

See the *C/C++ Language Reference* for more information on digraphs.

Effect on IPA Compile Step

The `DIGRAPH` option has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

The IPA Link step issues a diagnostic message if you specify the `DIGRAPH` option on that step.

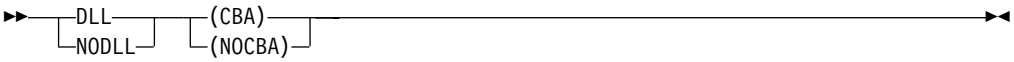
1. The digraphs %% and %%%% are not digraphs in the C Standard. For compatibility with z/OS C++, however, they are supported by z/OS C. Use the %: and %:%: digraphs instead of %% and %%%% whenever possible.

DLL | NODLL

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✔	✔	✔	✔	✔

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NODLL(NOCBA) for C compile and IPA Link step. DLL(NOCBA) for C++ Compile.						

CATEGORY: Object Code Control



The DLL option instructs the compiler to produce DLL code. The DLL code can export or import functions and external variables.

The DLL option has two suboptions:

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 non-XPLINK 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.

Note: You should write your code according to the rules listed in the chapter ″Building Complex DLLs″ in the *z/OS 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 the *z/OS C/C++ Programming Guide*.

Notes:

1. Code compiled with the z/OS C++ compiler, and code compiled with the XPLINK compiler option, is always DLL code. You can not specify NODLL for these cases.
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 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 without the XPLINK 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 */
         }
}
```

   The above 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” on page 92 for more information on obtaining diagnostic information for C applications.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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.

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.

ENUMSIZE

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
ENUM(SMALL)						

CATEGORY: Object Code Control

The ENUMSIZE option specifies the amount of storage occupied by enumerations. It enables a user to select the type used to represent all enums defined in a compilation unit. ENUMSIZE has the following suboptions:

SMALL

Specifies that enumerations occupy a minimum amount of storage, which is either 1, 2, or 4 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.

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.

The default is `ENUM(SMALL)`. It allocates the amount of storage that is required by the smallest prefedined type, which can represent that range of enum constants, to an enum variable. The other suboptions allocate a specific amount of storage to an enum variable.

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:

```
#pragma enum(1)
enum e_tag {
    a = 0,
     b = SHRT_MAX /* error CCN3387 for C, CCN5525 for C++ */
 } e_var;
 #pragma enum(reset)
```

The following tables illustrate the preferred sign and type for each range of enum constants:

Table 19. ENUM Constants for C and C++

ENUM Constants	small	1	2	4	int
0..127	unsigned char	signed char	short	int	int
-128..127	signed char	signed char	short	int	int
0..255	unsigned char	unsigned char	short	int	int
0..32767	unsigned short	ERROR	short	int	int
-32768..32767	short	ERROR	short	int	int
0..65535	unsigned short	ERROR	unsigned short	int	int
0..2147483647	unsigned int	ERROR	ERROR	int	int
-(2147483647+1)..2147483647	int	ERROR	ERROR	int	int
0..4294967295	unsigned int	ERROR	ERROR	unsigned int	unsigned int

You can use `#pragma enum` to change the `ENUM` option value used for individual enum declaration in a source file. Refer to the *C/C++ Language Reference* for more infomation regarding this pragma directive.

Effect on IPA Compile Step

The `ENUMSIZE` option has the same effect on the IPA Compile step that it does on a regular compilation.

Effect on IPA Link Step

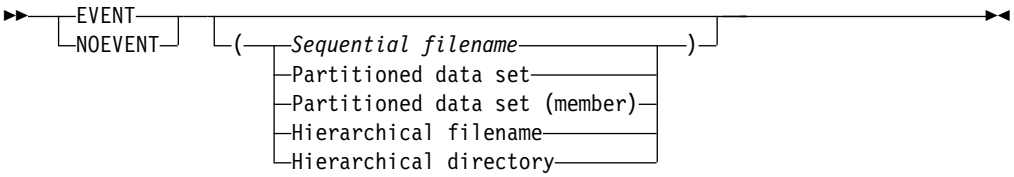
The IPA Link step accepts the `ENUMSIZE` option, but ignores it.

EVENTS | NOEVENTS

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOEVENTS	NOEVENTS	NOEVENTS				

CATEGORY: Debug/Diagnostic



The EVENTS option creates an events file that contains error information and source file statistics. The compiler writes the events data to the DD:SYSEVENT ddname, if you allocated one before you called the compiler. Otherwise, it allocates a data set, and the name is the file name with SYSEVENT as the lowest-level qualifier.

If you specified a suboption, the compiler uses the data set that you specified, and ignores the DD:SYSEVENT.

If the source file is an HFS file, and you do not specify the events file name as a suboption, the compiler writes the events file in the current working directory. The events 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 G. Layout of the Events File” on page 615.

Effect on IPA Compile Step

The EVENTS option has the same effect on the IPA Compile step that it does on a regular compilation.

Effect on IPA Link Step

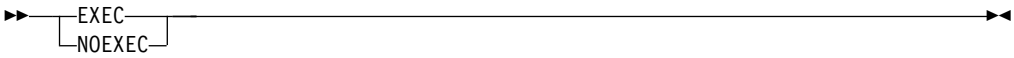
The IPA Link step accepts the EVENT option, but ignores it.

EXECOPS | NOEXECOPS

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
EXECOPS	EXECOPS	EXECOPS	EXECOPS			

CATEGORY: Program Execution



The EXECOPS option allows you to control whether run-time options will be recognized at run time without changing your source code. It is equivalent to including a #pragma runopts (EXECOPS) directive in your source code.

If this option is specified on both the command line and in a #pragma runopts directive, the option on the command line takes precedence.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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.

EXH | NOEXH

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
	✓			

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
EXH						

CATEGORY: Object Code Control

The EXH option controls the generation of C++ exception handling code.

The NOEXH option suppresses the generation of the exception handling code, which results in code that runs faster, but will not be ANSI-compliant if the program uses exception handling.

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 a source file has try/catch blocks or throws objects, you cannot compile it with the NOEXH option.

Effect on IPA Compile Step

The EXH option has the same effect on the IPA Compile step that it does on a regular compilation.

Effect on IPA Link Step

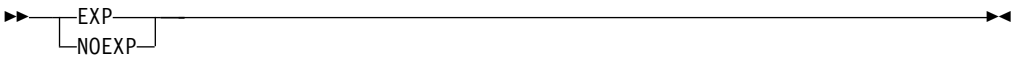
The IPA Link step issues a diagnostic message if you specify the EXH option for that step.

EXPMAC | NOEXPMAC

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOEXPMAC	NOEXPMAC	NOEXPMAC	NOEXPMAC			

CATEGORY: Listing



The EXPMAC option instructs the compiler to show all expanded macros in the source listing. 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.

In the z/OS UNIX System Services environment, this option is turned on by specifying -V when using the c89, cc or c++ commands.

Effect on IPA Compile Step

The EXPMAC option has the same effect on the IPA Compile step that it does on a regular compilation.

Effect on IPA Link Step

The IPA Link Step accepts the EXPMAC option, but ignores it.

EXPORTALL | NOEXPORTALL

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOEXPORTALL	NOEXPORTALL	NOEXPORTALL	NOEXPORTALL			

CATEGORY: Object Code Control

The EXPORTALL option instructs the compiler to export all external functions and variables in the compilation unit so that a DLL application can use them. Use this option if you are creating a DLL and want to export **all** externally defined functions and variables. You may not export the main() function.

Notes:

1. If you only want to export some of the externally defined functions and variables, use #pragma export, or the _Export keyword for C++. For more information see the *C/C++ Language Reference*.
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 C compiler turns them on.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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

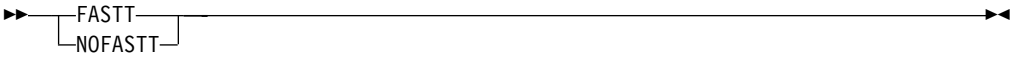
If you use the EXPORTALL option during the IPA Compile step, you severely limit IPA optimization. Refer to "DLL | NODLL" on page 106 for more information about the effects of this option on IPA processing.

FASTTEMPINC | NOFASTTEMPINC

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
	✓			

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOFASTT						

CATEGORY: File Management



The FASTTEMPINC option may improve template instantiation compilation time when large numbers of recursive templates are used in an application.

The FASTTEMPINC option defers generating object code until the final version of all template definitions have been determined. Then, a single compilation pass is made to generate the final object code. This means that time is not wasted on generating object code that will be discarded and generated again.

When NOFASTT is used, 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.

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.

Effect on IPA Compile Step

The FASTT option only affects the processing of source. It has no effect on code generation; therefore, it has the same effect on IPA Compile as it does on a regular compilation.

Effect on IPA Link Step

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

FLAG | NOFLAG

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities Note that FLAG(I) is used if the -V flag is set.					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
FLAG(I)	FLAG(W)	FLAG(W)	FLAG(W)	FLAG(W)	FLAG(W)	FLAG(W)

CATEGORY: Debug/Diagnostic

The FLAG option specifies the minimum severity level for which you want notification. You specify the minimum severity level by using the compiler option FLAG (*severity*), where *severity* is one of the following:

I An informational message that is generated by the compiler. This is the default.

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.

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(S) option.

Effect on IPA Compile Step

The FLAG option has the same effect on the IPA Compile step that it does on a regular compilation.

Effect on IPA Link Step

The IPA Link step uses the FLAG value that you specify for that step.

FLOAT

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✔	✔	✔	✔	✔

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
FLOAT(HEX, FOLD, NOMAF, NORRM, NOAFP or AFP). For ARCH(2) the default is NOAFP. For ARCH(3) or higher, the default is AFP.						

CATEGORY: Object Code Control

The FLOAT option selects the format of floating-point numbers; the format can be either base 2 IEEE-754 binary format, or base 16 S/390 hexadecimal format. In the description below, the IEEE-754 binary format is referred to as the binary floating-point format, and the S/390 hexadecimal format as the hexadecimal floating-point format. FLOAT has the following suboptions:

HEX | IEEE

DEFAULT: HEX

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 formatted floating-point numbers and instructions. The unabbreviated form of this suboption is HEXADECIMAL. In previous releases of z/OS C/C++ and OS/390 C/C++, the floating-point format was always hexadecimal.

FOLD | NOFOLD

DEFAULT: FOLD

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

DEFAULT:

- NOMAF
- If NOSTRICT and FLOAT(IEEE) are specified, MAF is the default.

Uses floating-point Multiply and Add, and Multiply and Subtract instructions where possible, instead of the separate Multiply Float, Add Float, or Multiply Float, Subtract Float instruction pairs. On the current generation of machines, the multiply-and-add and multiply-and-subtract operations are much slower than the corresponding multiply and add/subtract instructions. They should be used only when improved precision of intermediate results is desired.

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

DEFAULT: NORRM

RRM (run-time rounding mode) tells the compiler that the run-time rounding mode may not be the default, *round-to-nearest*, and prevents compiler optimizations that rely on *round-to-nearest* rounding mode. Use this option if your program changes the rounding mode by any means. Otherwise, the program may compute incorrect results.

RRM is not available for hexadecimal floating-point mode.

AFP | NOAFP

DEFAULT:

- If the level of the ARCH option is lower than 3, the default is NOAFP
- If the level of the ARCH option is 3 or higher, the default is AFP

AFP instructs the compiler to generate code which makes full 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, and 7 through 15.

The AFP registers are physically available only on the newer S/390 machine models, starting with the processors that are represented by the ARCH(3) setting. However, when the application executes under OS/390 Version 2 Release 6 and higher releases on a processor that does not have the AFP registers, the operating system is able to intercept the use of an AFP register and emulate the operation such that the AFP register appears to be available to the application.

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.

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 S/390 machine models.

Using IEEE Floating-Point

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 *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 run-time 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 S/390 hexadecimal floating-point format), and have no need to communicate these 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.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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.

GOFF | NOGOFF

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOGOFF						

CATEGORY: Object Code Control

The GOFF option instructs the compiler to produce an object file in the Generalized Object File Format (GOFF). The GOFF format supersedes the 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 CSECT 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.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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.

GONUMBER | NOGONUMBER

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOGONUMBER	NOGONUMBER	NOGONUMBER	NOGONUMBER	NOGONUMBER	NOGONUMBER	NOGONUMBER

CATEGORY: Debug/Diagnostic

The GONUMBER option generates line number tables that correspond to the input source file. These tables are for use by the Debug Tool and for error trace back information when an exception occurs.

The compiler turns on this option when you use the TEST option.

Note: 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.

You can specify this option using the #pragma option directive for C.

Effect on IPA Compile Step

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. The GONUMBER and LIST options use this information during the IPA Link step.

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.

In the z/OS UNIX System Services environment, this option is turned on by specifying -g when using the c89, cc or c++ commands.

Effect on IPA Link Step

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:

- If you specified the GONUMBER, LIST, IPA(GONUMBER), or IPA(LIST) option on the IPA Compile step, the GONUMBER tables contain complete 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.

Refer to “Interactions between Compiler Options and IPA Suboptions” on page 63 and “LIST | NOLIST” on page 150 for more information.

HALT(num)

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
HALT(16)	HALT(16)	HALT(16)	HALT(16)	HALT(16)	HALT(16)	HALT(16)

CATEGORY: Input Source File Processing Control

The HALT option stops compilation, depending on the return code from the compiler. This option applies to the compilation of all members of a PDS or an HFS directory. 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.

Valid codes for *num* correspond to return codes from the compiler. See *z/OS C/C++ Messages* for a list of return codes.

Effect on IPA Compile Step

The HALT option has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

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.

HALTONMSG | NOHALTONMSG

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOHALTON						

CATEGORY: Input Source File Processing Control

The HALTONMSG option instructs the C/C++ front end to stop after the compilation phase when it encounters the specified msg_number. When the compilation stops as a result of the HALTONMSG option, the compiler return code is nonzero.

Note: You can specify more than one message number by separating the message numbers with commas.

Effect on IPA Compile Step

The HALTONMSG option has the same effect on IPA Compile step processing as it does on a regular compilation.

Effect on IPA Link Step

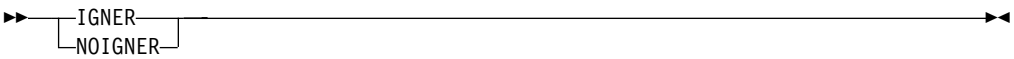
The IPA Link step accepts the HALTONMSG option but ignores it.

IGNERRNO | NOIGNERRNO

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOIGNERRNO						

CATEGORY: Object Code Control



The IGNERRNO option informs the compiler that your application is not using errno. Specifying this option allows the compiler to explore additional optimization opportunities for library functions in LIBANSI.

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.

You can specify this option using the `#pragma option` directive for C.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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`.

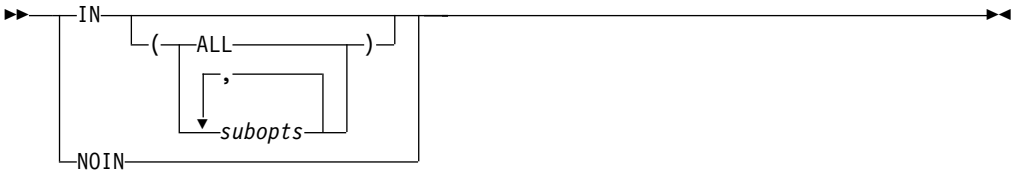
The value of the `IGNERRNO` option for a partition is set to the value of the first subprogram that is placed in the partition. The Partition Map sections of the IPA Link step listing and the object module display the value of the `IGNERRNO` option.

INFO | NOINFO

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
	✓			

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
`C++ only: INFO(LAN)`			`INFO(LAN)`			

CATEGORY: Debug/Diagnostic



The `INFO` option instructs the compiler to generate warning messages. Use *subopts* if you want to specify the type of warning messages.

If you specify INFO with no suboptions, it is the same as specifying INFO(ALL). The following is a list of the *subopts*:

CLS Emits class informational warning messages.

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.

EFF Emits information about statements with no effect.

ENU Emits information about ENUM checks.

GNR Emits information about the generation of temporary variables.

GEN Emits message if compiler generates temporaries.

LAN Emits language level checks.

PAR Emits warning messages on unused parameters.

POR Emits warnings about nonportable 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.

USE Emits information about usage of variables.

VFT Indicates where vftable is generated.

ALL Emits all of the above

no suboptions
Same result as INFO(ALL).

In the z/OS UNIX System Services environment, this option is turned on by specifying -V when using the c++ command. The suboption is ALL when -V is specified.

Note: Refer to “CHECKOUT | NOCHECKOUT” on page 92 for similar functionality in C.

Effect on IPA Compile Step

The INFO option has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

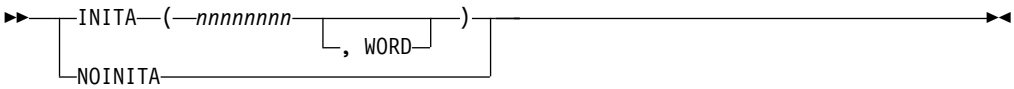
The IPA Link step issues a diagnostic message if you specify the INFO option.

INITAUTO | NOINITAUTO

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOINITAUTO						

CATEGORY: Object Code Control



The `INITAUTO` option tells the compiler to generate code to initialize automatic variables. Automatic variables require storage only while the block in which they are declared is active. See the *C/C++ Language Reference* for more information on automatic variables.

Automatic variables without initializers are not implicitly initialized. The `INITAUTO` option instructs the compiler to generate code to initialize these variables with a user-defined value.

In the above syntax, 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.

The suboption `Word` is optional, and can be abbreviated to `W`. If you specify `Word`, *nnnnnnnn* 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.

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.

Note: The word initializer is useful in checking uninitialized pointers.

Since extra code is generated, this option can reduce the run-time performance of the program.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

You can specify the INITAUTO option for an IPA link step, and it will override the setting in the compile step.

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.

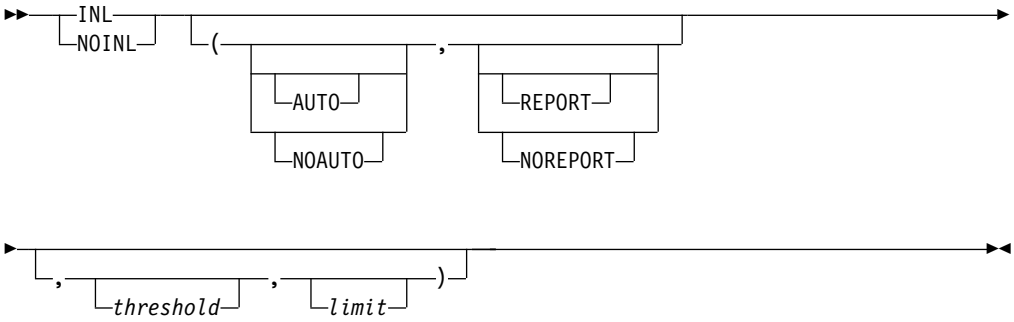
The Partition Map sections of the IPA Link step listing and the object module display the value of the INITAUTO option.

INLINE | NOINLINE

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	**Set by z/OS UNIX System Services Utilities** **Note that these values change if the -O flag is set. The default of NOREPORT is changed by -V to REPORT (and NOINLRPT to INLRPT).**					
	Regular Compile			**IPA Link**		
	c89	**cc**	**c++**	**c89**	**cc**	**c++**
C/C++ Compile: If NOOPT is in effect: NOINLINE (AUTO,NOREPORT, 100,1000) NOOPT is the default for C/C++ compile If OPT is in effect: INLINE (AUTO,NOREPORT, 100,1000) *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 NOOPT: NOINLINE (AUTO, NOREPORT, 100, 1000) For OPT: INLINE (AUTO, NOREPORT, 100, 1000)	For NOOPT: NOINLINE (AUTO, NOREPORT, 100, 1000) For OPT: INLINE (AUTO, NOREPORT, 100, 1000)		NOINLINE (AUTO, NOREPORT, ,)	NOINLINE (AUTO, NOREPORT, ,)	NOINLINE (AUTO, NOREPORT, ,)

CATEGORY: Object Code Control and Listing



The INLINE option instructs the compiler to place 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. It 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 C/C++ Programming Guide*.

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 the *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 272, “Inline Report” on page 254, and “Inline Report for IPA Inliner” on page 283.

  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 C compiler translates your z/OS 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`.

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 options. If an option on the invocation line conflicts with an option in the #pragma options directive, the one on the invocation line takes precedence.

For example, because you typically do not want to inline your subprograms when you are developing a program, you can specify the NOINLINE option on a #pragma options preprocessor directive. When you want to inline your subprograms, you can override the NOINLINE option 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.

You can specify this option using the #pragma option directive for C.

In the z/OS UNIX System Services environment, the INLINE(,REPORT,,) option is turned on by specifying -V when using the c89 or cc commands.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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.

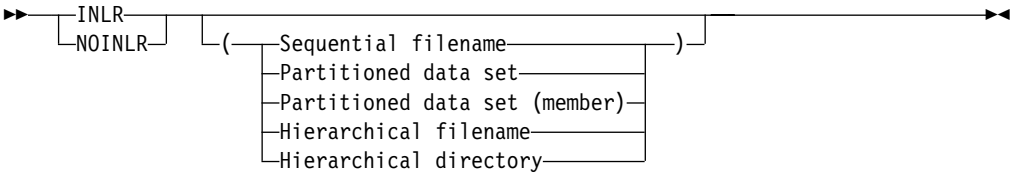
The IPA inliner has the inlining capabilities of the compilation unit inliner. In addition, the IPA inliner detects complex recursion, and may inline it. If you specify the `INLRPT` option, the IPA Link listing contains the IPA Inline Report section. This section is similar to the report that the compilation unit inliner generates. If you specify `NOINLINE(,REPORT,,)` or `NOINLINE INLRPT`, IPA generates an IPA Inline Report section that specifies that nothing was inlined.

INLRPT | NOINLRPT

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOINLRPT			NOINLRPT (/dev/fd1)			

CATEGORY: Listing



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. This report 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.

You can specify *filename* for the inline report output file. If you do not specify *filename*, 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 *filename* suboption. This *filename* then becomes the default. If you subsequently use the INLR option without *filename*, the compiler uses the *filename* 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 *filename* with any of the SOURCE, LIST, or INLRPT options, all the listing sections are combined into the last *filename* 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)`

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

Effect on IPA Compile Step

The INLRPT option has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

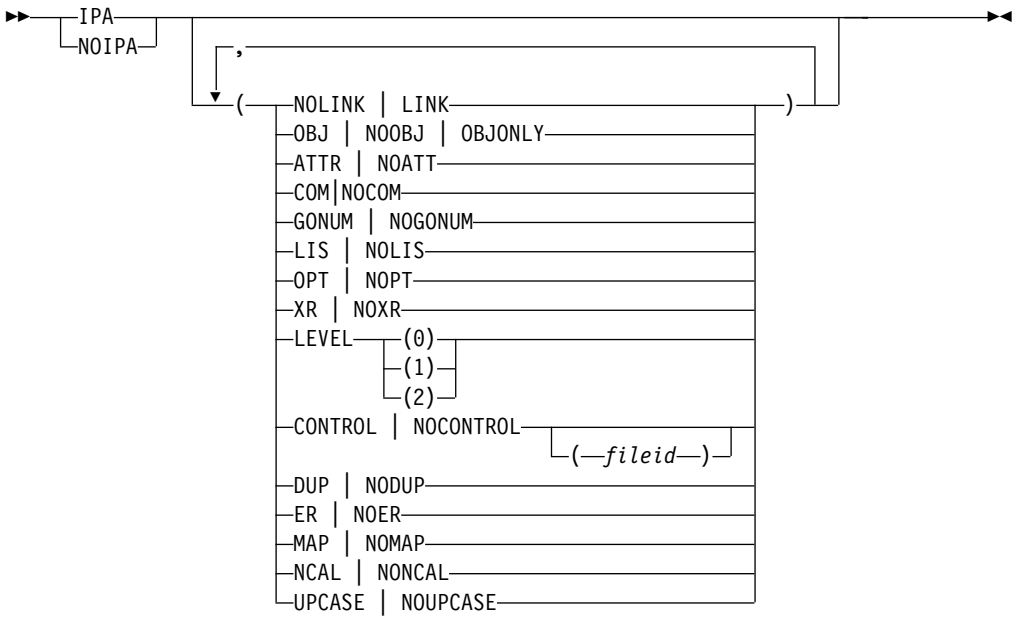
If you specify the INLRPT option on the IPA Link step, the IPA Link step listing contains an IPA Inline Report section. Refer to “INLINE | NOINLINE” on page 128 for more information about generating an IPA Inline Report section.

IPA | NOIPA

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOIPA	NOIPA	NOIPA	NOIPA	NOIPA(NOCONTROL (ipa.ctl), DUP,NOER, NOMAP, NOUPCASE, NONCAL) IPA(LINK, LEVEL(1))	NOIPA(NOCONTROL (ipa.ctl), DUP,NOER, NOMAP, NOUPCASE, NONCAL) IPA(LINK, LEVEL(1))	NOIPA(NOCONTROL (ipa.ctl), DUP,NOER, NOMAP, NOUPCASE, NONCAL) IPA(LINK, LEVEL(1))

CATEGORY: Object Code Control, IPA Link Control, IPA Object Control, File Management, Listing and Debug/Diagnostic



The IPA option instructs the compiler to perform Interprocedural Analysis across compilation units.

The NOIPA option instructs the compiler to perform a regular compilation.

IPA Compile Step Suboptions

IPA(NOLINK) invokes the IPA Compile step. NOLINK is the default suboption of the IPA option. Only the following IPA suboptions affect the IPA Compile step. You can specify other IPA suboptions, but they do not affect the IPA Compile 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 121 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 63.

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 150 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 63.

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 the *z/OS 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 220 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 Step Suboptions

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).

  For more information about the IPA control file directives, refer to "IPA Link Step Control File" on page 350.

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
  - Source File Map
  - Compiler Options Map
  - Global Symbols Map
  - Partition Map for each partition

  The default is IPA(NOMAP).

  See “Using the IPA Link Step Listing” on page 274 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 assembler 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).

Refer to the Interprocedural Analysis chapter in the *z/OS C/C++ Programming Guide* for an overview and more details about Interprocedural Analysis.

KEYWORD | NOKEYWORD

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
Recognizes all C++ keywords						

CATEGORY: Programming Language Characteristics Control

The KEYWORD option controls whether the specified name is created as a keyword or an identifier whenever it appears in your C++ source. 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).

Note: The suboption is case-sensitive.

Effect on IPA Compile Step

The KEYWORD option has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

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

LANGLVL

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
LANGLVL(EXTENDED), NORTTI, TMPLPARSE(NO), INFO(LAN)	LANGLVL(ANSI)	LANGLVL (COMMONC)	LANGLVL (EXTENDED, NOLIBEXT, NOLONGLONG), NORTTI, TMPLPARSE(NO), INFO(LAN)			

CATEGORY: Programming Language Characteristics Control for C

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.

The following suboptions are only available under z/OS C:

EXTENDED
: It indicates all language constructs available with z/OS C. It enables extensions to the ISO C standard. The macro __EXTENDED__is defined as 1.

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 bitfields
  - Bitfields other than int are tolerated, and a warning message is issued
  - Macro parameters within quotation marks are expanded
  - Macros may be redefined without first being undefined
  - 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 83 for more information.

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. It ensures 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 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.

`SAA` Indicates language constructs that are defined by SAA. See the *C/C++ Language Reference* for more information.

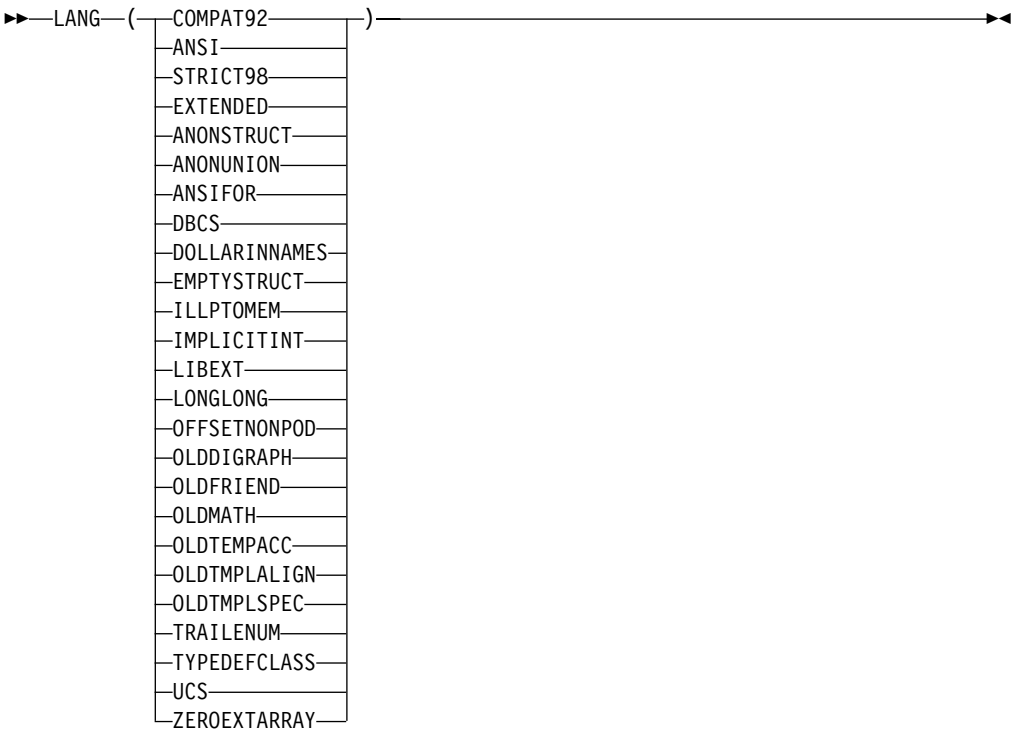
`SAAL2` Indicates language constructs that are defined by SAA Level 2. See the *C/C++ Language Reference* for more information.

`LIBEXT|NOLIBEXT`
Specifying this option affects the C run-time provided headers, which in turn control the availability of general ISO run-time extensions. In addition, it also defines the following macro and sets its value to 1:

- _MI_BUILTIN (this macro controls the availability of machine built-in instructions. Refer to the section on ″using built-in functions″ in the *z/OS C/C++ Programming Guide*)

`LONGLONG|NOLONGLONG`
This option controls the availability of long long integer types for your compilation.

CATEGORY: Programming Language Characteristics Control for C++



Three 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. The macro `__ANSI__` is defined as 1. It ensures that the compilation conforms to the ISO C and C++ standards.

`LANGLVL(EXTENDED)`
: This option group indicates all language constructs available with z/OS C/C++. It enables extensions to the ISO C/C++ standard. The macro `__EXTENDED__`is defined as 1.

The options and settings included in the `COMPAT92`, `STRICT98/ANSI`, and `EXTENDED` groups are listed in the table below. 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 20. Compatibility Options for OS/390 V2R10 and V1R2 Compilers

Options	**Group names**		
	compat92	**strict98/ ansi**	**extended**
KEYWORD(bool) | NOKEYWORD(bool)	Off	On	On
KEYWORD(explicit) | NOKEYWORD(explicit)	Off	On	On
KEYWORD(export) | NOKEYWORD(export)	Off	On	On
KEYWORD(false) | NOKEYWORD(false)	Off	On	On
KEYWORD(mutable) | NOKEYWORD(mutable)	Off	On	On
KEYWORD(namespace) | NOKEYWORD(namespace)	Off	On	On
KEYWORD(true) | NOKEYWORD(true)	Off	On	On
KEYWORD(typename) | NOKEYWORD(typename)	Off	On	On
KEYWORD(using) | NOKEYWORD(using)	Off	On	On
LANGLVL(ANONSTRUCT | NOANONSTRUCT)	Off	Off	On
LANGLVL(ANONUNION | NOANONUNION)	On	Off	On
LANGLVL(ANSIFOR | NOANSIFOR)	Off	On	On
LANGLVL(ILLPTOMEM | NOILLPTOMEM)	On	Off	On
LANGLVL(IMPLICITINT | NOIMPLICITINT)	On	Off	On
LANGLVL(LIBEXT | NOLIBEXT)	On	Off	On

Table 20. Compatibility Options for OS/390 V2R10 and V1R2 Compilers (continued)

Options	Group names		
	compat92	**strict98/ ansi**	**extended**
LANGLVL(LONGLONG | NOLONGLONG)	On	Off	On
LANGLVL(OFFSETNONPOD | OFFSETNONPOD)	On	Off	On
LANGLVL(OLDDIGRAPH | OLDDIGRAPH)	Off	On	Off
LANGLVL(OLDFRIEND | NOOLDFRIEND)	On	Off	On
LANGLVL(OLDMATH | NOOLDMATH)	On	Off	Off
LANGLVL(OLDTEMPACC | NOOLDTEMPACC)	On	Off	On
LANGLVL(OLDTMPLALIGN | NOOLDTMPLALIGN)	On	Off	Off
LANGLVL(OLDTMPLSPEC | NOOLDTMPLSPEC)	On	Off	On
LANGLVL(TRAILENUM | NOTRAILENUM)	On	Off	On
LANGLVL(TYPEDEFCLASS | TYPEDEFCLASS)	On	Off	On
LANGLVL(ZEROEXTARRAY | NOZEROEXTARRAY)	Off	Off	On
RTTI | NORTTI	Off	On	On
TMPLPARSE(NO | ERROR | WARN)	NO	WARN	WARN

You can control individual language features in the z/OS V1R2 C++ compiler by using the `LANGLVL` and `KEYWORD` suboptions listed in Table 20 on page 142. 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.

For C++, the following suboptions apply:

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)`.

`ANONSTRUCT|NOANONSTRUCT`

This option controls whether anonymous structs and anonymous classes are allowed in your C++ source. When `LANG(ANONSTRUCT)` is specified, z/OS C++ allows anonymous structs. This is an extension to the C++ standard. 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 for batch and C++ 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 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 for batch and C++ 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. For 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 for batch and C++ is LANG(ANSIFOR).

DBCS|NODBCS
: This option controls whether multi-byte characters are accepted in string literals and in comments. The default is LANG(NODBCS). The default for batch and C++ is LANG(NODBCS).

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 for batch and C++ is LANG(NODOLLARINNAMES).

EMPTYSTRUCT|NOEMPTYSTRUCT
: This option instructs the compiler to tolerate empty member declarations in

structs. ISO C++ does not permit empty member declaration in structs. 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). The default for batch and C++ is LANG(NOEMPTYSTRUCT).

ILLPTOMEM|NOILLPTOMEM

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(ILLPTOMEM) is in effect, the compiler allows these forms. For 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(NOILLPTOMEM) for compliance with the C++ standard. The example must be modified to use the & operator:

```
struct C {
void foo(int);
};

void (C::*p) (int) = &C::foo;
```

The default for batch and C++ is LANG(ILLPTOMEM).

IMPLICITINT|NOIMPLICITINT

This option controls whether z/OS 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(NOIMPLICITINT) 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. For 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++. For example, the function declaration above must be modified to:

```
int MyFunction()
{
   return 0;
}
```

The default for batch and C++ is LANG(IMPLICITINT).

LIBEXT|NOLIBEXT
: This option controls the macro _EXT that is used to control the availability of general ISO run-time extensions. LANG(LIBEXT) sets _EXT to 1. The default for batch is LANG(LIBEXT) and for C++ is LANG(NOLIBEXT).

LONGLONG|NOLONGLONG
: This option controls the availability of long long integer types for your compilation. The default for batch is LANG(LONGLONG) and for C++ is LANG(NOLONGLONG).

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 nonPOD 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 for batch and C++ 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 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 for batch and C++ 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, the statement below declares the class IFont to be a friend class and is valid when LANG(OLDFRIEND) is in effect.

```
friend IFont;
```

Specify LANG(NOOLDFRIEND) for compliance with ISO standard C++. The example declaration above causes a warning unless you modify it to the statement below, or suppress the message with SUPPRESS(CCN5070) option.

```
friend class IFont;
```

The default for batch and C++ 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 eliminating the possibility of ambiguous overload resolution. The default for batch and C++ is LANG(NOOLDMATH).

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 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++. For 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()
}
```

The example code above contains three ill formed uses of the copy constructor C(const C&). The default for batch and C++ 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 batch and C++ 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 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). For 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 the example above, the template specialization must be modified to:

```
template<class T> class ribbon { /*...*/};
template<> class ribbon<char> { /*...*/};
```

The default for batch and C++ is LANG(OLDTMPLSPEC).

TRAILENUM|NOTRAILENUM

This option controls whether trailing commas are allowed in enum declarations. When LANG(TRAILENUM) is in effect, z/OS 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 for batch and C++ is LANG(TRAILENUM).

TYPEDEFCLASS|NOTYPEDEFCLASS

This option provides backwards compatibility with previous versions of z/OS 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 for batch and C++ 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 for batch and C++ is LANG(NOUCS).

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 C++ allows arrays with zero elements. This is an extension to the C++ standard. The example declarations below 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(NOZEROEXTARRAY) is set, you receive warnings about zero-extent arrays in your code, unless you suppress the message with SUPPRESS(CCN6607). The default for batch and C++ is LANG(ZEROEXTARRAY).

Effect on IPA Compile Step

The LANGLVL option has the same effect on the IPA Compile step as it does on regular compilation.

Effect on IPA Link Step

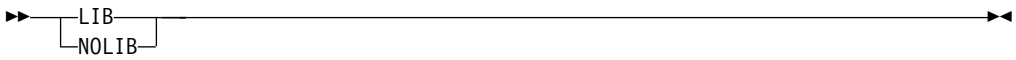
The IPA Link Step accepts but ignores the LANGLVL option.

LIBANSI | NOLIBANSI

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✔	✔	✔	✔	✔

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOLIBANSI	NOLIBANSI	NOLIBANSI	NOLIBANSI			

CATEGORY: Object Code Control



The LIBANSI option indicates whether the functions with the name of an ISO C library function are in fact ISO C library functions. If you specify LIBANSI, the compiler generates code that is based on existing knowledge concerning the behaviour of the ISO C library function; for example, whether or not any side effects are associated with a particular system function.

A comment that indicates the use of the LIBANSI option will be generated in your object module to aid you in diagnosing your program.

You can specify this option using the #pragma option directive for C.

Effect on IPA Compile Step

If you specify the LIBANSI 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.

Effect on IPA Link Step

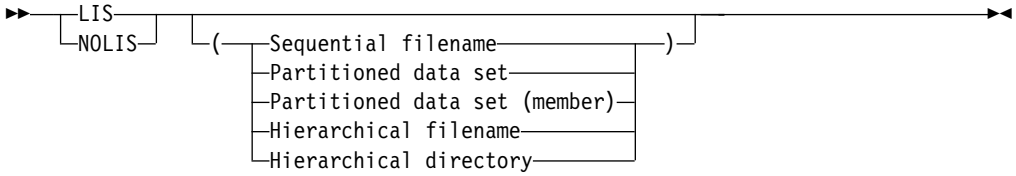
The LIBANSI option will be in effect for the IPA Link step unless the NOLIBANSI option is specified. The value of the LIBANSI option from the IPA compile step is ignored, but is shown in the IPA link listing Compile Option Map for reference.

LIST | NOLIST

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOLIST	NOLIST (/dev/fd1)	NOLIST (/dev/fd1)	NOLIST (/dev/fd1)	NOLIST (/dev/fd1)	NOLIST (/dev/fd1)	NOLIST (/dev/fd1)

CATEGORY: Listing



The LIST option instructs the compiler 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 an HFS file, the compiler stores the listing in a file that has the name of the source file with .lst extension.

The NOLIST option optionally takes a *filename* suboption. This *filename* then becomes the default. If you subsequently use the LIST option without a *filename* suboption, the compiler uses the *filename* that you specified in the earlier NOLIST. For example, the following specifications have the same effect:

```
CXX HELLO (NOLIST(/hello.lis) LIST
CXX HELLO (LIST(/hello.lis)
```

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.

In the z/OS UNIX System Services environment, this option is turned on by specifying -V when using the c89, cc or c++ commands.

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)
   ```

Effect on IPA Compile Step

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 63 and “GONUMBER | NOGONUMBER” on page 121 for more information.

Effect on IPA Link Step

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.

Refer to “Interactions between Compiler Options and IPA Suboptions” on page 63 and “GONUMBER | NOGONUMBER” on page 121 for more information.

LOCALE | NOLOCALE

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities (These 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`.)					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOLOCALE	LOCALE(POSIX)	LOCALE(POSIX)	LOCALE(POSIX)	LOCALE(POSIX)	LOCALE(POSIX)	LOCALE(POSIX)

CATEGORY: Preprocessor

The LOCALE option specifies the locale to be used by the compiler as the current locale throughout the compilation unit. To specify a locale, use the following format:

LOCALE(*name*)

The suboption *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, the compiler ignores the LOCALE, and assumes NOLOCALE.

NOLOCALE indicates that the compiler only uses the default code page, which is IBM-1047.

You cannot use the LOCALE | NOLOCALE option in the z/OS C `#pragma options` directive. You can only specify it 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.

For more information on locales, refer to the *z/OS C/C++ Programming Guide*.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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:

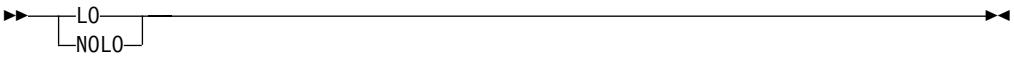
- The Prolog section of the listing displays the LOCALE or NOLOCALE option. If you specified the LOCALE option, the Prolog displays the locale and code set that are in effect.
- The Compiler Options Map listing section displays the LOCALE option active on the IPA Compile step for each IPA object. If 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.

LONGNAME | NOLONGNAME

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
C: NOLONGNAME C++:LONGNAME	LONGNAME	LONGNAME	LONGNAME			

CATEGORY: Object Code Control



The LONGNAME option generates untruncated and mixed case external names in the object module produced by the compiler for functions with non-C++ linkage. Functions with C++ linkage are always untruncated and mixed-case external names. These names may be up to 1024 characters in length. The system binder recognizes the format of long external names in object modules, but the system linkage editor does not.

For z/OS 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.

When you specify NOLONGNAME, only those functions that do not have C++ linkage are given truncated and uppercase names.

A comment that indicates the setting of the LONGNAME option will be generated in your object module to aid you in diagnosing your program.

Effect on IPA Compile Step

You must specify either the LONGNAME compiler option or the #pragma longname preprocessor directive for the IPA Compile step (unless you are using the c89 utility). Otherwise, the compiler issues an unrecoverable error diagnostic message.

Effect on IPA Link Step

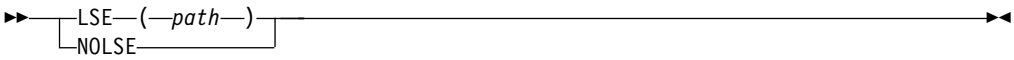
The IPA Link step ignores this option if you specify it, and uses the LONGNAME option for all partitions it generates.

LSEARCH | NOLSEARCH

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOLSEARCH	NOLSEARCH	NOLSEARCH	NOLSEARCH			

CATEGORY: File Management



The LSEARCH option directs the preprocessor to look for the user include files in the specified libraries.

The suboption *path* specifies one of the following:

- The name of a partitioned or sequential data set that contains user include files.
- An HFS path that contains user include files.
- A search path that is more complex. See “Additional Syntax” on page 157 for details.

The #include "*filename*" format of the #include C/C++ preprocessor directive indicates user include files. See “Using Include Files” on page 326 for a description of the #include preprocessor directive.

For further information on library search sequences, see “Search Sequences for Include Files” on page 334.

Searching for PDS or PDSE files

Example

You 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 user/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.

Searching for HFS Files

The compiler forms the search path for HFS 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 HFS file does not exist, the compiler looks in:
   `./sub/fred.h`
3. If that HFS 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 `NOLSEARCH` option instructs the preprocessor to search only those libraries that are specified on the `USERLIB DD` statement. A `NOLSEARCH` option cancels all previous `LSEARCH` specifications, and the compiler uses any `LSEARCH` options that follow it. When you specify more than one `LSEARCH` option, the compiler uses all the libraries in these `LSEARCH` options to find the user include files.

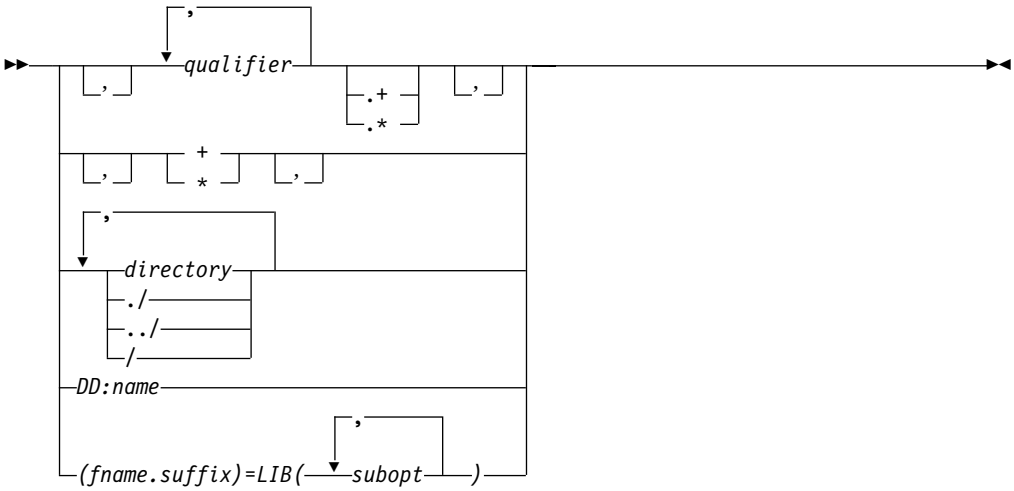
Note: If the *filename* in the `#include` directive is in absolute form, the compiler does not perform a search. See “Determining whether the File Name is in Absolute Form” on page 331 for more details on absolute `#include` *filename*.

Additional Syntax

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 the above syntax diagram, *opt* specifies one of the following:

- The name of a partitioned or sequential data set that contains user include files
- An HFS 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 HFS 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.
  - `LIB`s 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`.

Specifying Hierarchical File System Files

When specifying Hierarchical File System (HFS) library searches, do not put double slashes at the beginning of the `LSEARCH` *opt* . Use *pathnames* separated by slashes (/) in the `LSEARCH` *opt* for an HFS library. When the `LSEARCH` *opt* does not start with double slashes, any single slash in the name indicates an HFS library. If you do not have path separators (/), then setting the `OE` compile option on indicates that this is an HFS library; otherwise the library is interpreted as a data set. See “Using SEARCH and LSEARCH” on page 333 for additional information on HFS files.

The *opt* specified for `LSEARCH` is combined with the *filename* in `#include` to form the include file name, for example:

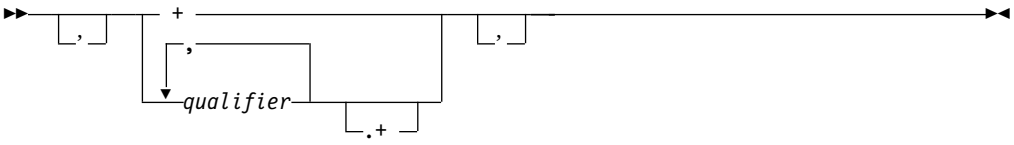
```
LSEARCH(/u/mike/myfiles)
#include "new/headers.h"
```

The resulting HFS file name is `/u/mike/myfiles/new/headers.h`.

Specifying Sequential Data Sets and PDSs

Use an asterisk (*) or a plus sign (+) in the `LSEARCH` *opt* to specify whether the library is a sequential or partitioned data set.

Partitioned Data Set (PDS): 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 leftmost *qualifier*
- For the PDS member name, the leftmost *qualifier* is used as the member name

See the first example in Table 21.

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 21.

See “Forming Data Set Names with LSEARCH | SEARCH Options” on page 328 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)
```

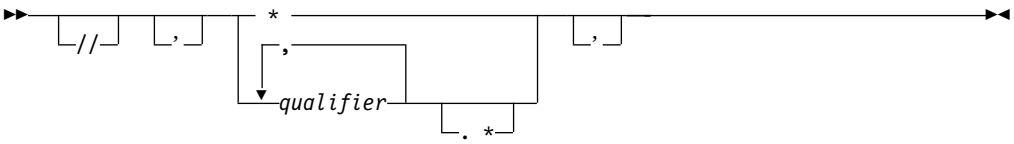
Also see the third example in Table 21.

Examples: The following example shows you how to specify a PDS search path:

Table 21. 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)’

Sequential Data Set: 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).

Examples: The following example shows you how to specify a search path for a sequential data set:

Table 22. 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.

Effect on IPA Compile Step

The `LSEARCH` option has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

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

MARGINS | NOMARGINS

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
C++ and C(V-format): NOMARGINS C(F-format): MARGINS(1,72)	NOMARGINS	NOMARGINS	NOMARGINS			

CATEGORY: Input Source File Processing Control

The MARGINS option specifies 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 on the MARGINS option.

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.

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.

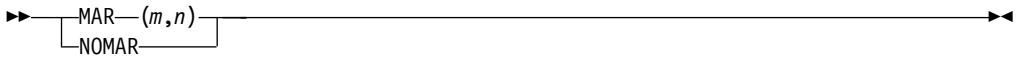
The NOMARGINS option specifies that the entire input source record is to be scanned for input to the compiler.

z/OS C++

C++ now supports the MAR(m,n) syntax.

In a C++ program, the MARGINS option specifies that columns 1 through 72 in the input record are to be scanned for input to the compiler. The compiler ignores any text in the source input that does not fall within that range.

z/OS C



In a C program, the MARGINS option has the following additional syntax:
MARGINS(m,n)

where:

m specifies the first column of the source input that contains valid z/OS 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 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.

If the `MARGINS` option is specified along with the `SOURCE` option in a 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 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 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.

Effect on IPA Compile Step

The `MARGINS` option is used for source code analysis, and has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

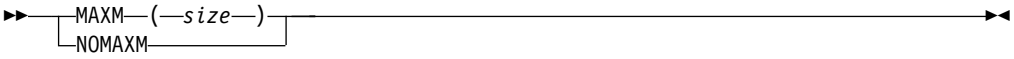
The IPA Link step accepts the `MARGINS` option, but ignores it.

MAXMEM | NOMAXMEM

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
`MAXMEM(2097152)` or `MAXMEM(*)` or `MAXMEM(0)`	`MAXMEM(*)`	`MAXMEM(*)`	`MAXMEM(*)`			

CATEGORY: Object Code Control



When compiling with `OPT`, the `MAXMEM(size)` option limits the amount of memory used for local tables of specific, memory intensive optimizations to *size* kilobytes. 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` option if you want to specify a memory size of less value than the default.

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.

You can specify this option using the `#pragma option` directive for C.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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.

MEMORY | NOMEMORY

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
MEMORY	MEMORY	MEMORY	MEMORY	MEMORY	MEMORY	MEMORY

CATEGORY: File Management

The MEMORY option specifies that the compiler is to use a MEMORY file in place of a work-file if possible. See the *z/OS C/C++ Programming Guide* for more information on memory files.

This option 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.

Effect on IPA Compile Step

The MEMORY compiler option has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

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.

NAMEMANGLING

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
	✔	✔		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NAMEMANGLING			NAMEMANGLING			

CATEGORY: Object Code Control

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. The NAMEMANGLING compiler option enables you to choose between the following two name mangling schemes:

COMPAT It indicates that the name mangling scheme is the same as that in earlier versions of the z/OS C++ and OS/390 C++ compiler, and is provided for compatibility with link modules created with earlier compilers.

ANSI This scheme is the default and complies with the C++ standard.

Effect on IPA Compile Step

The NAMEMANGLING compiler option has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

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

NESTINC | NONESTINC

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NESTINC(255)	NESTINC(255)	NESTINC(255)	NESTINC(255)			

CATEGORY: Input Source File Processing Control

The NESTINC option specifies the number of nested include files to be allowed in your source program. 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.

Specifying NONESTINC is equivalent to specifying NESTINC(255).

Note: If you use heavily nested include files, your program requires more storage to compile.

Effect on IPA Compile Step

The NESTINC option is used for source code analysis, and has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

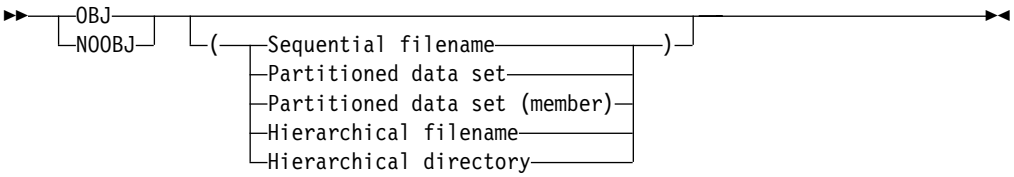
The IPA Link step accepts the NESTINC option, but ignores it.

OBJECT | NOOBJECT

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities Note that this changes if the -o flag is set.					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
OBJECT	OBJECT (*file_name*.o)	OBJECT (*file_name*.o)	OBJECT (*file_name*.o)	OBJECT (//DD:SYSLIPA)	OBJECT (//DD:SYSLIPA)	OBJECT (//DD:SYSLIPA)

CATEGORY: File Management and Object Code Control



The OBJECT option specifies whether the compiler is to produce an object module.

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 an HFS file, the compiler stores the object module in a file that has the name of the source file with an .o extension.

The NOOBJ option can optionally take a *filename* suboption. This *filename* then becomes the default. If you subsequently use the OBJ option without a *filename* suboption, the compiler uses the *filename* that you specified in the earlier NOOBJ. 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.

You can specify this option using the #pragma option directive for C.

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, or c++ commands.

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)
```

Effect on IPA Compile Step

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 133 for information about the IPA(OBJECT) suboption.

Effect on IPA Link Step

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

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 { _TMPS} environment variable information in the c89 section of *z/OS UNIX System Services Command Reference*.

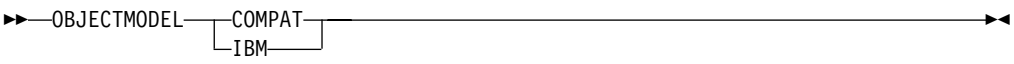
Note: The OBJECT compiler option is not the same as the OBJECT suboption of the IPA option. Refer to “IPA | NOIPA” on page 133 for information about the IPA(OBJECT) option.

OBJECTMODEL

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
OBJECTMODEL(COMPAT)						

CATEGORY: Object Code Control



The OBJECTMODEL compiler option sets the type of object model.

z/OS V1R2 C++ includes two ways to compile your programs using different object models. The two object models differ in the following areas:

- Layout for the virtual function table
- Name mangling scheme

The two object models are:

- COMPAT
- IBM

COMPAT is compatible with name mangling and the virtual function table that was available with the previous releases of the C++ compiler.

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.

Object model usage can be mixed in a single program (and a single object file as well). As described below, differing object models are not allowed in the same inheritance hierarchy. The different object models have different name-mangling schemes. Functions can take parameters of different object models. When using pre-built libraries, you should wrap the library headers with `#pragma object_model(compat)` and `#pragma object_model(pop)` (this is done in order to ensure that name-mangling for items declared in these headers are set up using the correct name-mangling scheme). When shipping library headers, you should either provide multiple versions (different object models) or ensure correct object model usage by placing `#pragma object_model(compat[or ibm if your library is using the ibm model])` and `#pragma object_model(pop)` appropriately.

All classes in the same inheritance hierarchy must have the same object model. Classes implicitly inherit the object model of their parent, overriding any local object model specification. An error is generated (CCN8200) if, through multiple inheritance, different object models are mixed; for example:

```
#pragma object_model(ibm)
class A{};  // ibm model: pragma is used
#pragma object_model(compat)
class B: A{};  // ibm model: pragma is ignored because of inheritance
                        // (A is "ibm", therefore B is "ibm")
#pragma object_model(ibm)
class C: B{};  // ibm model: pragma is ignored because of inheritance
                        // (B is "ibm", therefore C is "ibm")
#pragma object_model(compat)
class D{};  // compat model: no inheritance, pragma is used
class E: A, D{};  // error CCN8200: A and D have differing object models.
```

Effect on IPA Compile Step

The OBJECTMODEL compiler option has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

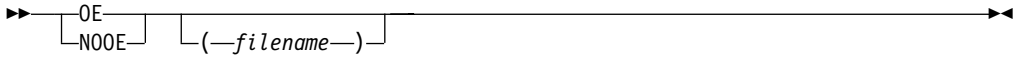
The IPA Link step accepts the OBJECTMODEL option, but ignores it.

OE | NOOE

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOOE	OE	OE	OE	OE	OE	OE

CATEGORY: File Management



Note: Diagnostics and listing information will refer to the file name that is specified for the OE option (in addition to the search information).

The OE option specifies that the compiler use the POSIX.2 standard rules for searching for files specified with #include directives. These rules state that the current 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. For example, the following specifications have the same result:

```
CXX HELLO (NOOE(/hello.c) OE
CXX HELLO (OE(/hello.c)
```

If you specify OE and NOOE 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 (NOOE(/hello.c) OE(/n1.c)  NOOE(/test.c) OE
CXX HELLO (OE(/test.c)
```

When the OE option is in effect and the main input file is an HFS 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 HFS, the presence of the OE option tells the compiler to interpret the ambiguous names as HFS 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 an HFS file, *filename* is ignored.

For example, if the compiler is invoked to compile HFS 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.

Effect on IPA Compile Step

The OE compiler option has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

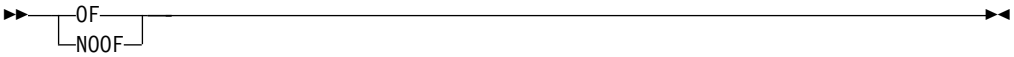
On the IPA Link step, the OE option controls the display of file names.

OFFSET | NOOFFSET

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOOFFSET	NOOFFSET	NOOFFSET	NOOFFSET	NOOFFSET	NOOFFSET	NOOFFSET

CATEGORY: Listing



The OFFSET option instructs the compiler to display, in the pseudo-assembly listing generated by the LIST option, the offset addresses relative to the entry point or start of each function.

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.

The NOOFFSET option specifies that the compiler is to display, in the pseudo-assembly listing generated by the LIST option, the offset addresses relative to the beginning of the generated code and not the entry point.

In the z/OS UNIX System Services environment, this option is turned on by specifying -V when using the c89, cc or c++ commands.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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.

OPTFILE | NOOPTFILE

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOOPTFILE						

CATEGORY: File Management

The OPTFILE option directs the compiler to look for compiler options in the file specified by *filename*.

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.

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. For 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. 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.
2. 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.
3. If the file cannot be opened or cannot be read, a warning message is issued and the OPTFILE option is ignored.
4. The options file can be an empty file.
5. You can use an option file only once in a compilation. For example, 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.

   You can specify OPTFILE more than once in a compilation, if you use a different options file with each specification. For example:

   ```
   OPTFILE(DD:OF)   OPTFILE(DD:OF1)
   ```

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
@@
```

Effect on IPA Compile Step

The OPTFILE option has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

The OPTFILE option has the same effect on the IPA Link step as it does on a regular compilation.

OPTIMIZE | NOOPTIMIZE

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✔	✔	✔	✔	✔

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities Note that the default is set to OPTIMIZE(1) if the -WI flag is set.					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
C and C++ compile: NOOPTIMIZE IPA Link: OPTIMIZE(2)	OPTIMIZE(0) for no IPA, OPTIMIZE(1) for IPA Compile	OPTIMIZE(0) for no IPA, OPTIMIZE(1) for IPA Compile	OPTIMIZE(0) for no IPA, OPTIMIZE(1) for IPA Compile	OPTIMIZE(1)	OPTIMIZE(1)	OPTIMIZE(1)

CATEGORY: Object Code Control

The OPTIMIZE option instructs the compiler 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. 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.

A list of the valid suboptions for OPT and their descriptions follow. *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 ensures that you will have compatibility with future compilers.

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.

no level
OPTIMIZE specified with no level defaults, depending on the compilation environment and IPA mode. See the Option Default table above for details.

In the z/OS UNIX System Services environment, this option is turned on by specifying -O (the letter) or -0 (the number), -1, or -2 when using the c89, cc, or c++ commands.

You can specify this option using the #pragma option directive for C.

You can specify this option for a specific subprogram using the #pragma option_override(subprogram_name, "OPT(LEVEL,n)") directive.

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 128 for more information about the INLINE option and the *z/OS C/C++ Programming Guide* for more information about optimization.

If you specify OPTIMIZE with TEST, 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. For more information, see “ANSIALIAS | NOANSIALIAS” on page 83.

Effect on IPA(OBJONLY) Compilation

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.

Effect on IPA Compile Step

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 133 for information about using the OPTIMIZE option under IPA.

Effect on IPA Link Step

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(1) or 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(1) or 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).

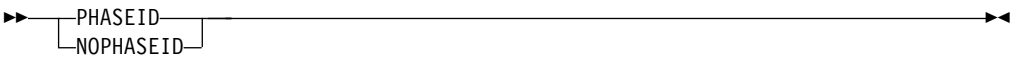
Refer to the descriptions for the OPTIMIZE and LEVEL suboptions of the IPA option in “IPA | NOIPA” on page 133 for information about using the OPTIMIZE option under IPA.

PHASEID | NOPHASEID

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOPHASEID						

CATEGORY: Debug/Diagnostic



If you specify the PHASEID option, it causes each compiler component (phase) to issue an informational message as the phase begins execution. This message identifies compiler phase module name, product identification, and build level. Use the PHASEID option to assist you with determining the maintenance level of each compiler component (phase).

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.

Effect on IPA Compile Step

The PHASEID option has the same effect on the IPA Compile Step as it does on a regular compilation.

Effect on IPA Link Step

The IPA Link step uses the PHASEID option that you specify for that step.

PLIST

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
PLIST(HOST)	PLIST(HOST)	PLIST(HOST)	PLIST(HOST)			

CATEGORY: Program Execution

When compiling `main()` programs, use the `PLIST` option to direct how the parameters from the caller are passed to `main()`.

If you specify `PLIST(HOST)`, the parameters are presented to `main()` as an argument list (`argv, argc`).

If you specify `PLIST(OS)`, the parameters are passed without restructuring, and the standard calling conventions of the operating system are used. See the *z/OS Language Environment Programming Guide* for details on how to access these parameters.

If you are compiling a `main()` program to run under IMS, you must specify the `PLIST(OS)` and `TARGET(IMS)` options together.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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()`.

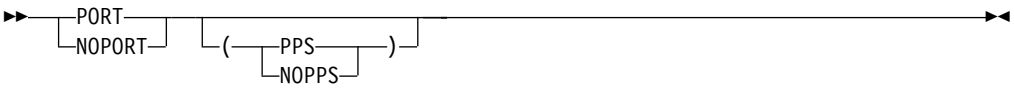
If you specify this option on both the IPA Compile and the IPA Link steps, the setting on the IPA Link step overrides the setting on the IPA Compile step. This situation occurs whether you use `PLIST` as a compiler option or specify it using the `#pragma runopts` directive (on the IPA Compile step).

PORT | NOPORT

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
	✓			

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
`NOPORT(NOPPS)`						

CATEGORY: Portability



The PORT option allows you to adjust the error recovery action that the compiler takes when it encounters an ill-formed #pragma pack directive. When you specify PORT(PPS), the compiler uses the strict error recovery mode. When you specify any other value for either PORT or NOPORT, the compiler uses the default error recovery mode. When you specify PORT without a suboption, the suboption setting is inherited from the default setting or from previous PORT specifications.

Default Error Recovery

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 S/390 value 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 S/390 value 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 S/390 value 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 S/390 value 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 does not issue a message.

Strict Error Recovery

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. For example, 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
```

Effect on IPA Compile Step

The PORT option is used for source code analysis, and has the same effect on the IPA compile step as it does on a regular compile.

Effect on IPA Link Step

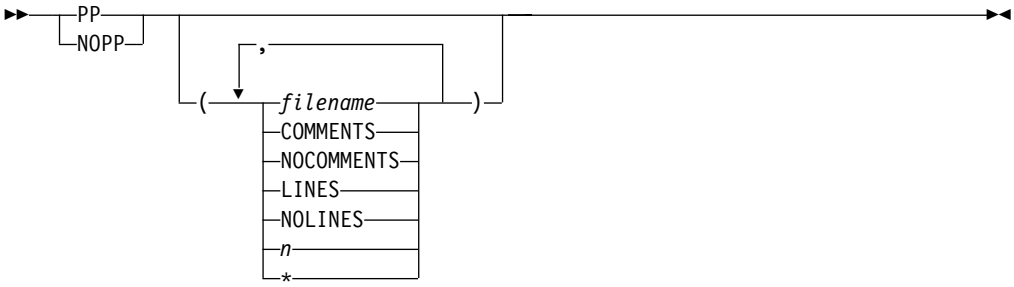
The IPA link step issues a diagnostic message if you specify the PORT option for that step.

PPONLY | NOPPONLY

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOPPONLY	NOPPONLY (NOCOMMENTS, NOLINES, /dev/fd1, 2048)	NOPPONLY (NOCOMMENTS, NOLINES, /dev/fd1, 2048)	NOPPONLY (NOCOMMENTS, NOLINES, /dev/fd1, 2048)			

CATEGORY: Preprocessor



The PPONLY option specifies that only the preprocessor is to be run against the source file. This 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.

The suboptions are:

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 an HFS 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 an HFS file, the preprocessed output is written to an HFS file that has the source file name with .i extension.

n — If a parameter *n*, which is an integer between 2 and 32760 inclusive, is specified, all lines are folded at column *n*.

***** — If an asterisk (*) is specified, the lines are folded at the maximum record length of 32760. Otherwise, all lines are folded to fit into the output file, based on the record length of the output file.

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. For 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 the above, ??= 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 the *z/OS C/C++ Programming Guide*.

The NOPPONLY option specifies that both the preprocessor and the compiler are to be run against the source file.

If you specify both PPONLY and NOPPONLY, the last one that is specified is used.

In the z/OS UNIX System Services environment, this option is turned on by specifying -E when using the c89, cc or c++ commands. To turn on the COMMENTS suboption, specify -C. The user cannot specify PPONLY, they must use -E and -C. The {_ELINES} envar is also relevant (for further information on {_ELINES}, refer to “Environment Variables” on page 589). The output always goes to stdout.

Effect on IPA Compile Step

The PPONLY has the same effect on the IPA Compile step as it does on a regular compilation. It processes source code, then causes the compiler to stop processing before it begins the IPA Compile step. You should not use this option for the IPA Compile step.

Effect on IPA Link Step

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

REDIR | NOREDIR

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
REDIR	REDIR	REDIR	REDIR			

CATEGORY: Program Execution

The REDIR option directs the compiler to create 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 specified from the UNIX shell).

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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().

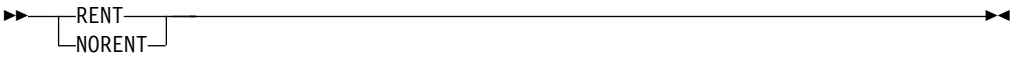
If you specify this option on both the IPA Compile and the IPA Link steps, the setting on the IPA Link step overrides the setting on the IPA Compile step. This situation occurs whether you use REDIR and NOREDIR as compiler options or specify them using the #pragma runopts directive (on the IPA Compile step).

RENT | NORENT

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓		✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NORENT	RENT	RENT				

CATEGORY: Object Code Control



The RENT option specifies that the compiler is to take code that is not naturally reentrant and make it reentrant. Refer to the *z/OS Language Environment Programming Guide* for a detailed description of reentrancy.

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 10. Binding z/OS C/C++ Programs" on page 365, or the prelinker, which is described in "Appendix A. Prelinking and Linking z/OS C/C++ Programs" on page 485.

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 C++ code always uses constructed reentrancy.
3. RENT variables reside in the modifiable writable static area for both z/OS C and z/OS C++ programs.
4. NORENT variables reside in the code area (which may be write protected) for both z/OS C and z/OS C++ programs.

The NORENT option specifies that the compiler is not to specifically generate reentrant code from non-reentrant code. Any naturally reentrant code remains reentrant.

You can specify this option using the `#pragma option` directive for C.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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.

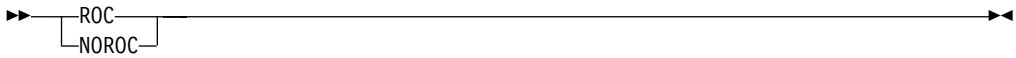
If you generate an IPA Link listing by using the LIST or MAP compiler option, the IPA Link step generates a Partition Map listing section for each partition. If any symbols within a partition are reentrant, the options section of the Partition Map displays the RENT compiler option.

ROCONST | NOROCONST

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOROCONST			ROCONST			

CATEGORY: Object Code Control



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.

Use ROCONST with the RENT option so that a const variable is not placed into the Writeable Static Area. This reduces the memory requirement for DLLs. This option has the same effect for all const variables as the #pragma variable(*var_name*, NORENT) directive. See the *C/C++ Language Reference* for more information on pragma directives.

Note that such const variables cannot be exported.

Interaction with #pragma variable

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.

Interaction with #pragma export

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.

Effect on IPA Compile Step

If you specify the ROCONST option during the IPA Compile step, the information in the IPA object file reflects the state of each symbol.

Effect on IPA Link Step

If you specify the ROCONST option on the IPA Link step, it ignores the option. The reentrant/nonreentrant and const/nonconst state of each symbol are 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. Subprograms that follow are placed in partitions that have the same ROCONST setting. A NOROCONST mode is placed in a NOROCONST partition, and a ROCONST is placed in a ROCONST 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 reentrant/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.

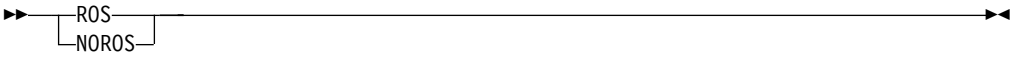
The Partition Map sections of the IPA Link step listing and the object module END information section display the value of the ROCONST option.

ROSTRING | NOROSTRING

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
ROSTRING						

CATEGORY: Object Code Control



The ROSTRING option informs the compiler that string literals are read-only. This option has the same effect as the #pragma strings(readonly) directive. See the *C/C++ Language Reference* for more information on pragma directives.

Specifying the ROSTRING option allows the compiler to place string literals into read-only memory. When you compile the program with the RENT option, such string literals are not placed into the Writeable Static Area (WSA). This reduces the memory requirement for DLLs.

Effect on IPA Compile Step

If you specify the ROSTRING option during the IPA Compile step, the information in the IPA object file reflects the state of each symbol.

Effect on IPA Link Step

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.

ROUND

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✔	✔	✔	✔	✔

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
For IEEE: ROUND(N) For HEX: ROUND(Z)						

CATEGORY: Object Code Control

The ROUND(*mode*) option sets the rounding mode for floating-point compilations at compile time where *mode* can be one of the following:

N	round to the nearest representable number
M	round towards minus infinity
P	round towards plus infinity
Z	round towards zero
ROUND()	is the same as ROUND(N)

The ROUND(mode) option only applies to IEEE floating-point mode. In hexadecimal mode, the rounding is always towards zero. If you specify ROUND(mode) in hexadecimal floating-point mode, where mode is not Z, the compiler ignores ROUND(mode) and issues a warning.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

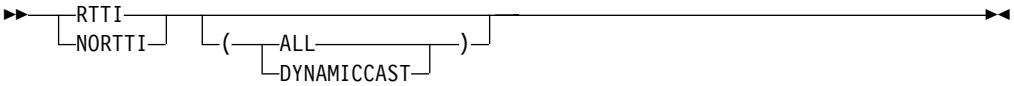
The IPA Link step merges and optimizes the application code, and then divides it into sections for code generation. Each of these section is a partition. The IPA Link step uses information from the IPA Compile step to ensure that an object is included in a compatible partition. Refer to the “FLOAT” on page 116 for further information.

RTTI | NORTTI

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
	✔	✔		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NORTTI			NORTTI			

CATEGORY: Program Execution



Use the RTTI option to generate run-time type identification (RTTI) information for the typeid operator and the dynamic_cast operator. For best run-time performance, suppress RTTI information generation with the default NORTTI setting.

The C++ language offers a (RTTI) mechanism for determining the class of an object at run time. It consists of two operators:

- One for determining the run-time type of an object (typeid), and
- One for doing type conversions that are checked at run time (dynamic_cast)

The suboptions are:

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.

Note: Even though the default is NORTTI, if you specify LANGLVL(EXTENDED) you will also implicitly select RTTI.

Effect on IPA Compile Step

The RTTI compiler option has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

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

SEARCH | NOSEARCH

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✔	✔	✔		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities The 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 Appendix F for more information. • z/OS UNIX environment variable settings: {_INCDIRS}, {_INCLIBS}, and {_CSYSLIB}. They are normally set during compiler installation to reflect the compiler and run-time include libraries. Refer to "Environment Variables" on page 589 for more information. This option is specified as NOSEARCH, SEARCH by these utilities, so it resets the SEARCH parameters you specify.					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
For C++, SE(//'CEE.SCEEH.+, //'CBC.SCLBH.+') For C, SE(//'CEE.SCEEH.+')						

CATEGORY: File Management

The SEARCH option directs the preprocessor to look for system include files in the specified libraries. 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 326 for a description of the #include preprocessor directive.

For further information on library search sequences, see "Search Sequences for Include Files" on page 334.

The suboptions for the SEARCH option are identical to those for the LSEARCH option, as described on page "LSEARCH | NOLSEARCH" on page 155.

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 SEARCH options that follow it are used. When more than one SEARCH compile option is specified, all libraries in the SEARCH options are used to find the system include files.

The NOSEARCH option instructs the preprocessor to search only those libraries that are specified on the SYSLIB DD statement.

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 331 for more details on absolute #include *filename*.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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

SERVICE | NOSERVICE

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOSERVICE						

CATEGORY: Debug/Diagnostic

The SERVICE option places a string in the object module. The string is loaded into memory when the program is executing. If the application fails abnormally, the string is displayed in the traceback.

For z/OS C, you can also specify this option in the source file by using the #pragma options directive. 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 quotes.

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 quotes 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.

You can specify this option using the #pragma option directive for C.

Effect on IPA Compile Step

The SERVICE option has the same effect on the IPA Compile step (if you request code generation by specifying the OBJECT suboption of the IPA option) as it does on a regular compilation.

Effect on IPA Link Step

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.

SEQUENCE | NOSEQUENCE

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
C++ and C(V-format and HFS): NOSEQUENCE C(F-format): SEQUENCE(73,80)	NOSEQUENCE	NOSEQUENCE	NOSEQUENCE			

CATEGORY: Input Source File Processing Control

z/OS C

The SEQUENCE option 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.

The SEQUENCE option has the following suboptions:

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.

Note: If your program uses the #include preprocessor directive to include z/OS 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 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.

Effect on IPA Compile Step

The SEQUENCE option is used for source code analysis, and has the same effect on an IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

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

SHOWINC | NOSHOWINC

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOSHOWINC	NOSHOWINC	NOSHOWINC	NOSHOWINC			

CATEGORY: Listing

The SHOWINC option instructs the compiler to show, in both the compiler listing and the pseudo-assembler listing, all include files processed. In the listing, the compiler replaces all #include preprocessor directives with the source that is contained in the include file. This option only applies if you also specify the SOURCE option.

In the z/OS UNIX System Services environment, this option is turned on by specifying -V when using the c89, cc or c++ commands.

Effect on IPA Compile Step

The SHOWINC option has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

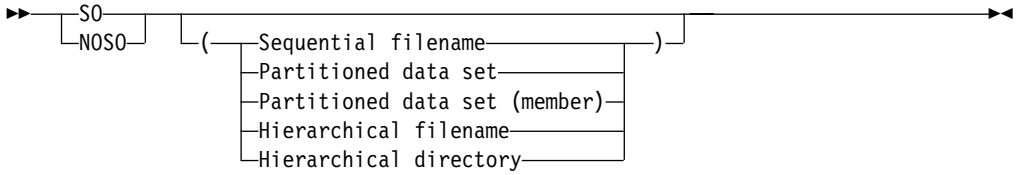
The IPA Link step accepts the SHOWINC option, but ignores it.

SOURCE | NOSOURCE

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOSOURCE	NOSOURCE (/dev/fd1)	NOSOURCE (/dev/fd1)	NOSOURCE (/dev/fd1)			

CATEGORY: Listing



The SOURCE option generates a listing that shows the original source input statements plus any diagnostic messages.

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 an HFS 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 *filename* suboption. This *filename* then becomes the default. If you subsequently use the SOURCE option without a *filename* suboption, the compiler uses the *filename* that you specified in the earlier NOSOURCE. For 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)
```

In the z/OS UNIX System Services environment, this option is turned on by specifying -V when using the c89, cc or c++ commands.

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)
   ```

Effect on IPA Compile Step

The SOURCE option has the same effect on the IPA Compile step that it does on a regular compilation.

Effect on IPA Link Step

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

SPILL | NOSPILL

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
SPILL(128)	SPILL(128)	SPILL(128)	SPILL(128)			

CATEGORY: Object Code Control

The SPILL option specifies the size of the spill area to be used for the compilation. When too many registers are in use at once, the compiler dumps some of the registers into temporary storage, called the spill area.

If you have to expand the spill area, you will receive a compiler message telling you the size to which you should increase it. Once you know the spill area that your source program requires, you can specify the required *size* (in bytes) as shown in the syntax diagram above. Alternatively for C code, you can add a `#pragma options(SPILL(size))` directive to your source. 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.

Notes:

1. 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.
2. If you specify NOSPILL, the compiler defaults to SPILL(128).

You can specify the SPILL option using the `#pragma option` directive for C.

You can specify this option for a specific subprogram using the `#pragma option_override(subprogram_name, "OPT(SPILL,size)")` directive.

Effect on IPA Compile Step

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) option.

Effect on IPA Link Step

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.

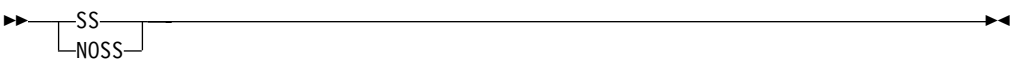
The Partition Map sections of the IPA Link step listing and the object module END information section display the value of the SPILL option. The Partition Map also displays any subprogram-specific SPILL values.

SSCOMM | NOSSCOMM

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓		✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOSSCOMM	NOSSCOMM	NOSSCOMM				

CATEGORY: Programming Language Characteristics Control



The SSCOMM option 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.

If you include your z/OS C program in your JCL stream, be sure to change the delimiters so that your comments are recognized as z/OS C comments and not as JCL statements. For example:

```
//COMPILE.SYSIN DD DATA,DLM=@@
#include <stdio.h>
void main(){
// z/OS C comment
printf("hello world\n");
// A nested z/OS C /*  */ comment
}
@@
//* JCL comment
```

NOSSCOMM indicates that /* */ is the only valid comment format.

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.

Effect on IPA Compile Step

The SSCOMM option is used for source code analysis, and has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

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

START | NOSTART

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
START	START	START	START	START	START	START

CATEGORY: Object Code Control

The START option specifies that CEESTART is to be generated whenever necessary.

NOSTART indicates that CEESTART is never to be generated.

Whenever you specify the START compiler option, a comment that indicates its use will be generated in your object module to aid you in diagnosing your program.

You can specify this option using the #pragma option directive for C.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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.

STATICINLINE | NOSTATICINLINE

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOSTATICINLINE						

CATEGORY: Programming Language Characteristics Control

The STATICINLINE option treats an inline function as static instead of extern. The z/OS V1R2 C/C++ compiler treats inline functions as extern. Previous versions of the z/OS C/C++ and OS/390 C/C++ compiler treated the inline functions as static.

Specify the STATICINLINE option for compatibility with C++ compilers provided by previous versions of the compiler.

For example, using the STATICINLINE compiler option causes function f in the following declaration to be treated as static, even though it is not explicitly declared as such.

```
inline void f() {/*...*/};
```

Using the NOSTATICINLINE compiler option gives f external linkage.

Effect on IPA Compile Step

The STATICINLINE compiler option has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

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

STRICT | NOSTRICT

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
STRICT						

CATEGORY: Object Code Control

The STRICT option instructs the compiler to perform computational operations in a rigidly-defined order such that the results are always determinable and recreatable.

NOSTRICT allows the compiler to reorder certain computations for better performance. However, the end result may differ from the result obtained when STRICT is specified.

In IEEE floating-point mode, NOSTRICT defaults to FLOAT(MAF). To avoid this behavior, explicitly specify FLOAT(NOMAF).

You can specify this option for a specific subprogram using the #pragma option_override(subprogram_name, "OPT(STRICT)") directive.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

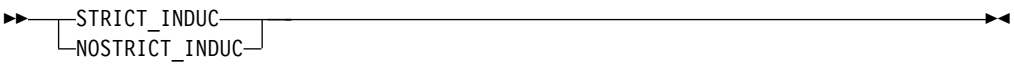
The IPA Link step merges and optimizes the application code, and then divides it into sections for code generation. Each of these sections is a partition. The IPA Link step uses information from the IPA Compile step to ensure that an object is included in a compatible partition. See “FLOAT” on page 116 for more information on the effect of the STRICT option on the IPA Link step.

STRICT_INDUCTION | NOSTRICT_INDUCTION

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOSTRICT_INDUCTION						

CATEGORY: Object Code Control



The STRICT_INDUCTION option instructs the compiler to disable loop induction variable optimizations. These optimizations have the potential to alter the semantics of your program. Such 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.

You can use the ST_IND alias for STRICT_INDUCTION, and the NOST_IND alias for NOSTRICT_INDUCTION.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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.

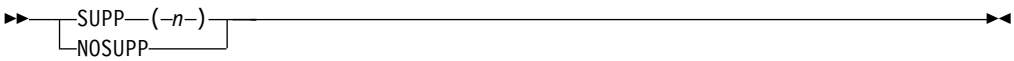
You can override the setting of STRICT_INDUCTION by specifying the option on the IPA Link step. If you do so, all partitions will contain that value, and the prolog section of the IPA Link step listing will display the value.

SUPPRESS| NOSUPPRESS

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOSUPPRESS						

CATEGORY: Debug/Diagnostic



Note: n is a comma separated list of messages IDs.

The SUPPRESS option prevents certain compiler informational or warning messages from being printed on the listings. The message ID range that is affected is CCN3000 through CCN3999, and CCN8000 through CCN9999. 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.

Effect on IPA Compile Step

The SUPPRESS option has the same effect on the IPA Compile step that is does on a regular compilation.

Effect on IPA Link Step

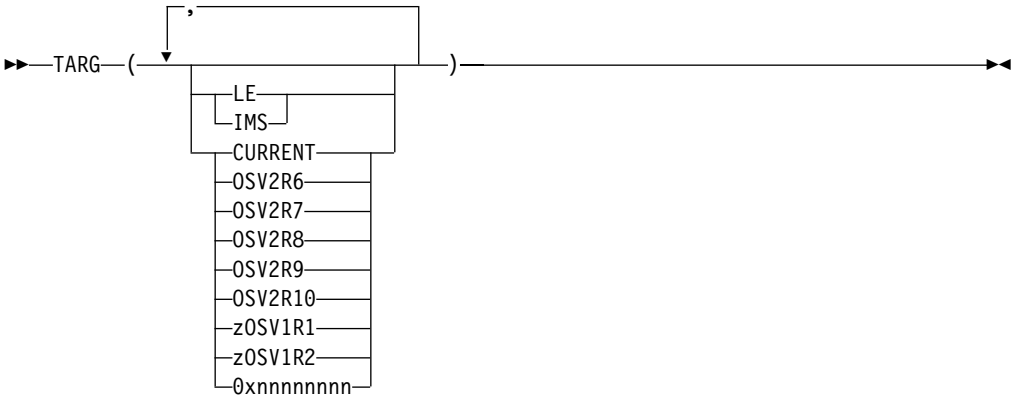
The IPA Link step accepts the SUPPRESS option, but ignores it.

TARGET

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
TARGET(LE, CURRENT)	TARGET(LE)	TARGET(LE)	TARGET(LE)			

CATEGORY: Program Execution and Object Code Control



Note: Suboptions 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 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 targetted 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.

To use the TARGET option, select a run-time environment of either LE or IMS. Then select the desired release, for example, CURRENT or OSV2R10. If you do not select a run-time environment or release, the compiler uses a default of TARGET(LE, CURRENT).

TARGET() Generates object code to run under z/OS Language Environment. It is the same as TARGET (LE, CURRENT).

The following suboptions target the run-time environment:

TARGET(LE) Generates object code to run under z/OS Language Environment. This is the default.

TARGET(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.

For more information about these suboptions refer to "TARGET Run-time Environment Suboptions (LE,IMS)" on page 204.

The following suboptions target the release at program run time:

TARGET(CURRENT) Generates object code to run under the same version of z/OS with which the compiler ships. As the compiler ships with V1R2 of z/OS, TARGET(CURRENT) is the same as TARGET(zOSV1R2). This is the default. [2]

TARGET(OSV2R6) Generates object code to run under OS/390 Version 2 Release 6 and subsequent releases.

TARGET(OSV2R7) Generates object code to run under OS/390 Version 2 Release 7 and subsequent releases.

TARGET(OSV2R8) Generates object code to run under OS/390 Version 2 Release 8 and subsequent releases.

TARGET(OSV2R9) Generates object code to run under OS/390 Version 2 Release 9 and subsequent releases.

TARGET(OSV2R10) Generates object code to run under OS/390 Version 2 Release 10 and subsequent releases.

TARGET(zOSV1R1) Generates object code to run under z/OS Version 1 Release 1 and subsequent releases.

TARGET(zOSV1R2) Generates object code to run under z/OS Version 1 Release 2 and subsequent releases.

2. Note that for some releases of z/OS, z/OS C/C++ might not ship a new version of the compiler. The same version of the compiler is then shipped 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 an zOSV*x*R*y* suboption that corresponds to a release that is earlier or the same as CURRENT. You cannot specify an zOSV*x*R*y* suboption that corresponds to a release later than CURRENT.

TARGET(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. For more information on using this literal, please see “Using the Hexadecimal String Literal Suboption” on page 202.

For more information about these suboptions refer to “TARGET Release Suboptions”.

The compiler generates a comment that indicates the value of TARGET in your object module to aid you in diagnosing problems in your program.

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,0x22060000, IMS, OSV2R8, LE) resolves to TARGET(LE, OSV2R8). TARGET(LE, 0x22060000, IMS, OSV2R8) resolves to TARGET(IMS, OSV2R8). TARGET(LE, 0x22060000, IMS) resolves to TARGET(IMS, 0x22060000).

The default value of the ARCHITECTURE compiler option depends on the value of the TARGET release suboption. For TARGET(OSV2R10 and above), the default is ARCH(2). For TARGET(OSV2R9 and below), the default is ARCH(0)

TARGET Release Suboptions

The TARGET release suboptions (CURRENT, OSV2R6, OSV2R7, OSV2R8, OSV2R9, OSV2R10, zOSV1R1 and zOSV1R2) allow you to generate code that can be executed on a particular release of an OS/390 or z/OS system, and on subsequent releases.

As of OS/390 V2R10, you can use the current Language Environment data sets during the assembly, compilation, pre-link, link-edit and bind phases. Prior to OS/390 V2R10, you had to use the Language Environment data sets for the targetted release. With the downward compatibility support provided by Language Environment if you target:

- OSV2R7 and above: The C/C++ compiler will issue a syntax error if the program that you are trying to compile contains functions that are not supported in your target release. The C/C++ headers provided by Language Environment, as of OS/390 V2R10, differentiate function provided in OS/390 V2R7 and higher. This support allows application developers to ″target″ a specific release, in order to ensure the application has not taken advantage of any new C/C++ library functions.
- OSV2R6: Although the system header files are not updated to differentiate functions provided in this range, toleration PTFs are available so that if an application attempts to exploit a Language Environment function that is unavailable on the release of OS/390 that the application is executed on, Language Environment will raise a new condition. However, with an unhandled condition, the application is terminated. In this case, instead of seeing an error during application development time, a condition will be produced at execution time when new functions are used.

For more information on the support available with Language Environment see the section on Downward Compatibility Considerations in the *z/OS Language Environment Programming Guide*.

In order to use these suboptions, you must:

- Use the z/OS V1R2 class library header files (found in the CBC.SCLBH.* data sets) during compilation
- If you are targetting OS/390 V2R9 or lower, use the class library data sets for the targetted release during pre-link, link-edit, and bind. For information on using previous releases of the class libraries refer to Informational APAR II11576 (available at http://www.ibm.com/software/ad/c390/service/ii11576.html)
- If you are targetting OS/390 V2R10, which is equivalent to z/OS V1R1 for the class libraries, you can use the z/OS V1R2 data sets for pre-link, link-edit and bind. The static version of the OS/390 V2R10 class library is in CBC.SCLBCPP and the sidedecks are in CBC.SCLBSID.

For example, to generate code that will execute on an OS/390 V2R6 system, using a z/OS V1R2 application development system:

- Use the z/OS V1R2 Language Environment data sets (CEE.SCEE*) during the assembly, compilation, pre-link, link-edit, and bind phases.
- Use the OS/390 V2R6 class library data sets (SCLBCPP, SCLBOBC, SCLBOXL, SCLBSID, SCLBXL) during pre-link, link-edit, and bind. Use the z/OS V1R2 class library header data sets (SCLBH.*) during compilation.
- Specify the compiler option TARGET(OSV2R6) 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 OS/390 V2R6.

See “Appendix A. Prelinking and Linking z/OS C/C++ Programs” on page 485 for details on prelinking and linking applications.

These compiler 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.

When you invoke the `TARGET(OSV`*x*`R`*y*`)` release suboptions, the compiler sets the `__TARGET_LIB__` and `__LIBREL__` macros. See the *C/C++ Language Reference* for more information about these macros.

Using the Hexadecimal String Literal Suboption: This hexadecimal literal string 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 to ensure 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 uses TARGET(OSV2R6). 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. Following are some examples:

TARGET(0x22060000)
: Equivalent to TARGET(OSV2R6).

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(0x22060001)
: This does not match any existing operating system release suboption name because of the 1 in the last digit. The next lower operating system level implied by this literal which the compiler considers valid is OSV2R6. Thus, the compiler sets the __TARGET_LIB__ macro to 0x22060001, and behaves as though you have specified TARGET(OSV2R6).

TARGET(0x21010000)
: This does not match any existing operating system release suboption name, and specifies a release earlier than the earliest supported release, OSV2R6. In this instance, the compiler sets the __TARGET_LIB__ macro to 0x21010000, and behaves as though you have specified TARGET(OSV2R6).

Restrictions for C/C++: All input libraries used during the application build process must be the appropriate level for the target release.

- As of OS/390 V2R10, 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 targetted 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
• CURRENT • zOSV1R2	None. All options and features are allowed.
• zOSV1R1 • OSV2R10	Standard C++ features that are not supported in the targetted run time are disabled. RTTI compiler option is not supported. ARCH(2) is the default.
• OSV2R9 • OSV2R8 • OSV2R7 • OSV2R6	All options and features except XPLINK and GOFF are allowed. ARCH(0) is the default.

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.

Restrictions for C: TARGET(OSV*x*R*y*) is not permitted in a #pragma target() directive.

If you specify TARGET(OSV*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.

Effect on IPA Compile Step: 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.

Effect on IPA Link Step: If you specify TARGET on the IPA Link step, it overrides the TARGET value that you specified for the IPA Compile step.

The IPA Link step accepts the release suboptions, for example, CURRENT or OSV2R6. 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

TARGET Run-time Environment Suboptions (LE,IMS)

The TARGET Run-time Environment suboption allows you to select a run-time environment of either Language Environment or IMS.

Effect on IPA Compile Step: 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.

Effect on IPA Link Step: If you specify TARGET on the IPA Link step, it has the following effects:

- 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.

TEMPINC | NOTEMPINC

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
	✔			

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
PDS: TEMPINC(TEMPINC) HFS Directory: TEMPINC(./tempinc)			TEMPINC (tempinc)			

CATEGORY: File Management

TEMPINC(*location*) places all template instantiation files into *location*, which may be a PDS or an HFS directory. 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 an HFS file, the default location is the HFS 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:

```
CXX HELLO (NOTEMPINC(/hello) TEMPINC

CXX HELLO (TEMPINC(/hello)
```

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:

```
CXX HELLO (NOTEMPINC(/hello) TEMPINC(/n1)  NOTEMPINC(/test) TEMPINC

CXX HELLO (TEMPINC(/test)
```

If you have large numbers of recursive templates, consider using FASTT. See “FASTTEMPINC | NOFASTTEMPINC” on page 114 for details.

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.

```
TEMPINC(xxx)
```

Effect on IPA Compile Step
The TEMPINC option has the same effect on the IPA Compile step as it does on a regular compilation.

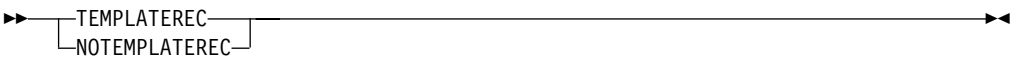
Effect on IPA Link Step
The IPA Link step issues a diagnostic message if you specify the TEMPINC option for that step.

TEMPLATERECOMPILE | NOTEMPLATERECOMPILE

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
TEMPLATERECOMPILE						

CATEGORY: File Management



The TEMPLATERECOMPILE option helps to manage dependencies between compilation units that have been compiled using the TEMPLATEREGISTRY option. Given a program in which multiple compilation units reference the same template instantiation, the TEMPLATEREGISTRY option nominates a single compilation unit to contain the instantiation. No other compilation units will contain this instantiation. Duplication of object code is thereby avoided. 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 have necessitated 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.

Effect on IPA Compile Step
The TEMPLATERECOMPILE option has the same effect on the IPA Compile step as it does on a regular compilation.

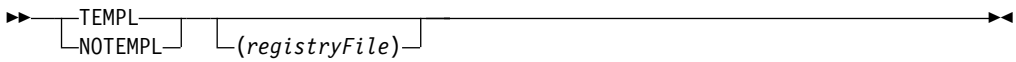
Effect on IPA Link Step
The IPA Link step accepts the TEMPLATERECOMPILE option, but ignores it.

TEMPLATEREGISTRY | NOTEMPLATEREGISTRY

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOTEMPLATEREGISTRY						

CATEGORY: File Management



The TEMPLATEREGISTRY option maintains records of all templates as they are encountered in the source and ensures that only one instantiation of each template is made. The first time that the compiler encounters a reference to a template instantiation, that 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 source resides in a data set, the default location is a sequential data set, whose high-level qualifier is the userid under which the compiler is running, with .TEMPLREG appended as the low-level qualifier. If the source resides in an HFS file, the default location is the HFS file ./templreg. For more information, see the *z/OS 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)
```

Effect on IPA Compile Step

The TEMPLATEREGISTRY option has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

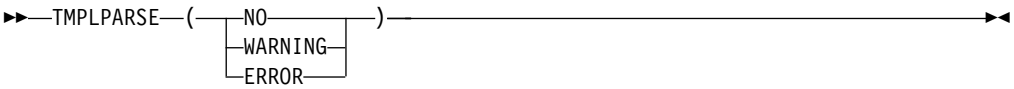
The IPA Link step issues a diagnostic message if you specify the TEMPLATEREGISTRY option for that step.

TMPLPARSE

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
	✓			

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
TMPLPARSE(NO)			TMPLPARSE(NO)			

CATEGORY: Programming Language Characteristics Control



The TMPLPARSE option controls whether parsing and semantic checking are applied to template definitions or only to template instantiations. The TMPLPARSE option applies to class templates definitions as well, for example, the class member list is skipped when the template is seen and is only parsed if the class template is instantiated.

The TMPLPARSE option supports the following suboptions:

no Do not parse the template implementations until they are instantiated. This is the default.

warning
Parses template implementations and issues warning messages for semantic errors.

error Treats problems in template implementations as errors, even if the template is not instantiated.

This option applies to template definitions but not their instantiations. Regardless of the setting of this option, error messages are produced for problems that appear outside definitions. For example, errors found during the parsing or semantic checking of constructs always cause error messages. Function template parameter lists must always be parsed so that the function can be identified. Errors in a function template parameter list always cause error messages.

Effect on IPA Compile Step

The TMPLPARSE compiler option has the same effect on the IPA Compile step as it does on a regular compilation.

Effect on IPA Link Step

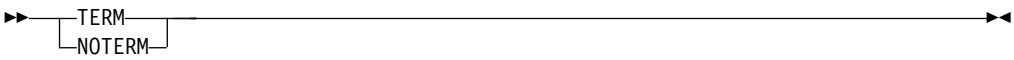
The IPA Link step issues a diagnostic message if you specify the TMPLPARSE option for that step.

TERMINAL | NOTERMINAL

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
TERMINAL	TERMINAL	TERMINAL	TERMINAL	TERMINAL	TERMINAL	TERMINAL

CATEGORY: Debug/Diagnostic



The TERMINAL option directs all of the diagnostic messages of the compiler to stderr.

If you specify NOTERMINAL, then no diagnostic messages are sent to stderr. Under z/OS batch, the default for stderr is SYSPRINT.

If you specify the PPONLY option, the compiler turns on TERM.

Effect on IPA

The TERMINAL option affects both the IPA Compile and the IPA Link steps in the same way that it affects a regular compilation.

TEST | NOTEST

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
Default for C++ compile: NOTEST(HOOK) Default for C compile: NOTEST(HOOK,SYM, BLOCK,LINE, PATH)	NOTEST-g sets TEST -s sets NOTEST	NOTEST-g sets TEST -s sets NOTEST	NOTEST-g sets TEST -s sets NOTEST	NOTEST	NOTEST	NOTEST

CATEGORY: Debug/Diagnostic

The TEST suboptions 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.

The TEST suboptions generate symbol tables and program hooks. The 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 the 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 the 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.

If you specify the NOTEST option, debugging information is not generated and you cannot trace your program with the Performance Analyzer.

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.

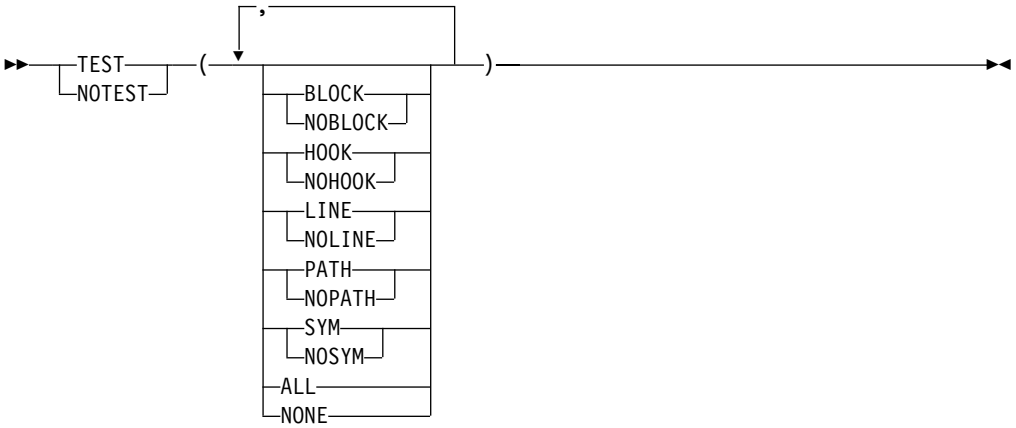
You can specify this option using the #pragma option directive for C.

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. For further information on the MFI Debug Tool, refer to the *Debug Tool User's Guide and Reference*, SC09-2137.

Additional z/OS C Compile Suboptions



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 the 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. The Debug Tool uses the symbol tables to obtain information about the variables in the program.

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 before a function call and after a function call.

- 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.
- The Debug Tool can gain control only at path points and block entry and exit points. If you attempt to STEP through your program, the 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).

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 the user has specified. The behavior of the TEST option in this case is as described in the table in the z/OS C/C++ section of the TEST | NOTEST option for the HOOK suboption.

For more information on debugging your program, see the *Debug Tool User's Guide and Reference*.

For z/OS 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.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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 NOTEST, the IPA Link step does not generate any debugging 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.

TUNE

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
TUNE(3) If the TUNE level is lower than the specified ARCH level, the compiler forces TUNE to match the ARCH level.						

CATEGORY: Object Code Control

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 86.

Specify 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:

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 below.

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 is the default. 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).

Note: For the above 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.

A comment that indicates the level of the TUNE option will be generated in your object module to aid you in diagnosing your program.

You can specify this option using the `#pragma option` directive for C.

Effect on IPA Compile Step

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.

Effect on IPA Link Step

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.

UNDEFINE

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
no action						

CATEGORY: Preprocessor

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 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.

Effect on IPA Compile Step
The UNDEFINE option is used for source code analysis, and has the same effect on the IPA Compile step that it does on a regular compilation.

Effect on IPA Link Step
The IPA Link step accepts the UNDEFINE option, but ignores it.

UPCONV | NOUPCONV

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓		✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOUPCONV	UPCONV	NOUPCONV				

CATEGORY: Programming Language Characteristics Control

The UPCONV option causes the z/OS C compiler to follow unsignedness preserving rules when doing z/OS C/C++ type conversions; that is, when widening all integral types (char, short, int, long). Use this option when compiling older z/OS C/C++ programs that depend on the older conversion rules.

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.

Effect on IPA Compile Step
The UPCONV option is used for source code analysis, and has the same effect on the IPA Compile step that it does on a regular compilation.

You can specify this option using the #pragma option directive for C.

Effect on IPA Link Step
The IPA Link step accepts UPCONV option, but ignores it.

WSIZEOF | NOWSIZEOF

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOWSIZEOF						

CATEGORY: Object Code Control

When you use the WSIZEOF option, sizeof returns the size of the widened type for function return types instead of the size of the original return type. For example, if you compile the following program with the WSIZEOF option, the value of i is 4.

```
char foo();
i = sizeof foo();
```

C/C++ compilers prior to and including C/C++ MVS/ESA Version 3 Release 1 returned the size of the widened type instead of the original type for function return types.

The z/OS C/C++ compiler now gives i the value 1, which is the size of the original type *char*.

If your source code depends on the behavior of the old compilers, use the WSIZEOF option to return the size of widened type for function return types.

The WSIZEOF option has exactly the same effect as putting a #pragma wsizeof(on) at the beginning of your source file. For more information on #pragma wsizeof(on), see *C/C++ Language Reference*.

You cannot specify the WSIZEOF option in a #pragma options directive.

Effect on IPA Compile Step

The WSIZEOF option has the same effect on the IPA Compile step that it does on a regular compilation.

Effect on IPA Link Step

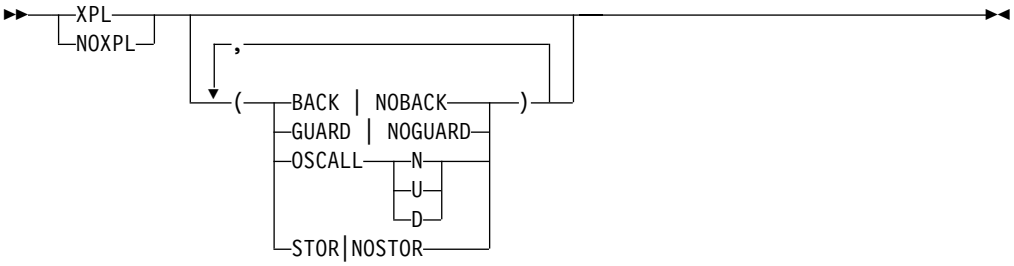
The IPA Link step accepts the WSIZEOF option, but ignores it.

XPLINK | NOXPLINK

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓		

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOXPLINK						

CATEGORY: Object Code Control, Debug/Diagnostic



The XPLINK (Extra Performance Linkage) option instructs the compiler to generate extra performance linkage for subroutine calls. Use this option to increase the performance of your applications.

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
- RENT

You may override these options. However, the XPLINK option requires the GOFF option. If you specify the NOGOFF option, the compiler issues an informational 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 an error message if you specify ARCH(0) or ARCH(1) with XPLINK.

The XPLINK option accepts the following suboptions:

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.

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 this suboption 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 (NOSTACK, UPSTACK, or DOWNSTACK) as specified in this suboption for any #pragma linkage(identifier, OS) calls in your application.

This value causes the compiler to use the following linkage wherever linkage OS is specified by #pragma linkage in C, or extern "linkage" in C++:

Linkage Used	Linkage Specified
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.

Effect on IPA Compile Step

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.

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.

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.

Effect on IPA Link Step

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.

XREF | NOXREF

Option Scope				
C Compile	C++ Compile	IPA Link	Special IPA Processing	
			IPA Compile	IPA Link
✓	✓	✓	✓	✓

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	Set by z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOXREF	NOXREF	NOXREF	NOXREF	NOXREF	NOXREF	NOXREF

CATEGORY: Listing

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.

For C, a separate offset listing of the variables will appear after the cross reference table.

You can specify this option using the #pragma option directive for C.

In the z/OS UNIX System Services environment, this option is turned on by specifying -V when using the c89, cc, or c++ commands.

Effect on IPA Compile Step

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 option (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.

Effect on IPA Link Step

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.

Using the z/OS 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 listings mentioned above. 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 274 for information about IPA Link step listings.

Example of a C Compiler Listing

Figure 17 shows an example of a C compiler listing.

```
5694A01 V1 R2 z/OS C                    'TSCTEST.ZOSV1R2.SCCNSAM(CCNUAAM)'                  05/14/2001 16:45:15   Page     1


                                  * * * * *  P R O L O G  * * * * *

 Compile Time Library . . . . . . : 41020000
 Command options:
    Program name. . . . . . . . . : 'TSCTEST.ZOSV1R2.SCCNSAM(CCNUAAM)'
    Compiler options. . . . . . . : *NOGONUMBER *NOALIAS    *NORENT     *TERMINAL   *NOUPCONV   *SOURCE     *LIST
                                  : *XREF       *AGGR       *NOPPONLY   *NOEXPMAC   *NOSHOWINC  *NOOFFSET   *MEMORY     *NOSSCOMM
                                  : *NOLONGNAME *START      *EXECOPS    *ARGPARSE   *NOEXPORTAL *NODLL(NOCALLBACKANY)
                                  : *NOLIBANSI  *NOWSIZEOF  *REDIR      *ANSIALIAS  *DIGRAPH    *NOROCONST  *ROSTRING
                                  : *TUNE(3)    *ARCH(2)    *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)
                                  : *FLOAT(HEX,FOLD,NOAFP)  *STRICT     *NOIGNERRNO *NOINITAUTO
                                  : *NOCOMPRESS *NOSTRICT_INDUCTION     *AGGRCOPY(NOOVERLAP)    *CHARS(UNSIGNED)
                                  : *NOCSECT
                                  : *NOEVENTS
                                  : *OBJECT
                                  : *NOOPTFILE
                                  : *NOSERVICE
                                  : *NOOE
                                  : *NOIPA
                                  : *SEARCH(//'CEE.SCEEH.+')
                                  : *NOLSEARCH
                                  : *NOLOCALE   *HALT(16)   *PLIST(HOST)
                                  : *NOCONVLIT
                                  : *NOASCII
                                  : *NOGOFF
                                  : *NOXPLINK(NOBACKCHAIN,NOSTOREARGS,GUARD,OSCALL(NOSTACK))
                                  : *ENUMSIZE(SMALL)
                                  : *NOHALTONMSG
                                  : *NOSUPPRESS
    Version Macros. . . . . . . . : __COMPILER_VER__=0x41020000 __LIBREL__=0x41020000 __TARGET_LIB__=0x41020000
    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 17. Example of a C listing (Part 1 of 31)

* * * * * 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       }                                                                                              33
   34                                                                                                      34
   35       void convert(double c_temp) {                                                                  35
   36    10    double f_temp = (c_temp * CONV + OFFSET);                                                   36
   37    11    printf("%5.2f Celsius is %5.2f Fahrenheit\n",c_temp, f_temp);                               37
   38       }                                                                                              38
                          * * * * *  E N D  O F  S O U R C E  * * * * *
```

Figure 17. Example of a C listing (Part 2 of 31)

```
5694A01 V1 R2 z/OS C                    'TSCTEST.ZOSV1R2.SCCNSAM(CCNUAAM)'                    05/14/2001 16:45:15   Page     3

                      * * * * *  I N C L U D E S  * * * * *

INCLUDE FILES  ---  FILE#   NAME

                       1   TSCTEST.CEEZ120.SCEEH.H(STDIO)
                       2   TSCTEST.CEEZ120.SCEEH.H(FEATURES)
                       3   TSCTEST.CEEZ120.SCEEH.SYS.H(TYPES)
                       4   TSCTEST.ZOSV1R2.SCCNSAM(CCNUAAN)

                      * * * * *  E N D  O F  I N C L U D E S  * * * * *
5694A01 V1 R2 z/OS C                    'TSCTEST.ZOSV1R2.SCCNSAM(CCNUAAM)'                    05/14/2001 16:45:15   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="-">

___valist           1-1:133         Class = typedef, Length = 8
                                    Type = array[2] of pointer to unsigned char
                                    1-1:136, 1-1:435, 1-1:436, 1-1:437

__abend             1-1:784         Type = struct with no tag in union at offset 0

__alloc             1-1:794         Type = struct with no tag in union at offset 0

__amrc_ptr          1-1:822         Class = typedef, Length = 4
                                    Type = pointer to struct __amrctype

__amrc_type         1-1:818         Class = typedef, Length = 224
                                    Type = struct __amrctype
                                    1-1:822

__amrctype          1-1:766         Class = struct tag

__amrc2_ptr         1-1:835         Class = typedef, Length = 4
                                    Type = pointer to struct __amrc2type

__amrc2_type        1-1:831         Class = typedef, Length = 32
                                    Type = struct __amrc2type
                                    1-1:835

__amrc2type         1-1:827         Class = struct tag

__blksize           1-1:669         Type = unsigned long in struct __fileData at offset 8

__bufPtr            1-1:70          Type = pointer to unsigned char in struct __file at offset 0

__cntlinterpret     1-1:75          Type = unsigned int:1 in struct __file at offset 20(0)

__code              1-1:795         Type = union with no tag in struct __amrctype at offset 0

__countIn           1-1:71          Type = long in struct __file at offset 4

__countOut          1-1:72          Type = long in struct __file at offset 8

__cusp              1-1:184         Class = typedef, Length = 4
                                    Type = pointer to const unsigned short

__device            1-1:668         Type = enum with no tag in struct __fileData at offset 4

__device_t          1-1:614         Class = typedef, Length = 1
                                    Type = enum with no tag
                                    1-1:668

__disk              1-1:599         Class = enumeration constant: 0, Length = 4
                                    Type = int

__dsname            1-1:674         Type = pointer to unsigned char in struct __fileData at offset 28
```

Figure 17. Example of a C listing (Part 3 of 31)

* * * * * 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="-">
__dsorgConcat	1-1:645	Type = unsigned int:1 in struct __fileData at offset 1(3)
__dsorgHiper	1-1:647	Type = unsigned int:1 in struct __fileData at offset 1(5)
__dsorgHFS	1-1:652	Type = unsigned int:1 in struct __fileData at offset 2(0)
__dsorgMem	1-1:646	Type = unsigned int:1 in struct __fileData at offset 1(4)
__dsorgPDSdir	1-1:643	Type = unsigned int:1 in struct __fileData at offset 1(1)
__dsorgPDSmem	1-1:642	Type = unsigned int:1 in struct __fileData at offset 1(0)
__dsorgPDSE	1-1:659	Type = unsigned int:1 in struct __fileData at offset 2(7)
__dsorgPO	1-1:641	Type = unsigned int:1 in struct __fileData at offset 0(7)
__dsorgPS	1-1:644	Type = unsigned int:1 in struct __fileData at offset 1(2)
__dsorgTemp	1-1:648	Type = unsigned int:1 in struct __fileData at offset 1(6)
__dsorgVSAM	1-1:649	Type = unsigned int:1 in struct __fileData at offset 1(7)
__dummy	1-1:604	Class = enumeration constant: 25769803776, Length = 4 Type = int
__error	1-1:778	Type = int in union at offset 0
__error2	1-1:828	Type = int in struct __amrc2type at offset 0
__fcb_ascii	1-1:76	Type = unsigned int:1 in struct __file at offset 20(1)
__fcbgetc	1-1:73	Type = pointer to function returning int in struct __file at offset 12
__fcbputc	1-1:74	Type = pointer to function returning int in struct __file at offset 16
__fdbk	1-1:789	Type = unsigned char in struct at offset 3
__fdbk_fill	1-1:786	Type = unsigned char in struct at offset 0
__feedback	1-1:790	Type = struct with no tag in union at offset 0
__ffile	1-1:79	Class = struct tag 1-1:84, 1-1:90
__file	1-1:65	Class = struct tag 1-1:66, 1-1:67, 1-1:81
__fileptr	1-1:829	Type = pointer to struct __ffile in struct __amrc2type at offset 4
__fileData	1-1:633	Class = struct tag 1-1:678
__fill	1-1:757	Type = unsigned int in struct at offset 0

Figure 17. Example of a C listing (Part 4 of 31)

* * * * * 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="-">
__filler1	1-1:586	Type = unsigned char in struct __S99emparms at offset 3
__fill2	1-1:809	Type = array[2] of unsigned int in struct at offset 136
__fp	1-1:81	Type = pointer to struct __file in struct __ffile at offset 0
__fpos_elem	1-1:95	Type = array[8] of long in struct __fpos_t at offset 0
__fpos_t	1-1:94	Class = struct tag 1-1:98
__ftncd	1-1:788	Type = unsigned char in struct at offset 2
__hfs	1-1:611	Class = enumeration constant: 38654705664, Length = 4 Type = int
__hiperspace	1-1:612	Class = enumeration constant: 42949672960, Length = 4 Type = int
__last_op	1-1:802	Type = unsigned int in struct __amrctype at offset 8
__len	1-1:805	Type = unsigned int in struct at offset 4
__len_fill	1-1:804	Type = unsigned int in struct at offset 0
__maxreclen	1-1:670	Type = unsigned long in struct __fileData at offset 12
__memory	1-1:610	Class = enumeration constant: 34359738368, Length = 4 Type = int
__modeflag	1-1:658	Type = unsigned int:4 in struct __fileData at offset 2(3)
__msg	1-1:812	Type = struct with no tag in struct __amrctype at offset 12
__msgfile	1-1:607	Class = enumeration constant: 30064771072, Length = 4 Type = int
__openmode	1-1:657	Type = unsigned int:2 in struct __fileData at offset 2(1)
__other	1-1:613	Class = enumeration constant: 1095216660480, Length = 4 Type = int
__parmr0	1-1:807	Type = unsigned int in struct at offset 128
__parmr1	1-1:808	Type = unsigned int in struct at offset 132
__printer	1-1:601	Class = enumeration constant: 8589934592, Length = 4 Type = int
__rc	1-1:783	Type = unsigned short in struct at offset 2
__rc	1-1:787	Type = unsigned char in struct at offset 1

Figure 17. Example of a C listing (Part 5 of 31)

* * * * * 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="-">

__recfmASA          1-1:639         Type = unsigned int:1 in struct __fileData at offset 0(5)

__recfmBlk          1-1:638         Type = unsigned int:1 in struct __fileData at offset 0(4)

__recfmF            1-1:634         Type = unsigned int:1 in struct __fileData at offset 0(0)

__recfmM            1-1:640         Type = unsigned int:1 in struct __fileData at offset 0(6)

__recfmS            1-1:637         Type = unsigned int:1 in struct __fileData at offset 0(3)

__recfmU            1-1:636         Type = unsigned int:1 in struct __fileData at offset 0(2)

__recfmV            1-1:635         Type = unsigned int:1 in struct __fileData at offset 0(1)

__recnum            1-1:758         Type = unsigned int in struct at offset 4

__reserved          1-1:569         Type = unsigned char in struct __S99rbx at offset 16

__reserved          1-1:830         Type = array[6] of int in struct __amrc2type at offset 8

__reserve2          1-1:663         Type = unsigned int:5 in struct __fileData at offset 3(3)

__reserve4          1-1:675         Type = unsigned int in struct __fileData at offset 32

__reserv1           1-1:591         Type = int in struct __S99emparms at offset 20

__reserv2           1-1:577         Type = int in struct __S99rbx at offset 32

__reserv2           1-1:592         Type = int in struct __S99emparms at offset 24

__rplfdbwd          1-1:815         Type = array[4] of unsigned char in struct __amrctype at offset 220

__rrds_key_type     1-1:759         Class = typedef, Length = 8
                                    Type = struct with no tag

__str               1-1:806         Type = array[120] of unsigned char in struct at offset 8

__str2              1-1:810         Type = array[64] of unsigned char in struct at offset 144

__svc99_error       1-1:793         Type = unsigned short in struct at offset 2

__svc99_info        1-1:792         Type = unsigned short in struct at offset 0

__syscode           1-1:782         Type = unsigned short in struct at offset 0

__tape              1-1:602         Class = enumeration constant: 12884901888, Length = 4
                                    Type = int

__tdq               1-1:603         Class = enumeration constant: 21474836480, Length = 4
                                    Type = int

__terminal          1-1:600         Class = enumeration constant: 4294967296, Length = 4
```

Figure 17. Example of a C listing (Part 6 of 31)

```
                          * * * * *  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="-">
                                    Type = int

__vsamkeylen        1-1:672         Type = unsigned long in struct __fileData at offset 20

__vsamtype          1-1:671         Type = unsigned short in struct __fileData at offset 16

__vsamRKP           1-1:673         Type = unsigned long in struct __fileData at offset 24

__vsamRLS           1-1:662         Type = unsigned int:3 in struct __fileData at offset 3(0)

__EMBUFP            1-1:590         Type = pointer to void in struct __S99emparms at offset 16

__EMCPPLP           1-1:589         Type = pointer to void in struct __S99emparms at offset 12

__EMFUNCT           1-1:583         Type = unsigned char in struct __S99emparms at offset 0

__EMIDNUM           1-1:584         Type = unsigned char in struct __S99emparms at offset 1

__EMNMSGBK          1-1:585         Type = unsigned char in struct __S99emparms at offset 2

__EMRETCOD          1-1:588         Type = int in struct __S99emparms at offset 8

__EMS99RBP          1-1:587         Type = pointer to void in struct __S99emparms at offset 4

__FILEP             1-1:84          Class = typedef, Length = 4
                                    Type = pointer to struct __ffile

__RBA               1-1:796         Type = unsigned int in struct __amrctype at offset 4

__S99emparms        1-1:582         Class = struct tag
                                    1-1:595

__S99emparms_t      1-1:595         Class = typedef, Length = 28
                                    Type = struct __S99emparms

__S99parms          1-1:555         Class = typedef, Length = 20
                                    Type = struct __S99struc
                                    1-1:727

__S99rbx            1-1:559         Class = struct tag
                                    1-1:580

__S99rbx_t          1-1:580         Class = typedef, Length = 36
                                    Type = struct __S99rbx

__S99struc          1-1:538         Class = struct tag
                                    1-1:555

__S99ECPPL          1-1:568         Type = pointer to void in struct __S99rbx at offset 12

__S99EERR           1-1:575         Type = unsigned short in struct __S99rbx at offset 28

__S99EID            1-1:561         Type = array[6] of unsigned char in struct __S99rbx at offset 0
```

Figure 17. Example of a C listing (Part 7 of 31)

```
                          * * * * *  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="-">

__S99EINFO          1-1:576         Type = unsigned short in struct __S99rbx at offset 30

__S99EKEY           1-1:565         Type = unsigned char in struct __S99rbx at offset 9

__S99EMGSV          1-1:566         Type = unsigned char in struct __S99rbx at offset 10

__S99EMSGP          1-1:574         Type = pointer to void in struct __S99rbx at offset 24

__S99ENMSG          1-1:567         Type = unsigned char in struct __S99rbx at offset 11

__S99EOPTS          1-1:563         Type = unsigned char in struct __S99rbx at offset 7

__S99ERCF           1-1:572         Type = unsigned char in struct __S99rbx at offset 19

__S99ERCO           1-1:571         Type = unsigned char in struct __S99rbx at offset 18

__S99ERES           1-1:570         Type = unsigned char in struct __S99rbx at offset 17

__S99ERROR          1-1:544         Type = unsigned short in struct __S99struc at offset 4

__S99ESUBP          1-1:564         Type = unsigned char in struct __S99rbx at offset 8

__S99EVER           1-1:562         Type = unsigned char in struct __S99rbx at offset 6

__S99EWRC           1-1:573         Type = int in struct __S99rbx at offset 20

__S99FLAG1          1-1:542         Type = unsigned short in struct __S99struc at offset 2

__S99FLAG2          1-1:550         Type = unsigned int in struct __S99struc at offset 16

__S99INFO           1-1:545         Type = unsigned short in struct __S99struc at offset 6

__S99RBLN           1-1:540         Type = unsigned char in struct __S99struc at offset 0

__S99S99X           1-1:548         Type = pointer to void in struct __S99struc at offset 12

__S99TXTPP          1-1:546         Type = pointer to void in struct __S99struc at offset 8

__S99VERB           1-1:541         Type = unsigned char in struct __S99struc at offset 1

_gtca                               Class = extern
                                    Type = function returning pointer to const void
                                    1-1:157

_Gtab                               Class = extern
                                    Type = function returning pointer to pointer to void
                                    1-1:147

_GETCFUNC           1-1:66          Class = typedef
                                    Type = function returning int
                                    1-1:73
```

Figure 17. Example of a C listing (Part 8 of 31)

```
                          * * * * *  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="-">
_PUTCFUNC           1-1:67          Class = typedef
                                    Type = function returning int
                                    1-1:74

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              35-0:35         Class = parameter, Length = 8
                                    Type = double in function convert
                                    36-0:36, 37-0:37

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
                                    1-1:389

clrmemf                             Class = extern
                                    Type = function returning int
                                    1-1:731

convert             35-0:35         Class = extern
                                    Type = function returning void
                                    5-0:5, 20-0:20, 30-0:30

f_temp              36-0:36         Class = auto, Length = 8
                                    Type = double in function convert
                                    36-0:36, 37-0:37

fclose                              Class = extern
                                    Type = function returning int
                                    1-1:390

fdelrec                             Class = extern
                                    Type = function returning int
                                    1-1:729

feof                                Class = extern
                                    Type = function returning int
                                    1-1:391

ferror                              Class = extern
                                    Type = function returning int
```

Figure 17. Example of a C listing (Part 9 of 31)

```
                          * * * * *  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="-">

                                    1-1:392

fflush                              Class = extern
                                    Type = function returning int
                                    1-1:393

fgetc                               Class = extern
                                    Type = function returning int
                                    1-1:394

fgetpos                             Class = extern
                                    Type = function returning int
                                    1-1:395

fgets                               Class = extern
                                    Type = function returning pointer to unsigned char
                                    1-1:396

fldata                              Class = extern
                                    Type = function returning int
                                    1-1:732

fldata_t            1-1:678         Class = typedef, Length = 36
                                    Type = struct __fileData
                                    1-1:732

flocate                             Class = extern
                                    Type = function returning int
                                    1-1:728

fopen                               Class = extern
                                    Type = function returning pointer to struct __ffile
                                    1-1:397

fpos_t              1-1:98          Class = typedef, Length = 32
                                    Type = struct __fpos_t
                                    1-1:395, 1-1:408

fprintf                             Class = extern
                                    Type = function returning int
                                    1-1:399

fputc                               Class = extern
                                    Type = function returning int
                                    1-1:400

fputs                               Class = extern
                                    Type = function returning int
                                    1-1:401

fread                               Class = extern
                                    Type = function returning unsigned int
                                    1-1:402
```

Figure 17. Example of a C listing (Part 10 of 31)

* * * * * 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="-">


freopen                             Class = extern
                                    Type = function returning pointer to struct __ffile
                                    1-1:404

fscanf                              Class = extern
                                    Type = function returning int
                                    1-1:406

fseek                               Class = extern
                                    Type = function returning int
                                    1-1:407

fsetpos                             Class = extern
                                    Type = function returning int
                                    1-1:408

ftell                               Class = extern
                                    Type = function returning long
                                    1-1:409

fupdate                             Class = extern
                                    Type = function returning unsigned int
                                    1-1:730

fwrite                              Class = extern
                                    Type = function returning unsigned int
                                    1-1:410

getc                                Class = extern
                                    Type = function returning int
                                    1-1:412

getchar                             Class = extern
                                    Type = function returning int
                                    1-1:413

gets                                Class = extern
                                    Type = function returning pointer to unsigned char
                                    1-1:414

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
                                    1-1:415

printf                              Class = extern
```

Figure 17. Example of a C listing (Part 11 of 31)

* * * * * 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="-">
                                    Type = function returning int
                                    1-1:416, 14-0:14, 17-0:17, 28-0:28, 37-0:37

putc                                Class = extern
                                    Type = function returning int
                                    1-1:417

putchar                             Class = extern
                                    Type = function returning int
                                    1-1:418

puts                                Class = extern
                                    Type = function returning int
                                    1-1:419

remove                              Class = extern
                                    Type = function returning int
                                    1-1:420

rename                              Class = extern
                                    Type = function returning int
                                    1-1:421

rewind                              Class = extern
                                    Type = function returning void
                                    1-1:422

scanf                               Class = extern
                                    Type = function returning int
                                    1-1:423, 16-0:16

setbuf                              Class = extern
                                    Type = function returning void
                                    1-1:424

setvbuf                             Class = extern
                                    Type = function returning int
                                    1-1:425

size_t              1-1:50          Class = typedef, Length = 4
                                    Type = unsigned int
                                    1-1:402, 1-1:402, 1-1:403, 1-1:410, 1-1:410, 1-1:410, 1-1:426, 1-1:728, 1-1:730, 1-1:730

sprintf                             Class = extern
                                    Type = function returning int
                                    1-1:428

sscanf                              Class = extern
                                    Type = function returning int
                                    1-1:430, 27-0:27

ssize_t             1-1:59          Class = typedef, Length = 4
                                    Type = int
```

Figure 17. Example of a C listing (Part 12 of 31)

```
                          * * * * *  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="-">

svc99                               Class = extern
                                    Type = function returning int
                                    1-1:727

tmpfile                             Class = extern
                                    Type = function returning pointer to struct __ffile
                                    1-1:432

tmpnam                              Class = extern
                                    Type = function returning pointer to unsigned char
                                    1-1:433

ungetc                              Class = extern
                                    Type = function returning int
                                    1-1:434

va_list             1-1:136         Class = typedef, Length = 8
                                    Type = array[2] of pointer to unsigned char

vfprintf                            Class = extern
                                    Type = function returning int
                                    1-1:435

vprintf                             Class = extern
                                    Type = function returning int
                                    1-1:436

vsprintf                            Class = extern
                                    Type = function returning int
                                    1-1:437

FILE                1-1:90          Class = typedef, Length = 4
                                    Type = struct __ffile
                                    1-1:389, 1-1:390, 1-1:391, 1-1:392, 1-1:393, 1-1:394, 1-1:395, 1-1:396, 1-1:397, 1-1:399,
                                    1-1:400, 1-1:401, 1-1:403, 1-1:404, 1-1:405, 1-1:406, 1-1:407, 1-1:408, 1-1:409, 1-1:411,
                                    1-1:412, 1-1:417, 1-1:422, 1-1:424, 1-1:425, 1-1:432, 1-1:434, 1-1:435, 1-1:728, 1-1:729,
                                    1-1:730, 1-1:732, 1-1:829

                      * * * * *  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 17. Example of a C listing (Part 13 of 31)

* * * * * S T R U C T U R E M A P S * * * * *

Aggregate map for: struct with no tag #1 — Total size: 8 bytes

__rrds_key_type

Offset Bytes(Bits)	Length Bytes(Bits)	Member Name
0	4	__fill
4	4	__recnum

Aggregate map for: _Packed struct with no tag #1 — Total size: 8 bytes

_Packed __rrds_key_type

Offset Bytes(Bits)	Length Bytes(Bits)	Member Name
0	4	__fill
4	4	__recnum

Aggregate map for: union with no tag #2 — Total size: 4 bytes

__code

Offset Bytes(Bits)	Length Bytes(Bits)	Member Name
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 #3 — Total size: 4 bytes

__abend

Offset Bytes(Bits)	Length Bytes(Bits)	Member Name
0	2	__syscode

Figure 17. Example of a C listing (Part 14 of 31)

```
                              * * * * *  S T R U C T U R E  M A P S  * * * * *
|       2           |       2           | __rc                                                                                                  |
=========================================================================================================================================================

=========================================================================================================================================================
| Aggregate map for: struct with no tag #4                                                                                  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 #5                                                                                  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 #6                                                                                  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: 224 bytes       |
|.......................................................................................................................................................|
|__amrc_type                                                                                                                                            |
|*__amrc_ptr                                                                                                                                            |
|=======================================================================================================================================================|
|      Offset       |      Length       | Member Name                                                                                               |
|    Bytes(Bits)    |    Bytes(Bits)    |                                                                                                           |
|===================|===================|===========================================================================================================|
|       0           |       4           | __code                                                                                                    |
```

Figure 17. Example of a C listing (Part 15 of 31)

```
5694A01 V1 R2 z/OS C                 'TSCTEST.ZOSV1R2.SCCNSAM(CCNUAAM)'              05/14/2001 16:45:15   Page    17

                                  * * * * *   S T R U C T U R E   M A P S   * * * * *
|       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]                                                                      |
|     156          |       64          |     __str2[64]                                                                      |
|     220          |        4          |   __rplfdbwd[4]                                                                     |
=====================================================================================================================================

=====================================================================================================================================
| Aggregate map for: _Packed struct __amrctype                                                         Total size: 224 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]                                                                      |
|     156          |       64          |     __str2[64]                                                                      |
```

Figure 17. Example of a C listing (Part 16 of 31)

```
                              * * * * *  S T R U C T U R E  M A P S  * * * * *
|     220          |       4          | __rplfdbwd[4]                                                                                                |
================================================================================================================================================

================================================================================================================================================
| 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 17. Example of a C listing (Part 17 of 31)

* * * * * S T R U C T U R E M A P S * * * * *

Aggregate map for: struct __file — Total size: 24 bytes

Offset Bytes(Bits)	Length Bytes(Bits)	Member Name
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)	3(6)	***PADDING***

Aggregate map for: _Packed struct __file — Total size: 21 bytes

Offset Bytes(Bits)	Length Bytes(Bits)	Member Name
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(6)	***PADDING***

Aggregate map for: struct __fileData — Total size: 36 bytes

fldata_t

Offset Bytes(Bits)	Length Bytes(Bits)	Member Name
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

Figure 17. Example of a C listing (Part 18 of 31)

```
                                  * * * * *   S T R U C T U R E   M A P S   * * * * *
|      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(5)         |  __reserve2                                                                          |
|      4             |       1            |  __device                                                                            |
|      5             |       3            |  ***PADDING***                                                                       |
|      8             |       4            |  __blksize                                                                           |
|     12             |       4            |  __maxreclen                                                                         |
|     16             |       2            |  __vsamtype                                                                          |
|     18             |       2            |  ***PADDING***                                                                       |
|     20             |       4            |  __vsamkeylen                                                                        |
|     24             |       4            |  __vsamRKP                                                                           |
|     28             |       4            |  __dsname                                                                            |
|     32             |       4            |  __reserve4                                                                          |
==================================================================================================================================

==================================================================================================================================
| Aggregate map for: _Packed struct __fileData                                                          Total size: 31 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                                                                          |
|      2(3)          |       0(4)         |  __modeflag                                                                          |
|      2(7)          |       0(1)         |  __dsorgPDSE                                                                         |
|      3             |       0(3)         |  __vsamRLS                                                                           |
|      3(3)          |       0(5)         |  __reserve2                                                                          |
|      4             |       1            |  __device                                                                            |
|      5             |       4            |  __blksize                                                                           |
|      9             |       4            |  __maxreclen                                                                         |
|     13             |       2            |  __vsamtype                                                                          |
|     15             |       4            |  __vsamkeylen                                                                        |
|     19             |       4            |  __vsamRKP                                                                           |
|     23             |       4            |  __dsname                                                                            |
```

Figure 17. Example of a C listing (Part 19 of 31)

```
                              * * * * *  S T R U C T U R E  M A P S  * * * * *
|      27          |       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                                                                                                 |
|       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                                                                                                |
```

Figure 17. Example of a C listing (Part 20 of 31)

```
                                  * * * * *   S T R U C T U R E   M A P S   * * * * *
|        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                                                                               |
===================================================================================================================================

===================================================================================================================================
| 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                                                                               |
```

Figure 17. Example of a C listing (Part 21 of 31)

```
                                  * * * * *   S T R U C T U R E   M A P S   * * * * *
|       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                                                                  |
|        2          |        2          |  __S99FLAG1                                                                 |
|        4          |        2          |  __S99ERROR                                                                 |
|        6          |        2          |  __S99INFO                                                                  |
|        8          |        4          |  __S99TXTPP                                                                 |
|       12          |        4          |  __S99S99X                                                                  |
|       16          |        4          |  __S99FLAG2                                                                 |
=====================================================================================================================
```

Figure 17. Example of a C listing (Part 22 of 31)

```
5694A01 V1 R2 z/OS C                    'TSCTEST.ZOSV1R2.SCCNSAM(CCNUAAM)'                    05/14/2001 16:45:15   Page    24

                          * * * * *  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  * * * * *
5694A01 V1 R2 z/OS C                    'TSCTEST.ZOSV1R2.SCCNSAM(CCNUAAM)'                    05/14/2001 16:45:15   Page    25

                   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      I       E      19               0/0       2/2        convert
    A      T,N     E      123   (77)       2/2       0/0        main

 Mode = AUTO    Inlining Threshold = 100    Expansion Limit = 1000

5694A01 V1 R2 z/OS C                    'TSCTEST.ZOSV1R2.SCCNSAM(CCNUAAM)'                    05/14/2001 16:45:15   Page    26

                   Inline Report (Call Structure)

 Defined Function      : convert
     Calls To          : 0
  Called From(2,2)     : main(2,2)

 Defined Function      : main
     Calls To(2,2)     : convert(2,2)
  Called From          : 0

5694A01 V1 R2 z/OS C                    'TSCTEST.ZOSV1R2.SCCNSAM(CCNUAAM)'                    05/14/2001 16:45:15   Page    27

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  F0F1                                    =C'2001'          Compiled Year
000004  F0F5  F1F4                                    =C'0514'          Compiled Date MMDD
000008  F1F6  F4F5  F1F5                              =C'164515'        Compiled Time HHMMSS
00000E  F0F1  F0F2  F0F0                              =C'010200'        Compiler Version
                         Timestamp and Version End
```

Figure 17. Example of a C listing (Part 23 of 31)

```
5694A01 V1 R2 z/OS C                    'TSCTEST.ZOSV1R2.SCCNSAM(CCNUAAM)': main             05/14/2001 16:45:15   Page    28

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

                         00001 |       *  #include <stdio.h>
                         00002 |       *
                         00003 |       *  #include "ccnuaan.h"
                         00004 |       *
                         00005 |       *  void convert(double);
                         00006 |       *
                         00007 |       *  int main(int argc, char **argv)
000018                   00007 |        main     DS    0D
000018  47F0  F022       00007 |                 B     34(,r15)
00001C  01C3C5C5                                 CEE eyecatcher
000020  000000D8                                 DSA size
000024  00000368                                 =A(PPA1-main)
000028  47F0  F001       00007 |                 B     1(,r15)
00002C  58F0  C31C       00007 |                 L     r15,796(,r12)
000030  184E             00007 |                 LR    r4,r14
000032  05EF             00007 |                 BALR  r14,r15
000034  00000000                                 =F'0'
000038  07F3             00007 |                 BR    r3
00003A  90E5  D00C       00007 |                 STM   r14,r5,12(r13)
00003E  58E0  D04C       00007 |                 L     r14,76(,r13)
000042  4100  E0D8       00007 |                 LA    r0,216(,r14)
000046  5500  C314       00007 |                 CL    r0,788(,r12)
00004A  4130  F03A       00007 |                 LA    r3,58(,r15)
00004E  4720  F014       00007 |                 BH    20(,r15)
000052  58F0  C280       00007 |                 L     r15,640(,r12)
000056  90F0  E048       00007 |                 STM   r15,r0,72(r14)
00005A  9210  E000       00007 |                 MVI   0(r14),16
00005E  50D0  E004       00007 |                 ST    r13,4(,r14)
000062  18DE             00007 |                 LR    r13,r14
000064                   End of Prolog

000064  5850  31C2       00007 |                 L     r5,=A(@CONSTANT_AREA)(,r3,450)
000068  5010  D0D0       00007 |                 ST    r1,#SR_PARM_1(,r13,208)
00006C  47F0  31AA       00007 |                 B     @1L2
000070                   00007 |        @1L1     DS    0H
                         00008 |       *  {
                         00009 |       *      double c_temp;
                         00010 |       *
                         00011 |       *      if (argc == 1) {  /* get Celsius value from stdin */
000070  5810  D0D0       00011 |                    L     r1,#SR_PARM_1(,r13,208)
000074  5800  1000       00011 |                    L     r0,argc(,r1,0)
000078  A70E  0001       00011 |                    CHI   r0,H'1'
00007C  4770  30C2       00011 |                    BNE   @1L3
                         00012 |       *            int ch;
                         00013 |       *
                         00014 |       *            printf("Enter Celsius temperature: \n");
000080  58F0  31C6       00014 |                    L     r15,=V(PRINTF)(,r3,454)
000084  1805             00014 |                    LR    r0,r5
000086  4110  D098       00014 |                    LA    r1,#MX_TEMP1(,r13,152)
00008A  5000  D098       00014 |                    ST    r0,#MX_TEMP1(,r13,152)
00008E  05EF             00014 |                    BALR  r14,r15
                         00015 |       *
                         00016 |       *            if (scanf("%f", &c_temp;) != 1) {
000090  4100  D0B0       00016 |                    LA    r0,c_temp(,r13,176)
000094  58F0  31CA       00016 |                    L     r15,=V(SCANF)(,r3,458)
000098  4120  501D       00016 |                    LA    r2,+CONSTANT_AREA(,r5,29)
```

Figure 17. Example of a C listing (Part 24 of 31)

```
5694A01 V1 R2 z/OS C                    'TSCTEST.ZOSV1R2.SCCNSAM(CCNUAAM)': main              05/14/2001 16:45:15   Page    29

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

00009C  4110  D098        00016 |                 LA    r1,#MX_TEMP1(,r13,152)
0000A0  5020  D098        00016 |                 ST    r2,#MX_TEMP1(,r13,152)
0000A4  5000  D09C        00016 |                 ST    r0,#MX_TEMP1(,r13,156)
0000A8  05EF              00016 |                 BALR  r14,r15
0000AA  180F              00016 |                 LR    r0,r15
0000AC  A70E  0001        00016 |                 CHI   r0,H'1'
0000B0  4780  3078        00016 |                 BE    @1L4
                          00017 |       *            printf("You must enter a valid temperature\n");
0000B4  58F0  31C6        00017 |                 L     r15,=V(PRINTF)(,r3,454)
0000B8  4100  5020        00017 |                 LA    r0,+CONSTANT_AREA(,r5,32)
0000BC  4110  D098        00017 |                 LA    r1,#MX_TEMP1(,r13,152)
0000C0  5000  D098        00017 |                 ST    r0,#MX_TEMP1(,r13,152)
0000C4  05EF              00017 |                 BALR  r14,r15
0000C6  47F0  30BE        00017 |                 B     @1L5
0000CA                    00017 |          @1L4   DS    0H
                          00018 |       *         }
                          00019 |       *         else {
                          00020 |       *            convert(c_temp);
0000CA  6800  D0B0        00020 |                 LD    f0,c_temp(,r13,176)
0000CE  6000  D0C0        00020 |                 STD   f0,c_temp:convert(,r13,192)
0000D2  47F0  30BA        00035 |       +         B     @1L13
0000D6                    00035 |       +@1L12   DS    0H
0000D6  6800  D0C0        00036 |       +         LD    f0,c_temp:convert(,r13,192)
0000DA  6820  5048        00036 |       +         LD    f2,+CONSTANT_AREA(,r5,72)
0000DE  2C02              00036 |       +         MDR   f0,f2
0000E0  6820  5050        00036 |       +         LD    f2,+CONSTANT_AREA(,r5,80)
0000E4  2A02              00036 |       +         ADR   f0,f2
0000E6  6000  D0C8        00036 |       +         STD   f0,f_temp:convert(,r13,200)
0000EA  6820  D0C0        00037 |       +         LD    f2,c_temp:convert(,r13,192)
0000EE  6020  D09C        00037 |       +         STD   f2,#MX_TEMP1(,r13,156)
0000F2  6000  D0A4        00037 |       +         STD   f0,#MX_TEMP1(,r13,164)
0000F6  58F0  31C6        00037 |       +         L     r15,=V(PRINTF)(,r3,454)
0000FA  4100  5058        00037 |       +         LA    r0,+CONSTANT_AREA(,r5,88)
0000FE  4110  D098        00037 |       +         LA    r1,#MX_TEMP1(,r13,152)
000102  5000  D098        00037 |       +         ST    r0,#MX_TEMP1(,r13,152)
000106  05EF              00037 |       +         BALR  r14,r15
000108  47F0  30BE        00038 |       +         B     @1L14
00010C                    00038 |       +@1L13   DS    0H
00010C  47F0  3084        00038 |       +         B     @1L12
000110                    00038 |       +@1L14   DS    0H
000110                    00038 |       +@1L5    DS    0H
000110  47F0  31A2        00038 |       +         B     @1L6
000114                    00038 |       +@1L3    DS    0H
                          00021 |       *         }
                          00022 |       *      }
                          00023 |       *      else {  /* convert the command-line arguments to Fahrenheit */
                          00024 |       *         int i;
                          00025 |       *
                          00026 |       *         for (i = 1; i < argc; ++i) {
000114  4100  0001        00026 |                 LA    r0,1
000118  5000  D0B8        00026 |                 ST    r0,i(,r13,184)
00011C  5810  D0D0        00026 |                 L     r1,#SR_PARM_1(,r13,208)
000120  5810  1000        00026 |                 L     r1,argc(,r1,0)
000124  1901              00026 |                 CR    r0,r1
000126  47B0  31A2        00026 |                 BNL   @1L8
00012A                    00026 |          @1L7   DS    0H
```

Figure 17. Example of a C listing (Part 25 of 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

                         00027 |       *             if (sscanf(argv[i], "%f", &c_temp;) != 1)
00012A  5810  D0D0       00027 |                 L     r1,#SR_PARM_1(,r13,208)
00012E  5810  1004       00027 |                 L     r1,argv(,r1,4)
000132  5820  D0B8       00027 |                 L     r2,i(,r13,184)
000136  8920  0002       00027 |                 SLL   r2,2
00013A  5822  1000       00027 |                 L     r2,(*)uchar*(r2,r1,0)
00013E  4100  D0B0       00027 |                 LA    r0,c_temp(,r13,176)
000142  58F0  31CE       00027 |                 L     r15,=V(SSCANF)(,r3,462)
000146  4110  D098       00027 |                 LA    r1,#MX_TEMP1(,r13,152)
00014A  5020  D098       00027 |                 ST    r2,#MX_TEMP1(,r13,152)
00014E  4120  501D       00027 |                 LA    r2,+CONSTANT_AREA(,r5,29)
000152  5020  D09C       00027 |                 ST    r2,#MX_TEMP1(,r13,156)
000156  5000  D0A0       00027 |                 ST    r0,#MX_TEMP1(,r13,160)
00015A  05EF             00027 |                 BALR  r14,r15
00015C  180F             00027 |                 LR    r0,r15
00015E  A70E  0001       00027 |                 CHI   r0,H'1'
000162  4780  3142       00027 |                 BE    @1L9
                         00028 |       *                  printf("%s is not a valid temperature\n",argv[i]);
000166  5810  D0D0       00028 |                 L     r1,#SR_PARM_1(,r13,208)
00016A  5810  1004       00028 |                 L     r1,argv(,r1,4)
00016E  5820  D0B8       00028 |                 L     r2,i(,r13,184)
000172  8920  0002       00028 |                 SLL   r2,2
000176  5802  1000       00028 |                 L     r0,(*)uchar*(r2,r1,0)
00017A  58F0  31C6       00028 |                 L     r15,=V(PRINTF)(,r3,454)
00017E  4120  507B       00028 |                 LA    r2,+CONSTANT_AREA(,r5,123)
000182  4110  D098       00028 |                 LA    r1,#MX_TEMP1(,r13,152)
000186  5020  D098       00028 |                 ST    r2,#MX_TEMP1(,r13,152)
00018A  5000  D09C       00028 |                 ST    r0,#MX_TEMP1(,r13,156)
00018E  05EF             00028 |                 BALR  r14,r15
000190  47F0  3188       00028 |                 B     @1L10
000194                   00028 |          @1L9   DS    0H
                         00029 |       *             else
                         00030 |       *                convert(c_temp);
000194  6800  D0B0       00030 |                 LD    f0,c_temp(,r13,176)
000198  6000  D0C0       00030 |                 STD   f0,c_temp:convert(,r13,192)
00019C  47F0  3184       00035 |       +         B     @1L16
0001A0                   00035 |       +@1L15   DS    0H
0001A0  6800  D0C0       00036 |       +         LD    f0,c_temp:convert(,r13,192)
0001A4  6820  5048       00036 |       +         LD    f2,+CONSTANT_AREA(,r5,72)
0001A8  2C02             00036 |       +         MDR   f0,f2
0001AA  6820  5050       00036 |       +         LD    f2,+CONSTANT_AREA(,r5,80)
0001AE  2A02             00036 |       +         ADR   f0,f2
0001B0  6000  D0C8       00036 |       +         STD   f0,f_temp:convert(,r13,200)
0001B4  6820  D0C0       00037 |       +         LD    f2,c_temp:convert(,r13,192)
0001B8  6020  D09C       00037 |       +         STD   f2,#MX_TEMP1(,r13,156)
0001BC  6000  D0A4       00037 |       +         STD   f0,#MX_TEMP1(,r13,164)
0001C0  58F0  31C6       00037 |       +         L     r15,=V(PRINTF)(,r3,454)
0001C4  4100  5058       00037 |       +         LA    r0,+CONSTANT_AREA(,r5,88)
0001C8  4110  D098       00037 |       +         LA    r1,#MX_TEMP1(,r13,152)
0001CC  5000  D098       00037 |       +         ST    r0,#MX_TEMP1(,r13,152)
0001D0  05EF             00037 |       +         BALR  r14,r15
0001D2  47F0  3188       00038 |       +         B     @1L17
0001D6                   00038 |       +@1L16   DS    0H
0001D6  47F0  314E       00038 |       +         B     @1L15
0001DA                   00038 |       +@1L17   DS    0H
0001DA                   00038 |       +@1L10   DS    0H
```

Figure 17. Example of a C listing (Part 26 of 31)

```
5694A01 V1 R2 z/OS C                    'TSCTEST.ZOSV1R2.SCCNSAM(CCNUAAM)': main             05/14/2001 16:45:15   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

0001DA  5800  D0B8        00038 |       +         L     r0,i(,r13,184)
0001DE  A70A  0001        00038 |       +         AHI   r0,H'1'
0001E2  5000  D0B8        00038 |       +         ST    r0,i(,r13,184)
0001E6  5810  D0D0        00038 |       +         L     r1,#SR_PARM_1(,r13,208)
0001EA  5810  1000        00038 |       +         L     r1,argc(,r1,0)
0001EE  1901              00038 |       +         CR    r0,r1
0001F0  4740  30D8        00038 |       +         BL    @1L7
0001F4                    00038 |       +@1L11   DS    0H
0001F4                    00038 |       +@1L8    DS    0H
0001F4                    00038 |       +@1L6    DS    0H
                          00031 |       *         }
                          00032 |       *      }
                          00033 |       *  }
0001F4  41F0  0000        00033 |                 LA    r15,0
0001F8  47F0  31AE        00033 |                 B     @1L21
0001FC                    00033 |        @1L2     DS    0H
0001FC  47F0  301E        00033 |                 B     @1L1
000200                    00033 |        @1L21    DS    0H

000200                    Start of Epilog
000200  180D              00033 |                 LR    r0,r13
000202  58D0  D004        00033 |                 L     r13,4(,r13)
000206  58E0  D00C        00033 |                 L     r14,12(,r13)
00020A  9825  D01C        00033 |                 LM    r2,r5,28(r13)
00020E  051E              00033 |                 BALR  r1,r14
000210  0707              00033 |                 NOPR  7
000212  0000

000214                    Start of Literals
000214  000002E0                                  =A(@CONSTANT_AREA)
000218  00000000                                  =V(PRINTF)
00021C  00000000                                  =V(SCANF)
000220  00000000                                  =V(SSCANF)
000224                    End of Literals

                          ***   General purpose registers used: 1111110000001111
                          ***   Floating point  registers used: 1010101000000000
                          ***   Size of register spill area: 128(max) 0(used)
                          ***   Size of dynamic storage: 216
                          ***   Size of executable code: 506
```

Figure 17. Example of a C listing (Part 27 of 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

                         00001 |       *  #include <stdio.h>
                         00002 |       *
                         00003 |       *  #include "ccnuaan.h"
                         00004 |       *
                         00005 |       *  void convert(double);
                         00006 |       *
                         00007 |       *  int main(int argc, char **argv)
                         00008 |       *  {
                         00009 |       *      double c_temp;
                         00010 |       *
                         00011 |       *      if (argc == 1) {  /* get Celsius value from stdin */
                         00012 |       *         int ch;
                         00013 |       *
                         00014 |       *         printf("Enter Celsius temperature: \n");
                         00015 |       *
                         00016 |       *         if (scanf("%f", &c_temp;) != 1) {
                         00017 |       *            printf("You must enter a valid temperature\n");
                         00018 |       *         }
                         00019 |       *         else {
                         00020 |       *            convert(c_temp);
                         00021 |       *         }
                         00022 |       *      }
                         00023 |       *      else {  /* convert the command-line arguments to Fahrenheit */
                         00024 |       *         int i;
                         00025 |       *
                         00026 |       *         for (i = 1; i < argc; ++i) {
                         00027 |       *             if (sscanf(argv[i], "%f", &c_temp;) != 1)
                         00028 |       *                printf("%s is not a valid temperature\n",argv[i]);
                         00029 |       *             else
                         00030 |       *                convert(c_temp);
                         00031 |       *         }
                         00032 |       *      }
                         00033 |       *  }
                         00034 |       *
                         00035 |       *  void convert(double c_temp) {
000228                   00035 |        convert  DS    0D
000228  47F0  F022       00035 |                 B     34(,r15)
00022C  01C3C5C5                                 CEE eyecatcher
000230  000000C0                                 DSA size
000234  00000198                                 =A(PPA1-convert)
000238  47F0  F001       00035 |                 B     1(,r15)
00023C  58F0  C31C       00035 |                 L     r15,796(,r12)
000240  184E             00035 |                 LR    r4,r14
000242  05EF             00035 |                 BALR  r14,r15
000244  00000000                                 =F'0'
000248  07F3             00035 |                 BR    r3
00024A  90E5  D00C       00035 |                 STM   r14,r5,12(r13)
00024E  58E0  D04C       00035 |                 L     r14,76(,r13)
000252  4100  E0C0       00035 |                 LA    r0,192(,r14)
000256  5500  C314       00035 |                 CL    r0,788(,r12)
00025A  4130  F03A       00035 |                 LA    r3,58(,r15)
00025E  4720  F014       00035 |                 BH    20(,r15)
000262  58F0  C280       00035 |                 L     r15,640(,r12)
000266  90F0  E048       00035 |                 STM   r15,r0,72(r14)
00026A  9210  E000       00035 |                 MVI   0(r14),16
00026E  50D0  E004       00035 |                 ST    r13,4(,r14)
```

Figure 17. Example of a C listing (Part 28 of 31)

```
5694A01 V1 R2 z/OS C                    'TSCTEST.ZOSV1R2.SCCNSAM(CCNUAAM)': convert          05/14/2001 16:45:15   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

000272  18DE              00035 |                LR    r13,r14
000274                    End of Prolog

000274  5850  3072        00035 |                L     r5,=A(@CONSTANT_AREA)(,r3,114)
000278  5010  D0B8        00035 |                ST    r1,#SR_PARM_2(,r13,184)
00027C  47F0  305C        00035 |                B     @2L20
000280                    00035 |         @2L19  DS    0H
                          00036 |        *     double f_temp = (c_temp * CONV + OFFSET);
000280  5810  D0B8        00036 |                L     r1,#SR_PARM_2(,r13,184)
000284  6800  1000        00036 |                LD    f0,c_temp(,r1,0)
000288  6820  5048        00036 |                LD    f2,+CONSTANT_AREA(,r5,72)
00028C  2C02              00036 |                MDR   f0,f2
00028E  6820  5050        00036 |                LD    f2,+CONSTANT_AREA(,r5,80)
000292  2A02              00036 |                ADR   f0,f2
000294  6000  D0B0        00036 |                STD   f0,f_temp(,r13,176)
                          00037 |        *     printf("%5.2f Celsius is %5.2f Fahrenheit\n",c_temp, f_temp);
000298  5810  D0B8        00037 |                L     r1,#SR_PARM_2(,r13,184)
00029C  6820  1000        00037 |                LD    f2,c_temp(,r1,0)
0002A0  6020  D09C        00037 |                STD   f2,#MX_TEMP2(,r13,156)
0002A4  6000  D0A4        00037 |                STD   f0,#MX_TEMP2(,r13,164)
0002A8  58F0  3076        00037 |                L     r15,=V(PRINTF)(,r3,118)
0002AC  4100  5058        00037 |                LA    r0,+CONSTANT_AREA(,r5,88)
0002B0  4110  D098        00037 |                LA    r1,#MX_TEMP2(,r13,152)
0002B4  5000  D098        00037 |                ST    r0,#MX_TEMP2(,r13,152)
0002B8  05EF              00037 |                BALR  r14,r15
                          00038 |        *  }
0002BA  47F0  3060        00038 |                B     @2L22
0002BE                    00038 |         @2L20  DS    0H
0002BE  47F0  301E        00038 |                B     @2L19
0002C2                    00038 |         @2L22  DS    0H

0002C2                    Start of Epilog
0002C2  58D0  D004        00038 |                L     r13,4(,r13)
0002C6  58E0  D00C        00038 |                L     r14,12(,r13)
0002CA  9825  D01C        00038 |                LM    r2,r5,28(r13)
0002CE  051E              00038 |                BALR  r1,r14
0002D0  0707              00038 |                NOPR  7
0002D2  0000

0002D4                    Start of Literals
0002D4  000002E0                                 =A(@CONSTANT_AREA)
0002D8  00000000                                 =V(PRINTF)
0002DC                    End of Literals

                          ***   General purpose registers used: 1101110000001111
                          ***   Floating point  registers used: 1010101000000000
                          ***   Size of register spill area: 128(max) 0(used)
                          ***   Size of dynamic storage: 192
                          ***   Size of executable code: 170

0002DC  0000  0000

                          Constant Area
0002E0  C595A385 9940C385 93A289A4 A240A385    |Enter Celsius te|
0002F0  94978599 81A3A499 857A4015 006C8600    |mperature: ..%f.|
000300  E896A440 94A4A2A3 408595A3 85994081    |You must enter a|
```

Figure 17. Example of a C listing (Part 29 of 31)

```
5694A01 V1 R2 z/OS C                    'TSCTEST.ZOSV1R2.SCCNSAM(CCNUAAM)': convert          05/14/2001 16:45:15   Page    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

000310  40A58193 898440A3 85949785 9981A3A4    | valid temperatu|
000320  99851500 C9C2D440 411CCCCC CCCCCCCC    |re..IBM ........|
000330  42200000 00000000 6CF54BF2 8640C385    |........%5.2f Ce|
000340  93A289A4 A24089A2 406CF54B F28640C6    |lsius is %5.2f F|
000350  81889985 95888589 A315006C A24089A2    |ahrenheit..%s is|
000360  409596A3 408140A5 81938984 40A38594    | not a valid tem|
000370  97859981 A3A49985 1500                |perature..      |
5694A01 V1 R2 z/OS C                    'TSCTEST.ZOSV1R2.SCCNSAM(CCNUAAM)'                  05/14/2001 16:45:15   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

                          PPA1: Entry Point Constants
000380  1CCEA106                                    =F'483303686'      Flags
000384  000003F0                                    =A(PPA2-main)
000388  00000000                                    =F'0'              No PPA3
00038C  00000000                                    =F'0'              No EPD
000390  FF000000                                    =F'-16777216'      Register save mask
000394  00000000                                    =F'0'              Member flags
000398  90                                          =AL1(144)          Flags
000399  000000                                      =AL3(0)            Callee's DSA use/8
00039C  0040                                        =H'64'             Flags
00039E  0012                                        =H'18'             Offset/2 to CDL
0003A0  00000000                                    =F'0'              Reserved
0003A4  500000FD                                    =F'1342177533'     CDL function length/2
0003A8  FFFFFC98                                    =F'-872'           CDL function EP offset
0003AC  38260000                                    =F'942014464'      CDL prolog
0003B0  400900F4                                    =F'1074331892'     CDL epilog
0003B4  00000000                                    =F'0'              CDL end
0003B8  0004  ****                                  AL2(4),C'main'
                          PPA1 End

                          PPA1: Entry Point Constants
0003C0  1CCEA106                                    =F'483303686'      Flags
0003C4  000001E0                                    =A(PPA2-convert)
0003C8  00000000                                    =F'0'              No PPA3
0003CC  00000000                                    =F'0'              No EPD
0003D0  FF000000                                    =F'-16777216'      Register save mask
0003D4  00000000                                    =F'0'              Member flags
0003D8  90                                          =AL1(144)          Flags
0003D9  000000                                      =AL3(0)            Callee's DSA use/8
0003DC  0040                                        =H'64'             Flags
0003DE  0012                                        =H'18'             Offset/2 to CDL
0003E0  00000000                                    =F'0'              Reserved
0003E4  50000055                                    =F'1342177365'     CDL function length/2
0003E8  FFFFFE68                                    =F'-408'           CDL function EP offset
0003EC  38260000                                    =F'942014464'      CDL prolog
0003F0  4008004D                                    =F'1074266189'     CDL epilog
0003F4  00000000                                    =F'0'              CDL end
0003F8  0007  ****                                  AL2(7),C'convert'
                          PPA1 End

                          PPA2: Compile Unit Block
000408  0300  2202                                  =F'50340354'       Flags
00040C  FFFF  FBF8                                  =A(CEESTART-PPA2)
000410  0000  0000                                  =F'0'              No PPA4
000414  FFFF  FBF8                                  =A(TIMESTMP-PPA2)
000418  0000  0000                                  =F'0'              No primary
00041C  0000  0000                                  =F'0'              Flags
                          PPA2 End
```

Figure 17. Example of a C listing (Part 30 of 31)

```
5694A01 V1 R2 z/OS C                    'TSCTEST.ZOSV1R2.SCCNSAM(CCNUAAM)'                  05/14/2001 16:45:15   Page    36

                              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   000420           MAIN         LD    0 000018   000001
          CONVERT      LD    0 000228   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
5694A01 V1 R2 z/OS C                    'TSCTEST.ZOSV1R2.SCCNSAM(CCNUAAM)'                  05/14/2001 16:45:15   Page    37

                           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

          main                                          MAIN
          convert                                       CONVERT
          CEESG003                                      CEESG003
          printf                                        PRINTF
          scanf                                         SCANF
          sscanf                                        SSCANF
          CEESTART                                      CEESTART
          CEEMAIN                                       CEEMAIN
          EDCINPL                                       EDCINPL
5694A01 V1 R2 z/OS C                    'TSCTEST.ZOSV1R2.SCCNSAM(CCNUAAM)'                  05/14/2001 16:45:15   Page    38

                               * * * * *   S T O R A G E   O F F S E T   L I S T I N G   * * * * *

IDENTIFIER          DEFINITION      ATTRIBUTES
                                    <SEQNBR>-<FILE NO="-">

argc                7-0:7           Class = parameter,          Location = 0(r13),                          Length = 4

argv                7-0:7           Class = parameter,          Location = 4(r13),                          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              35-0:35         Class = parameter,          Location = 0(r13),                          Length = 8

f_temp              36-0:36         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 17. Example of a C listing (Part 31 of 31)

z/OS C Compiler Listing Components

The following sections describe 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 274 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 HFS 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 HFS file that contains the source code is greater than 32 characters, it is truncated. Only the rightmost 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 sections describe 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 *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 115, and *z/OS 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. 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.
  - The subprogram is directly recursive, or some calls have mismatching parameters.

  Note: As of OS/390 V2R10 C/C++, ″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.)
    - 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 option `LIST` 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 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.

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 listings mentioned above. 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 274 for information about IPA Link step listings.

Example of a C++ Compiler Listing

Figure 18 on page 257 shows an example of a z/OS C++ compiler listing. Vertical ellipses indicate sections that have been truncated.

```
15694A01 V1 R2 z/OS C++               'TSCTEST.SAMPLE.SRC(CCNUBRC)'                    05/22/01 20:15:00
                                    * * * * *   P R O L O G   * * * * *
   Compiler options. . . . . . . :AGGRCOPY(NOOVERLAP)             NOASCII         ANSIALIAS       ARGPARSE
                                 :NOCOMPACT       NOCSECT         CVFT            NOCOMPRESS      DIGRAPH
                                 :NOEVENTS        EXECOPS         EXH             NOEXPORTALL     NOEXPMAC
                                 :NOGOFF          NOGONUMBER      NOIGNERRNO      INLRPT          NOLIBANSI
                                 :LONGNAME        LONGLONG        LIST            NOMARGINS       MEMORY
                                 :NESTINC(255)    OBJECT          NOOE()          NOOFFSET        NOPORT
                                 :NOPPONLY        REDIR           ROSTRING        NOROCONST       NOSTATICINLINE
                                 :NOSEQUENCE      NOSHOWINC       START           STRICT          NOSTRICT_INDUCTION
                                 :SOURCE          NOTEST(HOOK)    TERMINAL        NOFASTTEMPINC   NOWSIZEOF
                                 :XREF            NOINITAUTO      PLIST(HOST)     TMPLPARSE(NO)   FLAG(I)
                                 :DLL(NOCALLBACKANY)              ARCH(2)         ENUM(SMALL)     TUNE(3)
                                 :OPTIMIZE(2)     HALT(16)        MAXMEM(2097152) SPILL(128)
                                 :INFO(LAN)       NOATTRIBUTE     NORTTI
                                 :OBJECTMODEL(COMPAT)             TARGET(LE,CURRENT)
                                 :NOCONVLIT
                                 :NOIPA
                                 :TEMPINC(TEMPINC)
                                 :INLINE(AUTO,REPORT,100,1000)
                                 :NOSERVICE
                                 :BITFIELD(UNSIGNED)
                                 :CHARS(UNSIGNED)
                                 :NOCHECK
                                 :LANGLVL(ANONSTRUCT,ANSIFOR,NOEMPTYSTRUCT,ILLPTOM,DBCS,TRAILENUM)
                                 :LANGLVL(IMPLICITINT,NONEWEXCP,LIBEXT,OFFSETNONPOD,NOOLDDIGRAPH,OLDFRIEND,OLDTEMPACC)
                                 :LANGLVL(NOOLDMATH,NOOLDTMPLALIGN,OLDTMPLSPEC,TYPEDEFCLASS,NOUCS,ANONYMOUSUNIONS,ZEROEXTARRAY)
                                 :NOXPLINK(NOBACKCHAIN,NOSTOREARGS,GUARD,OSCALL(NOSTACK))
                                 :FLOAT(HEX,FOLD,NOAFP) ROUND(Z)
                                 :LSEARCH()
                                 :SEARCH(//'CEE.SCEEH.+',//'CBC.SCLBH.+',//'TSCTEST.CEEZ120.SCEEH.+',//'TSCTEST.ZOSV1R2.SCLBH.+')
   Version Macros. . . . . . . . :__COMPILER_VER__=0x41020000 __LIBREL__=0x41020000 __TARGET_LIB__=0x41020000
   Locale name . . . . . . . . . :POSIX
   Code set. . . . . . . . . . . :IBM-1047
   Listing name. . . . . . . . . :DD:SYSCPRT
15694A01 V1 R2 z/OS C++               'TSCTEST.SAMPLE.SRC(CCNUBRC)'                    05/22/01 20:15:00

                                    * * * * *   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 |using namespace std;
     16 |
     17 |#include "ccnubrh.h"  //BioRhythm class and Date class
     18 |
     19 |int main(void) {
     20 |  BioRhythm bio;
     21 |  int code;
     22 |
     23 |  if (!bio.ok()) {
     24 |    cerr << "Error in birthdate specification - format is yyyy/mm/dd";
     25 |    code = 8;
     26 |  }
     27 |  else {
     28 |    cout << bio;  // write out birthdate for bio
     29 |    code = 0;
     30 |  }
     31 |  return(code);
     32 |}
     33 |
     34 |ostream& operator<<(ostream& os, BioRhythm& bio) {
     35 |  os << "Total Days  : " << bio.AgeInDays() << "\n";
     36 |  os << "Physical    : " << bio.Physical() << "\n";
     37 |  os << "Emotional   : " << bio.Emotional() << "\n";
     38 |  os << "Intellectual: " << bio.Intellectual() << "\n";
     39 |
```

Figure 18. Example of a C++ Compiler Listing (Part 1 of 14)

```
40 |return(os);
41 |}
42 |
43 |Date::Date() {
44 |  time_t lTime;
45 |  struct tm *newTime;
46 |
47 |  time(&lTime;);
48 |  newTime = localtime(&lTime;);
49 |  cout << "local time is " << asctime(newTime) << endl;
50 |
51 |  curYear = newTime->tm_year + 1900;
52 |  curDay  = newTime->tm_yday + 1;
53 |}
54 |
55 |BirthDate::BirthDate(const char *birthText) {
56 |  strcpy(text, birthText);
57 |}
58 |
59 |BirthDate::BirthDate() {
60 |  cout << "Please enter your birthdate in the form yyyy/mm/dd\n";
61 |  cin >> setw(dateLen+1) >> text;
62 |}
63 |
64 |Date::DaysSince(const char *text) {
65 |
66 |  int year, month, day, totDays, delim;
67 |  int daysInYear = 0;
68 |  int i;
69 |  int leap = 0;
70 |
71 |  int rc = sscanf(text, "%4d%c%2d%c%2d",
72 |                  &year;, &delim;, &month;, &delim;, &day;);
73 |  --month;
74 |  if (rc != 5 || year  < 0 || year  > 9999 ||
75 |                 month < 0 || month >   11 ||
76 |                 day   < 1 || day   >   31 ||
77 |                (day   > numDays[month]&& month != 1)) {
78 |    return(-1);
79 |  }
80 |  if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
81 |     leap = 1;
82 |
83 |  if (month == 1 && day > numDays[month]) {
84 |    if (day > 29)
85 |      return(-1);
86 |    else if (!leap)
87 |      return (-1);
88 |  }
89 |
90 |  for (i=0;i<month> 1 || (month == 1 && day == 29)))
98 |     ++daysInYear;
99 |
100|  totDays = (curDay - daysInYear) + (curYear - year)*365;
101|
102|  // now, correct for leap year
103|  for (i=year+1; i < curYear; ++i) {
104|    if ((i % 4 == 0 && i % 100 != 0) || i % 400 == 0) {
105|      ++totDays;
106|    }
107|  }
108|  return(totDays);
109|}
                     * * * * *  E N D  O F  S O U R C E  * * * * *
```

Figure 18. Example of a C++ Compiler Listing (Part 2 of 14)

```
 15694A01 V1 R2 z/OS C++              'TSCTEST.SAMPLE.SRC(CCNUBRC)'                   05/22/01 20:15:00
                      * * * * *  C R O S S   R E F E R E N C E   L I S T I N G  * * * * *
    ___valist                          :
                 0:133 (D)    0:136 (R)
    __abs                              :
                 0:289 (R)    0:356 (R)
    __absd                             :
                 0:835 (R)    0:912 (R)
    __absf                             :
                 0:834 (R)    0:911 (R)
    __absl                             :
                 0:836 (R)    0:913 (R)
    __acos                             :
                 0:820 (R)    0:915 (R)
    __acosf                            :
                 0:837 (R)    0:914 (R)
    __acosl                            :
                 0:838 (R)    0:916 (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  * * * * *
15694A01 V1 R2 z/OS C++              'TSCTEST.SAMPLE.SRC(CCNUBRC)'                   05/22/01 20:15:00
                                   * * * * *  I N C L U D E S  * * * * *
     1  =  'TSCTEST.CEEZ120.SCEEH.H(STDIO)'
     2  =  'TSCTEST.CEEZ120.SCEEH.H(FEATURES)'
     3  =  'TSCTEST.CEEZ120.SCEEH.SYS.H(TYPES)'
     4  =  'TSCTEST.CEEZ120.SCEEH.H(STRING)'
     5  =  'TSCTEST.CEEZ120.SCEEH.H(MATH)'
     6  =  'TSCTEST.CEEZ120.SCEEH.H(BUILTINS)'
     7  =  'TSCTEST.CEEZ120.SCEEH.H(TIME)'
     8  =  'TSCTEST.CEEZ120.SCEEH(IOSTREAM)'
     9  =  'TSCTEST.CEEZ120.SCEEH(ISTREAM)'
    10  =  'TSCTEST.CEEZ120.SCEEH(OSTREAM)'
    11  =  'TSCTEST.CEEZ120.SCEEH(IOS)'
    12  =  'TSCTEST.CEEZ120.SCEEH(XLOCNUM)'
    13  =  'TSCTEST.CEEZ120.SCEEH(CERRNO)'
    14  =  'TSCTEST.CEEZ120.SCEEH.H(ERRNO)'
    15  =  'TSCTEST.CEEZ120.SCEEH(CLIMITS)'
    16  =  'TSCTEST.CEEZ120.SCEEH.H(LIMITS)'
    17  =  'TSCTEST.CEEZ120.SCEEH(CSTDIO)'
    18  =  'TSCTEST.CEEZ120.SCEEH(CSTDLIB)'
    19  =  'TSCTEST.CEEZ120.SCEEH.H(STDLIB)'
    20  =  'TSCTEST.CEEZ120.SCEEH(STREAMBU)'
    21  =  'TSCTEST.CEEZ120.SCEEH(XIOSBASE)'
    22  =  'TSCTEST.CEEZ120.SCEEH(XLOCALE)'
    23  =  'TSCTEST.CEEZ120.SCEEH(CSTRING)'
    24  =  'TSCTEST.CEEZ120.SCEEH(STDEXCEP)'
    25  =  'TSCTEST.CEEZ120.SCEEH(EXCEPTIO)'
    26  =  'TSCTEST.CEEZ120.SCEEH(XSTDDEF)'
    27  =  'TSCTEST.CEEZ120.SCEEH.H(YVALS)'
    28  =  'TSCTEST.CEEZ120.SCEEH(CSTDDEF)'
    29  =  'TSCTEST.CEEZ120.SCEEH.H(STDDEF)'
    30  =  'TSCTEST.CEEZ120.SCEEH(XSTRING)'
    31  =  'TSCTEST.CEEZ120.SCEEH(XMEMORY)'
    32  =  'TSCTEST.CEEZ120.SCEEH(NEW)'
    33  =  'TSCTEST.CEEZ120.SCEEH(XUTILITY)'
    34  =  'TSCTEST.CEEZ120.SCEEH(UTILITY)'
    35  =  'TSCTEST.CEEZ120.SCEEH(IOSFWD)'
    36  =  'TSCTEST.CEEZ120.SCEEH(CWCHAR)'
    37  =  'TSCTEST.CEEZ120.SCEEH.H(WCHAR)'
    38  =  'TSCTEST.CEEZ120.SCEEH.T(XUTILITY)'
    39  =  'TSCTEST.CEEZ120.SCEEH.T(XSTRING)'
    40  =  'TSCTEST.CEEZ120.SCEEH(TYPEINFO)'
    41  =  'TSCTEST.CEEZ120.SCEEH(XLOCINFO)'
    42  =  'TSCTEST.CEEZ120.SCEEH.H(XLOCINFO)'
    43  =  'TSCTEST.CEEZ120.SCEEH.H(CTYPE)'
    44  =  'TSCTEST.CEEZ120.SCEEH.H(LOCALE)'
    45  =  'TSCTEST.CEEZ120.SCEEH.H(LOCALDEF)'
    46  =  'TSCTEST.CEEZ120.SCEEH.H(LC@CORE)'
    47  =  'TSCTEST.CEEZ120.SCEEH.H(COLLATE)'
    48  =  'TSCTEST.CEEZ120.SCEEH.T(XLOCINFO)'
    49  =  'TSCTEST.CEEZ120.SCEEH.T(OSTREAM)'
    50  =  'TSCTEST.CEEZ120.SCEEH.T(ISTREAM)'
    51  =  'TSCTEST.CEEZ120.SCEEH(IOMANIP)'
    52  =  'TSCTEST.SAMPLE.SRC(CCNUBRH)'
                                 * * * * *  E N D   O F   I N C L U D E S  * * * * *
```

Figure 18. Example of a C++ Compiler Listing (Part 3 of 14)

```
15694A01 V1 R2 z/OS C++               'TSCTEST.SAMPLE.SRC(CCNUBRC)'                    05/22/01 20:15:00
                          * * * * *  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)
    2          0         0           0          0            2
                    * * * * *  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  * * * * *
⋮
15694A01 V1 R2 z/OS C++                                          CCNUBRC                  05/22/01 20:15:00

                     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

     M      N       A      5               0/0       0/0        std::_Lockit::__dftdt()
     M      I       E      7               0/0       1/1        std::allocator<char>::allocate<void>(unsigned int,const void*)
     M      I       E      25              0/0       1/1        std::allocator<char>::max_size() const
     M      N       A      7               1/0       0/0        std::bad_cast::__dftdt()
     M      N       E      36    (15)      1/1       0/0        std::bad_cast::_Doraise() const
     M      I       E      15              0/0       3/3        std::bad_cast::bad_cast(const char*)
     M      I       D      14              0/0       1/1        std::bad_cast::bad_cast(const std::bad_cast&)
     P      -       -      56              0/0       1/0        std::bad_cast::~bad_cast()
     M      N       A      7               0/0       0/0        std::exception::__dftbdt()
     F      I       E      25              0/0       1/1        std::operator>><char,std::char_traits<char>,int>(std::_EBCDIC::
                                                                _LFS_OFF::basic_istream<char,std::char_traits<char> >&,const std::_Smanip<int>&)
                                                                d::_Smanip<int>&)
     F      N       E      4               0/0       0/0        std::operator==<char,char>(const std::allocator<char>&,const st
                                                                d::allocator<char>&)
     M      I       E      5               0/0       2/2        std:ostreambuf_iterator<char,std::char_traits<char> >::failed(
                                                                ) const
     M      I       E      134   (38)      1/1       3/3        std::ostreambuf_iterator<char,std::char_traits<char> >::operato
                                                                r=(char)
     M      N       A      5               0/0       0/0        BioRhythm::__dftdt()
     M      I       E      27              0/0       3/3        BioRhythm::Cycle(int)
     -      C       -      60    (28)      4/1       1/0        BirthDate::BirthDate()
     -      N       E      17              1/0       0/0        BirthDate::BirthDate(const char*)
     -      C       -      43              2/0       2/0        Date::Date()
     -      L       -      236             0/0       1/0        Date::DaysSince(const char*)

  Mode = NOAUTO    Inlining Threshold = 100    Expansion Limit = 1000
⋮
```

Figure 18. Example of a C++ Compiler Listing (Part 4 of 14)

```
15694A01 V1 R2 z/OS C++                                    CCNUBRC                    05/22/01 20:15:00

                    Inline Report (Call Structure)

  Defined Function      : std::bad_cast::bad_cast(const char*)
      Calls To          : 0
   Called From(3,3)     : std::_EBCDIC::_LFS_OFF::use_facet<std> >(const std::_EBCDIC::_LFS_OFF
                          ::locale&)(1,1)
                          std::_EBCDIC::_LFS_OFF::use_facet<std> >> >(const std::_EBCDIC::_LFS_OFF::locale&)(1,1)
                          std::_EBCDIC::_LFS_OFF::use_facet<std> >(const std::_EBCDIC::_LFS_OFF
                          ::locale&)(1,1)


  Defined Function      : std::bad_cast::bad_cast(const std::bad_cast&)
      Calls To          : 0
   Called From(1,1)     : std::bad_cast::_Doraise() const(1,1)

  Defined Function      : std::bad_cast::~bad_cast()
      Calls To          : 0
   Called From(1,0)     : std::bad_cast::__dftdt()(1,0)

  Defined Function      : std::operator>><char,std::char_traits<char>,int>(std::_EBCDIC::_LFS_OFF::basic_istream<char,std
                          ::char_traits<char> >&,const std::_Smanip<int>&)
     Calls To           : 0
   Called From(1,1)     : BirthDate::BirthDate()(1,1)

  Defined Function      : std::ostreambuf_iterator<char,std::char_traits<char> >::failed() const
     Calls To           : 0
   Called From(2,2)     : std::_EBCDIC::_LFS_OFF::basic_ostream<char,std::char_traits<char> >::operator<<(double)(1,1)
                          std::_EBCDID::_LFS_OFF:vbasic_ostream<char,std::char_traits<char> >::operator<<(int)(1,1)

  Defined Function      : std::ostreambuf_iterator<char,std::char_traits<char> >::operator=(char)
     Calls To(1,1)      : std::_EBCDIC::_LFS_OFF::basic_streambuf<char,std::char_traits<char> >::sputc(char)(1,1)
   Called From(3,3)     : std::_EBCDIC::_LFS_OFF::num_put<char,std::ostreambuf_iterator<char,std::char_traits<char> > >::_Put(std
                          ::ostreambuf_iterator<char,std::char_traits<char> >,const char*,unsigned int)(1,1)
                          std::_EBCDIC::_LFS_OFF::num_put<char,std::ostreambuf_iterator<char,std::char_traits<char> > >::_Putc(std
                          ::ostreambuf_iterator<char,std::char_traits<char> >,const char*,unsigned int)(1,1)
                          std::_EBCDIC::_LFS_OFF::num_put<char,std::ostreambuf_iterator<char,std::char_traits<char> > >::_Rep(std
                          ::ostreambuf_iterator<char,std::char_traits<char> >,char,unsigned int)(1,1)
  Defined Function      : BioRhythm::Cycle(int)
      Calls To          : 0
   Called From(3,3)     : operator<<(std::_EBCDIC::_LFS_OFF::basic_ostream<char,std::char_traits<char> >&,BioRhythm&)(3,3)

  Defined Function      : BirthDate::BirthDate()
      Calls To(4,1)     : 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::operator>><char,std::char_traits<char> >(std::_EBCDIC::_LFS_OFF::basic_istream
                          <char,std::char_traits<char> >&,char*)(1.0)
                          std::operator>><char,std::char_traits<char>,int>(std::_EBCDIC::_LFS_OFF::basic_istream<char,std
                          ::char_traits<char> >&,const std::_Smanip<int>&)(1,1) Date::Date()(1,0)
   Called From(1,0)     : main(1,0)

15694A01 V1 R2 z/OS C++                                    CCNUBRC                    05/22/01 20:15:00

                    Inline Report (Call Structure)

  Defined Function      : BirthDate::BirthDate(const char*)
      Calls To(1,0)     : Date::Date()(1,0)
   Called From          : 0

  Defined Function      : Date::Date()
      Calls To(2,0)     : std::_EBCDIC::_LFS_OFF::operator<<<std::char_traits<char> >(std::_EBCDIC::_LFS_OFF::basic_ostream<char,
                          std::char_traits<char> >&,const char*)(2,0)
   Called From(2,0)     : BirthDate::BirthDate()(1,0) BirthDate::BirthDate(const char*)(1,0)

  Defined Function      : Date::DaysSince(const char*)
      Calls To          : 0
   Called From(1,0)     : main(1,0)
```

Figure 18. Example of a C++ Compiler Listing (Part 5 of 14)

```
15694A01 V1 R2 z/OS C++                                          CCNUBRC                  05/22/01 20:15:00

                    Inline Report (Additional Information)

 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::_EBCDIC::_LFS_OFF::basic_ 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.

 INFORMATIONAL CCN1051: Function std::_EBCDIC::_LFS_OFF::numpun exceeds size limit.

 INFORMATIONAL CCN1051: Function std::_EBCDIC::_LFS_OFF::numpun exceeds size limit.

 INFORMATIONAL CCN1051: Function std::_EBCDIC::_LFS_OFF::numpun exceeds size limit.

15694A01 V1 R2 z/OS C++                                          CCNUBRC                  05/22/01 20:15:00          28

 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  F0F1                                    =C'2001'           Compiled Year
 000004  F0F5  F2F2                                    =C'0522'           Compiled Date MMDD
 000008  F2F0  F1F5  F1F6                              =C'201516'         Compiled Time HHMMSS
 00000E  F0F4  F0F1  F0F2                              =C'040102'         Compiler Version
                           Timestamp and Version End
```

⋮

Figure 18. Example of a C++ Compiler Listing (Part 6 of 14)

```
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

                          00001 |       *  //
                          00002 |       *  // Sample Program: Biorhythm
                          00003 |       *  // Description   : Calculates biorhythm based on the current
                          00004 |       *  //                  system date and birth date entered
                          00005 |       *  //
                          00006 |       *  // File 2 of 2-other file is CCNUBRH
                          00007 |       *
                          00008 |       *  #include <stdio.h>
                          00009 |       *  #include <string.h>
                          00010 |       *  #include <math.h>
                          00011 |       *  #include <time.h>
                          00012 |       *  #include <iostream>
                          00013 |       *  #include <iomanip>
                          00014 |       *
                          00015 |       *  using namespace std;
                          00016 |       *
                          00017 |       *  #include "ccnubrh.h"  //BioRhythm class and Date class
                          00018 |       *
                          00019 |       *  int main(void) {
000288                    00019 |        main     DS    0D
000288  47F0  F028        00019 |                 B     40(,r15)
00028C  01C3C5C5                                  CEE eyecatcher
000290  000000C8                                  DSA size
000294  0000B058                                  =A(PPA1-main)
000298  47F0  F001        00019 |                 B     1(,r15)
00029C  58F0  C31C        00019 |                 L     r15,796(,r12)
0002A0  184E              00019 |                 LR    r4,r14
0002A2  05EF              00019 |                 BALR  r14,r15
0002A4  00000000                                  =F'0'
0002A8  0540              00019 |                 BALR  r4,0
0002AA  4140  401E        00019 |                 LA    r4,30(,r4)
0002AE  07F4              00019 |                 BR    r4
0002B0  90E5  D00C        00019 |                 STM   r14,r5,12(r13)
0002B4  58E0  D04C        00019 |                 L     r14,76(,r13)
0002B8  4100  E0C8        00019 |                 LA    r0,200(,r14)
0002BC  5500  C314        00019 |                 CL    r0,788(,r12)
0002C0  4140  F040        00019 |                 LA    r4,64(,r15)
0002C4  4720  F014        00019 |                 BH    20(,r15)
0002C8  5000  E04C        00019 |                 ST    r0,76(,r14)
0002CC  9210  E000        00019 |                 MVI   0(r14),16
0002D0  50D0  E004        00019 |                 ST    r13,4(,r14)
0002D4  18DE              00019 |                 LR    r13,r14
0002D6                    End of Prolog

0002D6  4100  0000        00019 |                 LA    r0,0
0002DA  5830  C1F4        00019 |                 L     r3,_CEECAA_(,r12,500)
0002DE  5850  40B8        00019 |                 L     r5,=Q(@STATIC)(,r4,184)
0002E2  5000  D0A0        00019 |                 ST    r0,<a5:d160:l4>(,r13,160)
0002E6  5000  D0A8        00019 |                 ST    r0,<a5:d168:l4>(,r13,168)
0002EA  5000  D0AC        00019 |                 ST    r0,<a5:d172:l4>(,r13,172)
0002EE  4105  3130        00019 |                 LA    r0,__fsm_tab(r5,r3,304)
0002F2  5000  D0A4        00019 |                 ST    r0,<a5:d164:l4>(,r13,164)
0002F6  1803              00040 |    52           LR    r0,r3
0002F8  58F0  40BC        00040 |    52           L     r15,=V(BirthDate::BirthDate())(,r4,188)
0002FC  4110  D0B4        00040 |    52           LA    r1,bio.birthDate@4(,r13,180)
000300  4DE0  F010        00040 |    52           BAS   r14,16(,r15)
```

Figure 18. Example of a C++ Compiler Listing (Part 7 of 14)

```
15694A01 V1 R2 z/OS C++                                  CCNUBRC: main             05/22/01 20:15:00          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

 000304  4700  0004       00040 |   52           NOP   4
 000308  4120  D0BC       00041 |   52           LA    r2,bio.text@c(,r13,188)
 00030C  1803             00041 |   52           LR    r0,r3
 00030E  58F0  40C0       00041 |   52           L     r15,=V(Date::DaysSince(const char*))(,r4,192)
 000312  4110  D0B4       00041 |   52           LA    r1,bio.birthDate@4(,r13,180)
 000316  4DE0  F010       00041 |   52           BAS   r14,16(,r15)
 00031A  4700  0008       00041 |   52           NOP   8
                          00020 |       *    BioRhythm bio;
                          00021 |       *    int code;
                          00022 |       *
                          00023 |       *    if (!bio.ok()) {
 00031E  1211             00023 |                LTR   r1,r1
 000320  5010  D0B0       00041 |   52           ST    r1,<a5:d176:l4>(,r13,176)
 000324  9201  D0A3       00040 |   52           MVI   <a5:d163:l1>(r13,163),1
 000328  47B0  4088       00023 |                BNL   @5L45
                          00024 |       *      cerr << "Error in birthdate specification - format is yyyy/mm/dd";
 00032C  58E0  40C4       00024 |                L     r14,=A(@CONSTANT_AREA)(,r4,196)
 000330  5815  3008       00024 |                L     r1,cerr__Q3_3std7_EBCDIC8_LFS_OFF(r5,r3,8)
 000334  1803             00024 |                LR    r0,r3
 000336  58F0  40C8       00024 |                L     r15,=V(std::_EBCDIC::_LFS_OFF::operator<<<std::char_traits<...(,r4,200)
 00033A  4120  E0F8       00024 |                LA    r2,+CONSTANT_AREA(,r14,248)
 00033E  4DE0  F010       00024 |                BAS   r14,16(,r15)
 000342  4700  0008       00024 |                NOP   8
                          00025 |       *      code = 8;
 000346  41F0  0008       00025 |                LA    r15,8
                          00026 |       *    }
 00034A  47F0  40A2       00026 |                B     @5L5442
 00034E  0700             00026 |                NOPR  0
 000350                   00026 |        @5L45   DS    0H
                          00027 |       *    else {
                          00028 |       *      cout << bio;  // write out birthdate for bio
 000350  5815  300C       00028 |                L     r1,cout__Q3_3std7_EBCDIC8_LFS_OFF(r5,r3,12)
 000354  4120  D0B0       00028 |                LA    r2,bio(,r13,176)
 000358  1803             00028 |                LR    r0,r3
 00035A  58F0  40CC       00028 |                L     r15,=V(operator<<(std::_EBCDIC::_LFS_OFF::basic_ostream<cha...(,r4,204)
 00035E  4DE0  F010       00028 |                BAS   r14,16(,r15)
 000362  4700  0008       00028 |                NOP   8
 000366  41F0  0000       00043 |   52           LA    r15,0
                          00029 |       *      code = 0;
                          00030 |       *    }
                          00031 |       *    return(code);
                          00032 |       *  }
 00036A                   00032 |        @5L5442  DS    0H
 00036A  5030  C1F4       00019 |                ST    r3,_CEECAA_(,r12,500)

 00036E                   Start of Epilog
 00036E  180D             00032 |                LR    r0,r13
 000370  58D0  D004       00032 |                L     r13,4(,r13)
 000374  58E0  D00C       00032 |                L     r14,12(,r13)
 000378  9825  D01C       00032 |                LM    r2,r5,28(r13)
 00037C  051E             00032 |                BALR  r1,r14
 00037E  0707             00032 |                NOPR  7

 000380                   Start of Literals
 000380  00000000                                =Q(@STATIC)
 000384  00000000                                =V(BirthDate::BirthDate())
```

Figure 18. Example of a C++ Compiler Listing (Part 8 of 14)

```
15694A01 V1 R2 z/OS C++                                          CCNUBRC: main             05/22/01 20:15:00          38

 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

 000388  00000000                                     =V(Date::DaysSince(const char*))
 00038C  0000AFF8                                     =A(@CONSTANT_AREA)
 000390  00000000                                     =V(std::_EBCDIC::_LFS_OFF::operator<<<std::char_traits<char> >(std::_EB
 000394  00000000                                     =V(operator<<(std::_EBCDIC::_LFS_OFF::basic_ostream<char,std::char_trai
 000398                   End of Literals

                          ***   General purpose registers used: 1111110000001111
                          ***   Floating point  registers used: 1010101000000000
                          ***   Size of register spill area: 128(max) 0(used)
                          ***   Size of dynamic storage: 200
                          ***   Size of executable code: 248

15694A01 V1 R2 z/OS C++                                          CCNUBRC: BioRhythm::__...) 05/22/01 20:15:00          39

 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

                                        BioRhythm::__dftdt()
 000398                   00000 |                DS    0D
 000398  47F0  F001       00000 |                B     1(,r15)
 00039C  01C3C5C5                                      CEE eyecatcher
 0003A0  000000C0                                      DSA size
 0003A4  0000AF88                                      =A(PPA1-BioRhythm::__dftdt())
 0003A8                   End of Prolog

 0003A8                   00000 |        @6L5457  DS    0H

 0003A8                   Start of Epilog
 0003A8  47F0  E004       00000 |                B     4(,r14)
 0003AC  0707             00000 |                NOPR  7

                          ***   General purpose registers used: 0000000000000000
                          ***   Floating point  registers used: 0000000000000000
                          ***   Size of register spill area: 128(max) 0(used)
                          ***   Size of dynamic storage: 192
                          ***   Size of executable code: 22
```

Figure 18. Example of a C++ Compiler Listing (Part 9 of 14)

```
 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

                           00059 |       *  BirthDate::BirthDate() {
                                          BirthDate::BirthDate()
 0003B0                    00059 |                 DS    0D
 0003B0  47F0  F001        00059 |                 B     1(,r15)
 0003B4  01C3C5C5                                        CEE eyecatcher
 0003B8  000000E8                                        DSA size
 0003BC  0000AFC0                                        =A(PPA1-BirthDate::BirthDate())
 0003C0  90E7  D00C        00059 |                 STM   r14,r7,12(r13)
 0003C4  58E0  D04C        00059 |                 L     r14,76(,r13)
 0003C8  4100  E0E8        00059 |                 LA    r0,232(,r14)
 0003CC  5500  C314        00059 |                 CL    r0,788(,r12)
 0003D0  4140  F04C        00059 |                 LA    r4,76(,r15)
 0003D4  47D0  F03A        00059 |                 BNH   58(,r15)
 0003D8  58F0  C31C        00059 |                 L     r15,796(,r12)
 0003DC  184E              00059 |                 LR    r4,r14
 0003DE  05EF              00059 |                 BALR  r14,r15
 0003E0  00000004                                        =F'4'
 0003E4  0540              00059 |                 BALR  r4,0
 0003E6  4140  4016        00059 |                 LA    r4,22(,r4)
 0003EA  5000  E04C        00059 |                 ST    r0,76(,r14)
 0003EE  9210  E000        00059 |                 MVI   0(r14),16
 0003F2  50D0  E004        00059 |                 ST    r13,4(,r14)
 0003F6  5800  D014        00059 |                 L     r0,20(,r13)
 0003FA  18DE              00059 |                 LR    r13,r14
 0003FC                    End of Prolog

 0003FC  1830              00059 |                 LR    r3,r0
 0003FE  1851              00059 |                 LR    r5,r1
 000400  58F0  40B0        00059 |                 L     r15,=V(Date::Date())(,r4,176)
 000404  4DE0  F010        00059 |                 BAS   r14,16(,r15)
 000408  4700  0004        00059 |                 NOP   4
                           00060 |       *    cout << "Please enter your birthdate in the form yyyy/mm/dd\n";
 00040C  5860  40B4        00060 |                 L     r6,=Q(@STATIC)(,r4,180)
 000410  58E0  40B8        00060 |                 L     r14,=A(@CONSTANT_AREA)(,r4,184)
 000414  1803              00060 |                 LR    r0,r3
 000416  58F0  40BC        00060 |                 L     r15,=V(std::_EBCDIC::_LFS_OFF::operator<<<std::char_traits<...(,r4,188)
 00041A  5816  300C        00060 |                 L     r1,cout__Q3_3std7_EBCDIC8_LFS_OFF(r6,r3,12)
 00041E  4120  E0C4        00060 |                 LA    r2,+CONSTANT_AREA(,r14,196)
 000422  4DE0  F010        00060 |                 BAS   r14,16(,r15)
 000426  4700  0008        00060 |                 NOP   8
                           00061 |       *    cin >> setw(dateLen+1) >> text;
 00042A  5816  3038        00061 |                 L     r1,dateLen__4Date(r6,r3,56)
 00042E  58E6  303C        00061 |                 L     r14,setw__Q3_3std7_EBCDIC8_LFS_OFFFi(r6,r3,60)
 000432  4170  D0D8        00061 |                 LA    r7,#wtemp_22(,r13,216)
 000436  5810  1000        00061 |                 L     r1,dateLen__4Date(,r1,0)
 00043A  A71A  0001        00061 |                 AHI   r1,H'1'
 00043E  98F0  E008        00061 |                 LM    r15,r0,&Func_&WSA(r14,8)
 000442  4DE0  F010        00061 |                 BAS   r14,16(,r15)
 000446  4700  0004        00061 |                 NOP   4
 00044A  5866  3034        00061 |                 L     r6,cin__Q3_3std7_EBCDIC8_LFS_OFF(r6,r3,52)
 00044E  1266              00098 |   51  +         LTR   r6,r6
 000450  5010  D0D8        00061 |                 ST    r1,<a7:d216:l4>(,r13,216)
 000454  5020  D0DC        00061 |                 ST    r2,<a7:d220:l4>(,r13,220)
 000458  D207  D0E0  7000  00061 |                 MVC   __l3(8,r13,224),(_Smanip<int>)(r7,0)
 00045E  4770  406E        00098 |   51  +         BNE   @7L51
 000462  4110  0000        00098 |   51  +         LA    r1,0
```

Figure 18. Example of a C++ Compiler Listing (Part 10 of 14)

```
15694A01 V1 R2 z/OS C++                                  CCNUBRC: BirthDate::Bi...) 05/22/01 20:15:00          41

 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

 000466  47F0  4072        00098 |   51  +         B     @7L52
 00046A                    00098 |   51  +@7L51    DS    0H
 00046A  5810  6004        00098 |   51  +         L     r1,(basic_istream<char,std::char_traits<char> >)[]@4(,r6,4)
 00046E                    00098 |   51  +@7L52    DS    0H
 00046E  58E0  D0E0        00098 |   51  +         L     r14,<a7:d224:l4>(,r13,224)
 000472  5820  D0E4        00098 |   51  +         L     r2,<a7:d228:l4>(,r13,228)
 000476  98F0  E008        00098 |   51  +         LM    r15,r0,&Func_&WSA(r14,8)
 00047A  4DE0  F010        00098 |   51  +         BAS   r14,16(,r15)
 00047E  4700  0008        00098 |   51  +         NOP   8
 000482  4120  5008        00099 |   51  +         LA    r2,(char)(,r5,8)
 000486  1803              00099 |   51  +         LR    r0,r3
 000488  58F0  40C0        00099 |   51  +         L     r15,=V(std::_EBCDIC::_LFS_OFF::operator>><char,std::char_tr...(,r4,192)
 00048C  1816              00099 |   51  +         LR    r1,r6
 00048E  4DE0  F010        00099 |   51  +         BAS   r14,16(,r15)
 000492  4700  0008        00099 |   51  +         NOP   8
                           00062 |       *  }
 000496  1815              00062 |                 LR    r1,r5
 000498                    00062 |        @7L5446  DS    0H

 000498                    Start of Epilog
 000498  58D0  D004        00062 |                 L     r13,4(,r13)
 00049C  58E0  D00C        00062 |                 L     r14,12(,r13)
 0004A0  9827  D01C        00062 |                 LM    r2,r7,28(r13)
 0004A4  47F0  E004        00062 |                 B     4(,r14)
 0004A8  0707              00062 |                 NOPR  7
 0004AA  0000

 0004AC                    Start of Literals
 0004AC  00000000                                  =V(Date::Date())
 0004B0  00000000                                  =Q(@STATIC)
 0004B4  0000AFF8                                  =A(@CONSTANT_AREA)
 0004B8  00000000                                  =V(std::_EBCDIC::_LFS_OFF::operator<<<std::char_traits<char> >(std::_EB
 0004BC  00000000                                  =V(std::_EBCDIC::_LFS_OFF::operator>><char,std::char_traits<char> >(std
 0004C0                    End of Literals

                           ***   General purpose registers used: 1111111100001111
                           ***   Floating point  registers used: 1010101000000000
                           ***   Size of register spill area: 128(max) 0(used)
                           ***   Size of dynamic storage: 232
                           ***   Size of executable code: 250
```

⋮

Figure 18. Example of a C++ Compiler Listing (Part 11 of 14)

```
                    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   00EDD8           @STATICP
                   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           __fsm_tab____ls__Q3_3std7_EBCDIC8_LF
                                                     S_OFFHQ2_3std11char_traitsXTc__RQ4_3
                                                     std7_EBCDIC8_LFS_OFF13basic_ostreamX
                                                     TcTQ2_3std11char_traitsXTc__PCc_RQ4_
                                                     3std7_EBCDIC8_LFS_OFF13basic_ostream
                                                     XTcTQ2_3std11char_traitsXTc__
                   PR    9 000000   000058           __fsm_tab____rs__Q3_3std7_EBCDIC8_LF
                                                     S_OFFHcQ2_3std11char_traitsXTc__RQ4_
                                                     3std7_EBCDIC8_LFS_OFF13basic_istream
                                                     XTcTQ2_3std11char_traitsXTc__Pc_RQ4_
                                                     3std7_EBCDIC8_LFS_OFF13basic_istream
                                                     XTcTQ2_3std11char_traitsXTc__
                   PR   10 000000   000030           numDays__4Date
                   PR   11 000000   00000C           __3__use_facet__Q3_3std7_EBCDIC8_LFS
                                                     _OFFHQ4_3std7_EBCDIC8_LFS_OFF5ctypeX
                                                     Tc__RCQ4_3std7_EBCDIC8_LFS_OFF6local
                                                     e_RCQ4_3std7_EBCDIC8_LFS_OFF5ctypeXT
                                                     c_
                   PR   12 000000   000004           _Psave__use_facet__Q3_3std7_EBCDIC8_
                                                     LFS_OFFHQ4_3std7_EBCDIC8_LFS_OFF5cty
                                                     peXTc__RCQ4_3std7_EBCDIC8_LFS_OFF6lo
                                                     cale_RCQ4_3std7_EBCDIC8_LFS_OFF5ctyp
                                                     eXTc___2
                   PR   13 000000   00000C           __3__use_facet__Q3_3std7_EBCDIC8_LFS
                                                     _OFFHQ4_3std7_EBCDIC8_LFS_OFF7num_pu
                                                     tXTcTQ2_3std19ostreambuf_iteratorXTc
                                                     TQ2_3std11char_traitsXTc____RCQ4_3st
                                                     d7_EBCDIC8_LFS_OFF6locale_RCQ4_3std7
                                                     _EBCDIC8_LFS_OFF7num_putXTcTQ2_3std1
                                                     9ostreambuf_iteratorXTcTQ2_3std11cha
                                                     r_traitsXTc___
                   PR   14 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
                   PR   15 000000   000004           id__Q4_3std7_EBCDIC8_LFS_OFF7num_put
                                                     XTcTQ2_3std19ostreambuf_iteratorXTcT
                                                     Q2_3std11char_traitsXTc___
                   PR   16 000000   000004           _Facsav__Q4_3std7_EBCDIC8_LFS_OFF8_T
                                                     idyfacXTQ4_3std7_EBCDIC8_LFS_OFF5cty
                                                     peXTc__
                   PR   17 000000   000004           _Facsav__Q4_3std7_EBCDIC8_LFS_OFF8_T
                                                     idyfacXTQ4_3std7_EBCDIC8_LFS_OFF7num
                                                     _putXTcTQ2_3std19ostreambuf_iterator
```

⋮

Figure 18. Example of a C++ Compiler Listing (Part 12 of 14)

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
Date::dateLen	dateLen__4Date
Date::numMonths	numMonths__4Date
BioRhythm::pCycle	pCycle__9BioRhythm
BioRhythm::eCycle	eCycle__9BioRhythm
BioRhythm::iCycle	iCycle__9BioRhythm
std::_EBCDIC::_LFS_OFF ::__fsm_tab____ls<std::char_traits <char> >(std::_EBCDIC::_LFS_OFF ::basic_ostream<char,std ::char_traits<char> >&,const char*)	__fsm_tab____ls__Q3_3std7_EBCDIC8_LF S_OFFHQ2_3std11char_traitsXTc__RQ4_3 std7_EBCDIC8_LFS_OFF13basic_ostreamX TcTQ2_3std11char_traitsXTc__PCc_RQ4_ 3std7_EBCDIC8_LFS_OFF13basic_ostream XTcTQ2_3std11char_traitsXTc__
std::_EBCDIC::_LFS_OFF ::__fsm_tab____rs<char,std ::char_traits<char> >(std::_EBCDIC ::_LFS_OFF::basic_istream<char,std ::char_traits<char> >&,char*)	__fsm_tab____rs__Q3_3std7_EBCDIC8_LF S_OFFHcQ2_3std11char_traitsXTc__RQ4_ 3std7_EBCDIC8_LFS_OFF13basic_istream XTcTQ2_3std11char_traitsXTc__Pc_RQ4_ 3std7_EBCDIC8_LFS_OFF13basic_istream XTcTQ2_3std11char_traitsXTc__
Date::numDays	numDays__4Date
std::_EBCDIC::_LFS_OFF ::__3__use_facet<std::_EBCDIC ::_LFS_OFF::ctype<char> >(const std ::_EBCDIC::_LFS_OFF::locale&)	__3__use_facet__Q3_3std7_EBCDIC8_LFS _OFFHQ4_3std7_EBCDIC8_LFS_OFF5ctypeX Tc__RCQ4_3std7_EBCDIC8_LFS_OFF6locale_RCQ4_3std7_EBCDIC8_LFS_OFF5ctypeXT c_
std::_EBCDIC::_LFS_OFF ::_Psave__use_facet<std::_EBCDIC ::_LFS_OFF::ctype<char> >(const std ::_EBCDIC::_LFS_OFF::locale&)	_Psave__use_facet__Q3_3std7_EBCDIC8_ LFS_OFFHQ4_3std7_EBCDIC8_LFS_OFF5cty peXTc__RCQ4_3std7_EBCDIC8_LFS_OFF6lo cale_RCQ4_3std7_EBCDIC8_LFS_OFF5ctyp eXTc___2
std::_EBCDIC::_LFS_OFF ::__3__use_facet<std::_EBCDIC ::_LFS_OFF::num_put<char,std ::ostreambuf_iterator<char,std ::char_traits<char> > > >(const std ::_EBCDIC::_LFS_OFF::locale&)	__3__use_facet__Q3_3std7_EBCDIC8_LFS _OFFHQ4_3std7_EBCDIC8_LFS_OFF7num_pu tXTcTQ2_3std19ostreambuf_iteratorXTc TQ2_3std11char_traitsXTc____RCQ4_3st d7_EBCDIC8_LFS_OFF6locale_RCQ4_3std7 _EBCDIC8_LFS_OFF7num_putXTcTQ2_3std1 9ostreambuf_iteratorXTcTQ2_3std11cha r_traitsXTc___
std::_EBCDIC::_LFS_OFF ::_Psave__use_facet<std::_EBCDIC ::_LFS_OFF::num_put<char,std ::ostreambuf_iterator<char,std ::char_traits<char> > > >(const std ::_EBCDIC::_LFS_OFF::locale&)	_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
std::_EBCDIC::_LFS_OFF::num_put <char,std::ostreambuf_iterator<char, std::char_traits<char> > >::id	id__Q4_3std7_EBCDIC8_LFS_OFF7num_put XTcTQ2_3std19ostreambuf_iteratorXTcT Q2_3std11char_traitsXTc___
std::_EBCDIC::_LFS_OFF::_Tidyfac <std::_EBCDIC::_LFS_OFF::ctype<char> >::_Facsav	_Facsav__Q4_3std7_EBCDIC8_LFS_OFF8_T idyfacXTQ4_3std7_EBCDIC8_LFS_OFF5cty peXTc__
std::_EBCDIC::_LFS_OFF::_Tidyfac <std::_EBCDIC::_LFS_OFF::num_put <char,std::ostreambuf_iterator<char,	_Facsav__Q4_3std7_EBCDIC8_LFS_OFF8_T idyfacXTQ4_3std7_EBCDIC8_LFS_OFF7num _putXTcTQ2_3std19ostreambuf_iterator

⋮

Figure 18. Example of a C++ Compiler Listing (Part 13 of 14)

* * * * * S T A T I C M A P * * * * *

OFFSET (HEX)	LENGTH (HEX)	NAME
0	1	_C
4	4	__2
8	4	cerr__Q3_3std7_EBCDIC8_LFS_OFF
C	4	cout__Q3_3std7_EBCDIC8_LFS_OFF
10	4	pCycle__9BioRhythm
14	4	__fmod
18	4	__sin
1C	4	eCycle__9BioRhythm
20	4	iCycle__9BioRhythm
24	4	time
28	4	localtime
2C	4	asctime
30	4	endl__Q3_3std7_EBCDIC8_LFS_OFFHcQ2_3std11char_traitsXTc__RQ4_3std7_EBCDIC8_LFS_OFF13basic_ostream Q2_3std11char_traitsXTc___RQ4_3std7_EBCDIC8_LFS_OFF13basic_ostreamXTcTQ2_3std11char_traitsXTc__
34	4	cin__Q3_3std7_EBCDIC8_LFS_OFF
38	4	dateLen__4Date
3C	4	setw__Q3_3std7_EBCDIC8_LFS_OFFFi
40	4	sscanf
44	4	numDays__4Date
48	4	__ct__Q2_3std7_LockitFi
4C	4	id__Q4_3std7_EBCDIC8_LFS_OFF5ctypeXTc_
50	4	_Getfacet__Q4_3std7_EBCDIC8_LFS_OFF6localeCFUi
54	4	_Facsav__Q4_3std7_EBCDIC8_LFS_OFF8_TidyfacXTQ4_3std7_EBCDIC8_LFS_OFF5ctypeXTc__
58	4	_Tidy__Q4_3std7_EBCDIC8_LFS_OFF8_TidyfacXTQ4_3std7_EBCDIC8_LFS_OFF5ctypeXTc__Fv
5C	4	realloc
60	4	malloc
64	4	__vftQ2_3std9exception__3std
68	4	__Throw
6C	4	clear__Q4_3std7_EBCDIC8_LFS_OFF8ios_baseFib
70	4	id__Q4_3std7_EBCDIC8_LFS_OFF7num_putXTcTQ2_3std19ostreambuf_iteratorXTcTQ2_3std11char_traitsXTc__
74	4	_Facsav__Q4_3std7_EBCDIC8_LFS_OFF8_TidyfacXTQ4_3std7_EBCDIC8_LFS_OFF7num_putXTcTQ2_3std19ostreamb teratorXTcTQ2_3std11char_traitsXTc____
78	4	_Tidy__Q4_3std7_EBCDIC8_LFS_OFF8_TidyfacXTQ4_3std7_EBCDIC8_LFS_OFF7num_putXTcTQ2_3std19ostreambuf ratorXTcTQ2_3std11char_traitsXTc____Fv
7C	4	free
80	4	__dl__FPv
84	4	_Id_cnt__Q5_3std7_EBCDIC8_LFS_OFF6locale2id
88	4	__dt__Q2_3std7_LockitFv
8C	4	__unexpected__Fv
90	4	terminate__Fv
94	4	__ReThrow
98	4	__nw__FUi
9C	4	uncaught_exception__3stdFv
A0	4	__ct__Q4_3std7_EBCDIC8_LFS_OFF8_LocinfoFPCc
A4	4	__dt__Q4_3std7_EBCDIC8_LFS_OFF8_LocinfoFv
A8	4	__dt__Q2_3std9exceptionFv
AC	4	_Getctype
B0	4	_Cltab__Q4_3std7_EBCDIC8_LFS_OFF5ctypeXTc_
B4	4	_Tolower
B8	4	_Toupper
BC	4	id__Q4_3std7_EBCDIC8_LFS_OFF8numpunctXTc_
C0	4	_Facsav__Q4_3std7_EBCDIC8_LFS_OFF8_TidyfacXTQ4_3std7_EBCDIC8_LFS_OFF8numpunctXTc__
C4	4	_Tidy__Q4_3std7_EBCDIC8_LFS_OFF8_TidyfacXTQ4_3std7_EBCDIC8_LFS_OFF8numpunctXTc__Fv
C8	4	npos__Q4_3std7_EBCDIC8_LFS_OFF12basic_stringXTcTQ2_3std11char_traitsXTc_TQ2_3std9allocatorXTc__

⋮

* * * * * E N D O F S T A T I C M A P * * * * *

* * * * * E N D O F C O M P I L A T I O N * * * * *

Figure 18. Example of a C++ Compiler Listing (Part 14 of 14)

z/OS C++ Compiler Listing Components

The following sections describe 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 274 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 HFS 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 HFS file that contains the source code is greater than 32 characters, it is truncated. Only the rightmost 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 sections describe 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 option `XREF` 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 option `ATTR` 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 *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 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 option `LIST` 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 15. Filter Utility” on page 447.

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 19 on page 275 shows an example of an IPA Link step listing.

```
15694A01 V1 R2 z/OS C/C++ IPA               'TSIPA.TEST.LINKCNTL(INCLCNTL)'                  05/23/2001 15:09:23   Page     1

                                  * * * * *  P R O L O G  * * * * *

  Compile Time Library . . . . . . : 41020000
  Command options:
     Primary input name. . . . . . : 'TSIPA.TEST.LINKCNTL(INCLCNTL)'
     Compiler options. . . . . . . : *IPA(LINK,MAP,NOREFMAP,LEVEL(1),DUP,ER,NCAL,NOUPCASE,NOCONTROL)  *NOGONUMBER  *NOALIAS
                                   : *NODECK      *TERMINAL    *LIST        *XREF        *NOATTR      *NOOFFSET    *MEMORY
                                   : *NOCSECT     *LIBANSI     *FLAG(I)     *NOTEST(NOSYM,NOBLOCK,NOLINE,NOPATH,HOOK)
                                   : *OPTIMIZE(1) *INLINE(AUTO,REPORT,1000,8000)          *OBJECT      *OPTFILE(DD:OPTION)
                                   : *NOSERVICE   *NOOE        *NOLOCALE    *HALT(16)    *NOGOFF      *IPADBG(TRACETPO)

                               * * * * *  E N D  O F  P R O L O G  * * * * *
15694A01 V1 R2 z/OS C/C++ IPA               'TSIPA.TEST.LINKCNTL(INCLCNTL)'                  05/23/2001 15:09:23   Page     2

                            * * * * *  O B J E C T  F I L E  M A P  * * * * *

 *ORIGIN  IPA  FILE ID  FILE NAME
    P              1   TSIPA.TEST.LINKCNTL(INCLCNTL)
    PI     Y       2   TSIPA.TEST.PASS1.OBJ(INCLMAIN)
    PI     Y       3   TSIPA.TEST.PASS1.OBJ(INCLRTN1)
    PI     Y       4   TSIPA.TEST.PASS1.OBJ(INCLRTN2)

 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  * * * * *
15694A01 V1 R2 z/OS C/C++ IPA               'TSIPA.TEST.LINKCNTL(INCLCNTL)'                  05/23/2001 15:09:23   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(2)     *ARGPARSE
                       *CHARSET(BIAS=EBCDIC,LIB=EBCDIC)       *NOCOMPACT   *NOCOMPRESS  *NODLL(NOCALLBACKANY)     *ENV(MVS)
                       *EXECOPS     *FLOAT(HEX,FOLD,NOAFP)    *NOGONUMBER  *NOIGNERRNO  *NOINITAUTO
                       *IPA(NOLINK,NOOBJECT,COMPRESS)         *NOLIBANSI   *NOLIST      *NOLOCALE    *LONGNAME
                       *MAXMEM(2097152)          *OPTIMIZE(1) *PLIST(HOST) *REDIR       *NORENT      *NOROCONST   *SPILL(128)
                       *NOSTART     *STRICT      *NOSTRICT_INDUCTION       *NOTEST      *TUNE(3)     *NOXPLINK    *NOXREF

              2        *AGGRCOPY(NOOVERLAP)      *NOALIAS     *ANSIALIAS   *ARCH(2)     *ARGPARSE
                       *CHARSET(BIAS=EBCDIC,LIB=EBCDIC)       *NOCOMPACT   *NOCOMPRESS  *NODLL(NOCALLBACKANY)     *ENV(MVS)
                       *EXECOPS     *FLOAT(HEX,FOLD,NOAFP)    *NOGONUMBER  *NOIGNERRNO  *NOINITAUTO
                       *IPA(NOLINK,NOOBJECT,COMPRESS)         *NOLIBANSI   *NOLIST      *NOLOCALE    *LONGNAME
                       *MAXMEM(2097152)          *OPTIMIZE(1) *PLIST(HOST) *REDIR       *NORENT      *NOROCONST   *SPILL(128)
                       *NOSTART     *STRICT      *NOSTRICT_INDUCTION       *NOTEST      *TUNE(3)     *NOXPLINK    *NOXREF

              3        *AGGRCOPY(NOOVERLAP)      *NOALIAS     *ANSIALIAS   *ARCH(2)     *ARGPARSE
                       *CHARSET(BIAS=EBCDIC,LIB=EBCDIC)       *NOCOMPACT   *NOCOMPRESS  *NODLL(NOCALLBACKANY)     *ENV(MVS)
                       *EXECOPS     *FLOAT(HEX,FOLD,NOAFP)    *NOGONUMBER  *NOIGNERRNO  *NOINITAUTO
                       *IPA(NOLINK,NOOBJECT,COMPRESS)         *NOLIBANSI   *NOLIST      *NOLOCALE    *LONGNAME
                       *MAXMEM(2097152)          *OPTIMIZE(1) *PLIST(HOST) *REDIR       *NORENT      *NOROCONST   *SPILL(128)
                       *NOSTART     *STRICT      *NOSTRICT_INDUCTION       *NOTEST      *TUNE(3)     *NOXPLINK    *NOXREF


                  * * * * *  E N D  O F  C O M P I L E R  O P T I O N S  M A P  * * * * *
```

Figure 19. Example of an IPA Link Step Listing (Part 1 of 7)

```
15694A01 V1 R2 z/OS C/C++ IPA               'TSIPA.TEST.LINKCNTL(INCLCNTL)'                  05/23/2001 15:09:23   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      N       -      192 (94)         11/11     0          main
     A      I       D      0 (18)           0         1/1        Incl_Rtn1
     A      I       D      0 (8)            0         10/10      Incl_Rtn2
  Mode = AUTO     Inlining Threshold =    1000    Expansion Limit =    8000

15694A01 V1 R2 z/OS C/C++ IPA                'TSIPA.TEST.LINKCNTL(INCLCNTL)'                  05/23/2001 15:09:23   Page     5

                    IPA Inline Report (Call Structure)

  Defined Subprogram    : main
      Calls To(11,11)   : Incl_Rtn2(10,10)
                          Incl_Rtn1(1,1)
   Called From          : 0

  Defined Subprogram    : Incl_Rtn2
      Calls To          : 0
   Called From(10,10)   : main(10,10)

  Defined Subprogram    : Incl_Rtn1
      Calls To          : 0
   Called From(1,1)     : main(1,1)


                      * * * * *  E N D  O F  I N L I N E  R E P O R T  * * * * *
```

Figure 19. Example of an IPA Link Step Listing (Part 2 of 7)

```
15694A01 V1 R2 z/OS C/C++ IPA                    Partition 0                    05/23/2001 15:09:23   Page     6

                          * * * * *  P A R T I T I O N  M A P  * * * * *

 PARTITION 0

 PARTITION CSECT NAMES:
      Code: none
    Static: none
      Test: none

 PARTITION DESCRIPTION:
    Initialization data partition

 COMPILER OPTIONS FOR PARTITION 0:
   *AGGRCOPY(NOOVERLAP)      *NOALIAS     *ARCH(2)     *ARGPARSE    *CHARSET(BIAS=EBCDIC,LIB=EBCDIC)       *NOCOMPACT   *NOCOMPRESS
   *NOCSECT     *NODLL       *ENV(MVS)    *EXECOPS     *FLOAT(HEX,FOLD,NOAFP)    *NOGOFF      *NOGONUMBER  *NOIGNERRNO  *NOINITAUTO
   *IPA(LINK)   *LIBANSI     *LIST        *NOLOCALE    *LONGNAME    *MAXMEM(2097152)          *OPTIMIZE(1) *PLIST(HOST) *REDIR
   *NORENT      *NOROCONST   *SPILL(128)  *START       *STRICT      *NOSTRICT_INDUCTION       *NOTEST      *TUNE(3)     *NOXPLINK
   *XREF

 SYMBOLS IN PARTITION 0:

  *TYPE     FILE ID    SYMBOL
     D           1     gbl

   TYPE:  F=function     D=data

 SOURCE FILES FOR PARTITION 0:

  *ORIGIN     FILE ID     SOURCE FILE NAME
      P            1      TSIPA.TEST.C(INCLMAIN)

  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 19. Example of an IPA Link Step Listing (Part 3 of 7)

```
15694A01 V1 R2 z/OS C/C++ IPA                              Partition 0                  05/23/2001 15:09:23   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  F0F1                                  =C'2001'           Compiled Year
 000004  F0F5  F2F3                                  =C'0523'           Compiled Date MMDD
 000008  F1F5  F0F9  F1F1                            =C'150911'         Compiled Time HHMMSS
 00000E  F0F1  F0F2  F0F0                            =C'010200'         Compiler Version
                           Timestamp and Version End

15694A01 V1 R2 z/OS C/C++ IPA                              Partition 0                  05/23/2001 15:09:23   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

15694A01 V1 R2 z/OS C/C++ IPA                              Partition 0                  05/23/2001 15:09:23   Page     9

                                  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   000018           @STATICP
                         SD    2 000000   000004           gbl
15694A01 V1 R2 z/OS C/C++ IPA                              Partition 0                  05/23/2001 15:09:23   Page    10

                              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
           gbl                                             gbl
15694A01 V1 R2 z/OS C/C++ IPA                              Partition 1                  05/23/2001 15:09:23   Page    11

                                * * * * *   P A R T I T I O N   M A P   * * * * *

 PARTITION 1 OF 1

 PARTITION SIZE:
    Actual: 3432
     Limit: 102400

 PARTITION CSECT NAMES:
      Code: none
    Static: none
      Test: none

 PARTITION DESCRIPTION:
    Primary partition

 COMPILER OPTIONS FOR PARTITION 1:
   *AGGRCOPY(NOOVERLAP)      *NOALIAS     *ARCH(2)     *ARGPARSE    *CHARSET(BIAS=EBCDIC,LIB=EBCDIC)       *NOCOMPACT   *NOCOMPRESS
   *NOCSECT     *NODLL       *ENV(MVS)    *EXECOPS     *FLOAT(HEX,FOLD,NOAFP)    *NOGOFF      *NOGONUMBER  *NOIGNERRNO  *NOINITAUTO
   *IPA(LINK)   *LIBANSI     *LIST        *NOLOCALE    *LONGNAME    *MAXMEM(2097152)          *OPTIMIZE(1) *PLIST(HOST) *REDIR
   *NORENT      *NOROCONST   *SPILL(128)  *START       *STRICT      *NOSTRICT_INDUCTION       *NOTEST      *TUNE(3)     *NOXPLINK
   *XREF

 SYMBOLS IN PARTITION 1:

  *TYPE     FILE ID    SYMBOL
     F           1     main

   TYPE:  F=function     D=data

 SOURCE FILES FOR PARTITION 1:

  *ORIGIN     FILE ID     SOURCE FILE NAME
      P            1      TSIPA.TEST.C(INCLMAIN)
      P            2      TSIPA.TEST.C(INCLRTN1)
      P            3      TSIPA.TEST.C(INCLRTN2)

  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 19. Example of an IPA Link Step Listing (Part 4 of 7)

```
15694A01 V1 R2 z/OS C/C++ IPA                              Partition 1                  05/23/2001 15:09:23   Page    12

 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  F0F1                                    =C'2001'           Compiled Year
 000004  F0F5  F2F3                                    =C'0523'           Compiled Date MMDD
 000008  F1F5  F0F9  F1F1                              =C'150911'         Compiled Time HHMMSS
 00000E  F0F1  F0F2  F0F0                              =C'010200'         Compiler Version
                           Timestamp and Version End

15694A01 V1 R2 z/OS C/C++ IPA                              Partition 1: main            05/23/2001 15:09:23   Page    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

 000018                    00010 |    1   main     DS    0D
 000018  47F0  F022        00010 |    1            B     34(,r15)
 00001C  01C3C5C5                                        CEE eyecatcher
 000020  00000098                                        DSA size
 000024  00000098                                        =A(PPA1-main)
 000028  47F0  F001        00010 |    1            B     1(,r15)
 00002C  58F0  C31C        00010 |    1            L     r15,796(,r12)
 000030  184E              00010 |    1            LR    r4,r14
 000032  05EF              00010 |    1            BALR  r14,r15
 000034  00000000                                        =F'0'
 000038  07F3              00010 |    1            BR    r3
 00003A  90E4  D00C        00010 |    1            STM   r14,r4,12(r13)
 00003E  58E0  D04C        00010 |    1            L     r14,76(,r13)
 000042  4100  E098        00010 |    1            LA    r0,152(,r14)
 000046  5500  C314        00010 |    1            CL    r0,788(,r12)
 00004A  4130  F03A        00010 |    1            LA    r3,58(,r15)
 00004E  4720  F014        00010 |    1            BH    20(,r15)
 000052  5000  E04C        00010 |    1            ST    r0,76(,r14)
 000056  9210  E000        00010 |    1            MVI   0(r14),16
 00005A  50D0  E004        00010 |    1            ST    r13,4(,r14)
 00005E  18DE              00010 |    1            LR    r13,r14
 000060                    End of Prolog
```

Figure 19. Example of an IPA Link Step Listing (Part 5 of 7)

```
 000060  180F             00008 |    3  +         LR    r0,r15
 000062  8900  0001       00008 |    3  +         SLL   r0,1
 000066  181F             00008 |    3  +         LR    r1,r15
 000068  B252  0010       00008 |    3  +         MSR   r1,r0
 00006C  180F             00008 |    3  +         LR    r0,r15
 00006E  B252  0001       00008 |    3  +         MSR   r0,r1
 000072  181F             00008 |    3  +         LR    r1,r15
 000074  B252  0010       00008 |    3  +         MSR   r1,r0
 000078  180F             00008 |    3  +         LR    r0,r15
 00007A  B252  0001       00008 |    3  +         MSR   r0,r1
 00007E  181F             00008 |    3  +         LR    r1,r15
 000080  B252  0010       00008 |    3  +         MSR   r1,r0
 000084  180F             00008 |    3  +         LR    r0,r15
 000086  B252  0001       00008 |    3  +         MSR   r0,r1
 00008A  181F             00008 |    3  +         LR    r1,r15
 00008C  B252  0010       00008 |    3  +         MSR   r1,r0
 000090  180F             00008 |    3  +         LR    r0,r15
 000092  B252  0001       00008 |    3  +         MSR   r0,r1
 000096  B252  00F0       00008 |    3  +         MSR   r15,r0
 00009A                   00022 |    1   @1L20    DS    0H

 00009A                   Start of Epilog
 00009A  180D             00023 |    1            LR    r0,r13
 00009C  58D0  D004       00023 |    1            L     r13,4(,r13)
 0000A0  58E0  D00C       00023 |    1            L     r14,12(,r13)
 0000A4  9824  D01C       00023 |    1            LM    r2,r4,28(r13)
 0000A8  051E             00023 |    1            BALR  r1,r14
 0000AA  0707             00023 |    1            NOPR  7

                          ***   General purpose registers used: 1101100000001111
                          ***   Floating point  registers used: 0000000000000000
                          ***   Size of register spill area: 128(max) 0(used)
                          ***   Size of dynamic storage: 15215694A01 V1 R2 z/OS C/C++ IPA                                Partition 1: main

 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

                          ***   Size of executable code: 148

 0000AC  0000  0000
15694A01 V1 R2 z/OS C/C++ IPA                                Partition 1                  05/23/2001 15:09:23   Page    15

 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
 0000B0  1CCEA106                                      =F'483303686'      Flags
 0000B4  000000D8                                      =A(PPA2-main)
 0000B8  00000000                                      =F'0'              No PPA3
 0000BC  00000000                                      =F'0'              No EPD
 0000C0  FE000000                                      =F'-33554432'      Register save mask
 0000C4  00000000                                      =F'0'              Member flags
 0000C8  90                                            =AL1(144)          Flags
 0000C9  000000                                        =AL3(0)            Callee's DSA use/8
 0000CC  0040                                          =H'64'             Flags
 0000CE  0012                                          =H'18'             Offset/2 to CDL
 0000D0  00000000                                      =F'0'              Reserved
 0000D4  5000004A                                      =F'1342177354'     CDL function length/2
 0000D8  FFFFFF68                                      =F'-152'           CDL function EP offset
 0000DC  38240000                                      =F'941883392'      CDL prolog
 0000E0  40090041                                      =F'1074331713'     CDL epilog
 0000E4  00000000                                      =F'0'              CDL end
 0000E8  0004  ****                                    AL2(4),C'main'
                          PPA1 End

                          PPA2: Compile Unit Block
 0000F0  0300  2202                                    =F'50340354'       Flags
 0000F4  FFFF  FF10                                    =A(CEESTART-PPA2)
 0000F8  0000  0000                                    =F'0'              No PPA4
 0000FC  FFFF  FF10                                    =A(TIMESTMP-PPA2)
 000100  0000  0000                                    =F'0'              No primary
 000104  0000  0000                                    =F'0'              Flags
                          PPA2 End
```

Figure 19. Example of an IPA Link Step Listing (Part 6 of 7)

```
15694A01 V1 R2 z/OS C/C++ IPA                          Partition 1                 05/23/2001 15:09:23   Page    16

                              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   000108           @STATICP
                        LD    0 000018   000001           main
                        ER    2 000000                    CEESG003
                        ER    3 000000                    CEESTART
                        SD    4 000000   000008           @@PPA2
                        SD    5 000000   00000C           CEEMAIN
                        ER    6 000000                    EDCINPL
15694A01 V1 R2 z/OS C/C++ IPA                          Partition 1                 05/23/2001 15:09:23   Page    17

                            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
          main                                          main
          CEESG003                                      CEESG003
          CEESTART                                      CEESTART
          @@PPA2                                        @@PPA2
          CEEMAIN                                       CEEMAIN
          EDCINPL                                       EDCINPL
15694A01 V1 R2 z/OS C/C++ IPA                          Partition 1                 05/23/2001 15:09:23   Page    18

                              * * * * *   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     TSIPA.TEST.C(INCLMAIN)
                                     - Compiled by 5694A01 V1 R2 z/OS C
                                                on 05/23/2001 15:09:11
      P          3           2     TSIPA.TEST.C(INCLRTN1)
                                     - Compiled by 5694A01 V1 R2 z/OS C
                                                on 05/23/2001 15:09:15
      P          4           3     TSIPA.TEST.C(INCLRTN2)
                                     - Compiled by 5694A01 V1 R2 z/OS C
                                                on 05/23/2001 15:09:19

  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   * * * * *
15694A01 V1 R2 z/OS C/C++ IPA                          Partition 1                 05/23/2001 15:09:23   Page    19

                              * * * * *   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 19. Example of an IPA Link Step Listing (Part 7 of 7)

IPA Link Step Listing Components

The following sections describe 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, ARCHITECTURE). 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.

HFS 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.

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 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 option LIST 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 121, "IPA | NOIPA" on page 133, and "LIST | NOLIST" on page 150 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 option (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 115, and *z/OS C/C++ Messages*.

Message Summary

This listing section displays the total number of messages and the number of messages for each severity level.

Chapter 6. Binder Options and Control Statements

This chapter describes only 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 DFSMS Program Management*.

Binder Options

The binder processes options from left to right. If you specify a binder option more than once, the binder uses the last, or rightmost option. The default options used by the z/OS C/C++ supplied cataloged procedures, the `CXXBIND` REXX exec, and the `c89`, `cc`, and `c++` utilities are shown only where they differ from the binder default.

There are two option formats supported by the binder: `KEYWORD=OPTION` and `KEYWORD(OPTION)`. Both formats are supported for Batch and TSO. `c89` only supports the `KEYWORD=OPTION` format.

ALIASES(ALL | NO)

Table 23. ALIASES Default

Binder Default	Batch	TSO (CXXBIND)	z/OS UNIX System Services Utilities - c89/cc/c++
`ALIASES=NO`			

The `ALIASES(ALL)` option instructs the binder to create hidden aliases for all externally defined symbols (functions and variables). Hidden aliases are marked as ″not executable″, to prevent an unintentional load and execution. These aliases might not be visible to some system utilities. Also, if the target of an `ALIAS` control statement is a symbol, the binder does not mark the alias as hidden.

The binder does not create hidden aliases if `ALIASES(NO)` is in effect, or if the module is saved in a PM2 or earlier format. PM2 format is available only for C NORENT NOLONGNAME compilations and programs that are first processed by the prelinker. See “COMPAT(PM1 | PM2 | PM3 | CURRENT | CURR)” on page 288 for information on setting the compatibility format.

Hidden aliases are for autocall purposes only. See “Generating Aliases for Automatic Library Call (Library Search)” on page 392.

AMODE

Table 24. AMODE Default

Binder Default	Batch	TSO (CXXBIND)	z/OS UNIX System Services Utilities - c89/cc/c++
`AMODE=24`	`AMODE=31`	`AMODE(31)`	`AMODE=31`

To assign the addressing mode for all the entry points into a program module (the main entry point, its true aliases, and all the alternate entry points), you should code the `AMODE` parameter as follows: `AMODE={24/31/ANY/MIN}`

Note: You cannot use `AMODE=24` for XPLink applications or DLLs.

CALL(YES | NO)

Table 25. CALL Default

Binder Default	Batch	TSO (CXXBIND)	z/OS UNIX System Services Utilities - c89/cc/c++
CALL=YES			CALL=NO if the -r flag is set. CALL=YES otherwise.

The CALL(YES) option specifies that the binder should search the libraries that are defined by the DD SYSLIB to find symbol definitions (see “Final Autocall Processing (SYSLIB)” on page 390).

The CALL(NO) option instructs the binder not to perform final autocall processing of the libraries that are defined by DD SYSLIB to resolve unresolved references.

CASE(UPPER | MIXED)

Table 26. CASE Default

Binder Default	Batch	TSO (CXXBIND)	z/OS UNIX System Services Utilities - c89/cc/c++
CASE=UPPER	CASE=MIXED	CASE(MIXED)	CASE=MIXED

The CASE option controls the sensitivity of the binder to case. When you specify CASE(MIXED):

- The binder distinguishes between uppercase characters and lowercase characters, and treats two strings as different if their cases do not match exactly.
- The binder does not convert lowercase characters to uppercase in names that are encountered in input modules, control statements, and call parameters.

When you specify CASE(UPPER), the binder converts all lowercase characters to uppercase during processing.

Note: z/OS C++ does not support the CASE(UPPER) option. Use CASE(MIXED) for C++ code.

COMPAT(PM1 | PM2 | PM3 | CURRENT | CURR)

Table 27. COMPAT Default

Binder Default	Batch	TSO (CXXBIND)	z/OS UNIX System Services Utilities - c89/cc/c++
COMPAT=CURRENT or COMPAT=PM2			COMPAT=CURRENT

The COMPAT option specifies the compatibility level of the binder.

COMPAT=CURRENT is the current level of the binder and allows you to use all features.

COMPAT=PM2 allows you to link-edit into a load module. The prelinker is required if any of the object files in the application use constructed reentrancy, use long names, are DLL or are C++. This option is required for modules which will be executed on OS/390 releases prior to OS/390 Version 2 Release 4.

Higher levels than PM2 do not require but can use the prelinker. Higher levels than PM2 normally require that you link into a program object or HFS file. You can link-edit into a load module if you ensure that none of the object files in the application use constructed reentrancy, use long names, are DLL or are C++.

c89/cc/c++ will default to `COMPAT=PM2` when:

- The prelinker is used during the link step.
- Link-editing without the prelinker, and the target library (according to {_PVERSION}) is lower than OS/390 Version 2 Release 4. You may encounter problems at link-edit or run-time if any of the object files in the application use constructed reentrancy, use long names, are DLL or are C++.

Otherwise, the default is `COMPAT=CURRENT`.

DYNAM(DLL | NO)

Table 28. DYNAM Default

Binder Default	Batch	TSO (CXXBIND)	z/OS UNIX System Services Utilities - c89/cc/c++
`DYNAM=NO`	`DYNAM=DLL`	`DYNAM(DLL)`	`DYNAM=DLL` `DYNAM=NO` if prelinker is active during link-edit

The `DYNAM` option specifies whether the binder should enable the resultant module for DLL-type dynamic binding. You must specify `DYNAM(DLL)` if the program object is to be a DLL or will need to load DLLs. If you specify `DYNAM(DLL)`, the binder does the following:

- Creates the Import/Export Table in section IEWBCIE of class B_IMPEXP. This element contains information about imported and exported symbols that is necessary to support run-time library dynamic linking and loading.
- Performs DLL-specific bind processing; that is, generates linkage areas (descriptors) in class C_WSA for run-time library fixup.

Import/export tables and the definition side-deck are not created if you specify `DYNAM(NO)`, or if it is in effect by default. If you specify `DYNAM(DLL)`, the binder `RES` option is disabled. DLL-enabled modules require PM3 program objects. If you attempt to save them in down-level program objects or load modules using `COMPAT`, the binder issues a severity 12 error, and does not save the module.

LET(0 | 4 | 8 | 12)

Table 29. LET Default

Binder Default	Batch	TSO (CXXBIND)	z/OS UNIX System Services Utilities - c89/cc/c++
`LET=4`			If `LET` is specified without a value, then you get `LET=8`.

The `LET` option specifies that a generated program object should be marked as executable even if the return code is not zero: for example, if symbols are unresolved. For example, `LET(4)` marks the generated program object as executable even if there are errors of severity 4 or less. `LET` is the equivalent of `LET(8)`.

LIST(OFF | STMT | SUMMARY | NOIMP | ALL)

Table 30. LIST Default

Binder Default	Batch	TSO (CXXBIND)	z/OS UNIX System Services Utilities - c89/cc/c++
LIST=SUMMARY			If -V is not specified, the default is LIST=NO. If -V is specified, then LIST=NOIMP. If LIST is specified without a value, then you get LIST=SUMMARY.

The LIST option specifies the type of information that is written to the binder map. Use one of the following suboptions:

ALL produces a listing of individual function calls, save summary, control statements, and messages

SUMMARY produces a listing of the summary information which includes processing options, module attributes, save summary, and the entry point summary, and echoes IMPORT control statements

NOIMP produces the same output as SUMMARY, but does not echo IMPORT control statements

STMT produces a listing of control statements and binder messages

OFF produces a listing that contains only binder messages

Note: The binder map contains a summary of the modules only if you specify the suboptions SUMMARY, ALL, or NOIMP.

NOLIST is equivalent to LIST(OFF).

MAP(YES | NO)

Table 31. MAP Default

Binder Default	Batch	TSO (CXXBIND)	z/OS UNIX System Services Utilities - c89/cc/c++
MAP=NO	MAP	MAP	MAP=YES when the -V flag is set. MAP=NO otherwise.

The option MAP(YES) instructs the binder to write a printed map of the program object to DD SYSPRINT. The option MAP(NO) specifies that the binder does not generate a map.

OPTIONS

This option specifies the DDname of a file that contains other options. For example, OPTIONS=OPT1 specifies that further options should be read from the DDname OPT1. This option is useful if the length of the PARM keyword in your JCL is longer that 100 characters.

REUS(NONE | SERIAL | RENT)

Table 32. REUS Default

Binder Default	Batch	TSO (CXXBIND)	z/OS UNIX System Services Utilities - c89/cc/c++
REUS=NONE	REUS=RENT	REUS(RENT)	REUS=SERIAL when the -g flag is set. REUS=RENT otherwise.

The REUS option specifies the reusability of the output program object. For C/C++ code the suboptions that you are most likely to use are the following:

RENT
: Specifies that other users or programs can share a read-only copy of the code.

NONE
: Specifies that the code cannot be shared. Use this option if you have NORENT variables which are modified during program execution. Such a program object cannot be in the LPA or ELPA.

  If you built a DLL with REUS(NONE), any program that links to the DLL will get a new load of both the code and data (C_WSA). This may be a problem if other DLLs in the same program share this DLL. See “Non-reentrant DLL Problems” on page 411.

SERIAL
: Specifies that the code is loaded into modifiable storage and that it can be reused without being reloaded. Simultaneous use of the code, for example by more than one user or more than one thread, is not supported unless explicitly allowed for by the programmer. Use this option if you have NORENT variables that are modified during program execution. Such a program object cannot be in the LPA or ELPA.

  If you built a DLL with REUS(SERIAL), any program within a single Language Environment enclave that links to that DLL will share the same code and data (C_WSA).

RMODE

Table 33. RMODE Default

Binder Default	Batch	TSO (CXXBIND)	z/OS UNIX System Services Utilities - c89/cc/c++
RMODE=24			RMODE=ANY

To assign the residence mode for all the entry points into a program module, you can code the RMODE parameter as follows: RMODE={24/ANY/SPLIT}

UPCASE(YES | NO)

Table 34. UPCASE Default

Binder Default	Batch	TSO (CXXBIND)	z/OS UNIX System Services Utilities - c89/cc/c++
UPCASE=NO			

The UPCASE option specifies that some additional rename processing is to be done. You should not confuse this option with the CASE(UPPER) option.

UPCASE by itself is equivalent to UPCASE(YES). The UPCASE(YES) option enforces the uppercase mapping of some symbol names. See “Rename Processing” on page 391 for its effect.

If you use the UPCASE option, external symbols in C programs are no longer case-sensitive. The binder does not support the use of the UPCASE option with C++ code. Therefore, you should use the RENAME control statement rather than the UPCASE option.

XREF(YES | NO)

Table 35. XREF Default

Binder Default	Batch	TSO (CXXBIND)	z/OS UNIX System Services Utilities - c89/cc/c++
XREF=NO			XREF=YES when the -V flag is set. XREF=NO otherwise.

In the z/OS UNIX System Services environment, the c89, cc, and c++ utilities specify XREF=YES when you use the -V flag.

The XREF(YES) option instructs the binder to generate a cross-reference list of data variables. If the XREF(NO) option is in effect, the binder does not generate a cross-reference list of data variables.

Binder Control Statements

Binder control statements specify how the binder processes its input.

The important binder control statements for a C/C++ programmer are the following (this is not a complete list):

- AUTOCALL
- ENTRY
- INCLUDE
- IMPORT
- LIBRARY
- NAME
- RENAME

You can place the control statements in a permanent data set that has the attributes RECFM=F or RECFM=FB, and LRECL=80.

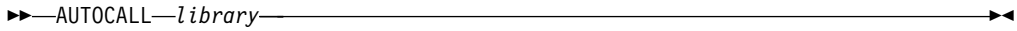
If all of the information does not fit on one control statement, you can use one or more continuations. You must put a non-blank character in column 72 if you need to continue a control statement on the next record. The first column of the continued card that follows must be blank, and the statement must continue in column 2. The binder ignores leading blanks unless they are in a quoted string. You may optionally enclose a named token in single quotes.

You can specify input files on the INCLUDE, LIBRARY, and AUTOCALL statements as HFS pathnames rather than DD names. Pathnames can be distinguished from DD names by the preceding ″/″, which indicates an absolute pathname, or ″./″, which indicates a relative pathname.

AUTOCALL Control Statement

The AUTOCALL control statement causes the binder to perform an immediate (incremental) library search on the named library. Incremental autocall attempts to resolve any unresolved symbols at this point in the processing, using a single library or library concatenation. The binder searches the library before it processes more primary or secondary input.

The AUTOCALL control statement has the following syntax:



library If *library* identifies the DD name of the library or library concatenation, it cannot exceed 8 bytes in length.

If *library* identifies an HFS filename, it cannot exceed 1024 bytes. The binder assumes that the file is an archive file. If it is an HFS directory file, then for purposes of symbol resolution, the binder uses the filenames of the files in the directory in the same way as it uses PDSE aliases and member names.

During incremental autocall, the binder ignores LIBRARY control statements and the CALL option.

ENTRY Control Statements

The ENTRY control statement specifies the entry point for program execution.

The ENTRY control statement has the following syntax:

name The name of the entry point for execution when the program is loaded.

By default, the program entry point for a C or C++ application is CEESTART. The program entry point is nominated in one of three ways (listed from weakest to strongest nomination).

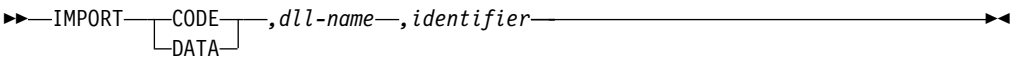
1. The name of the first section that is processed by the binder
2. The name that is nominated in the object module (CEESTART for C/C++ main())
3. The name explicitly specified on an ENTRY control statement

IMPORT Control Statements

The IMPORT control statement describes an external function or variable to be imported, and the name of the DLL that contains its definition. The DLL name can be a PDS or PDSE member, or an HFS filename. The function or variable should be one that is being exported by a DLL.

If you do not specify DYNAM(DLL), the binder ignores the IMPORT control statement.

The IMPORT control statement has the following syntax:



`CODE | DATA` Specifies the type of contents of the module that the imported symbol represents. A function and a variable cannot have the same name.

dll-name The directory name (primary member or alias) or HFS filename of the load module or program object that contains the imported function or variable. The maximum length of a dll-name is 1024 characters. The maximum length of an HFS filename is 255 bytes.

identifier The name of the symbol (function or variable) that is to be imported. The name cannot be longer than 1024 characters. If the symbol has a C++ mangled name, then you must use the mangled name on the `IMPORT` statement. If the identifier contains lowercase letters, you must specify the binder option `CASE(MIXED)`.

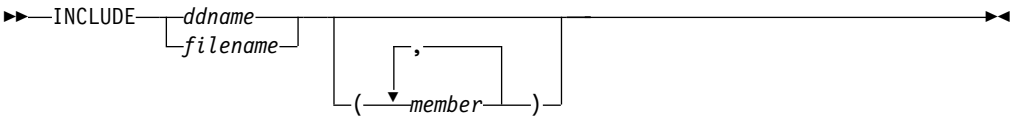
Typically, a DLL has an associated definition side-deck, containing `IMPORT` control statements, which was created by the binder when the DLL was created. You include this definition side-deck when you bind the application that imports functions or variables from that DLL. You can edit the records in the side-deck file, or substitute your own `IMPORT` control statements so that some symbols are imported from DLLs in a different library.

If your program exports symbols, the binder may also generate an output file of corresponding `IMPORT` control statements. See “Output IMPORT Statements” on page 394 .

INCLUDE Control Statements

You typically place `INCLUDE` control statements in DD `SYSLIN` to include multiple program objects, load modules, or object modules in primary input.

The `INCLUDE` control statement has the following syntax:



filename is the name of the file to be included.

ddname is a `DD` name associated with a file to be included.

member is the member of the `DD` to be included.

The binder attempts to read the file that is specified.

LIBRARY Control Statement

You can use the `LIBRARY` control statement to resolve conflicts that you cannot resolve by changing the order of libraries in the `SYSLIB` concatenation.

To specify that the binder should never search for an unresolved reference *neversrch*, use the following syntax for the `LIBRARY` control statement:

neversrch The binder never searches for the reference that you marked as *neversrch*, on this bind step or on future rebinds.

To specify that the binder should not search for an unresolved reference *nosrch*, use the following syntax for the `LIBRARY` control statement:

nosrch An external reference which may be unresolved at the end of `SYSLIN` processing. Automatic library call in `SYSLIB` does not search for such references on this bind step. Case-sensitivity is maintained you enclose *nosrch* in single quotes.

To direct the binder to search for an unresolved reference *srch* in a particular library, use the following syntax of the `LIBRARY` control statement:

ddname The name of a `DD` that defines a library (PDS or PDSE), or a concatenation of one or more PDS or PDSEs.

srch An external reference which may be a variable or a function. Should this symbol be unresolved after `SYSLIN` is processed, and library search is requested, the libraries pointed to by `SYSLIB` are not searched. Rather, the library (PDS or PDSE) that is defined by `ddname` will be searched for an alias or a member of name *srch*. See “Generating Aliases for Automatic Library Call (Library Search)” on page 392. If the binder finds the member, it reads it as input to the bind step. If you enclose *srch* in single quotes, the search is case-sensitive.

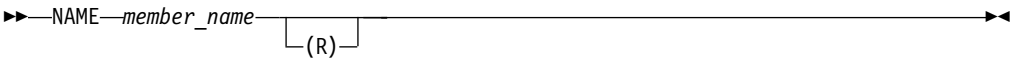
For example, if you have a program that has both Fortran and C code, both libraries define the member ABS and COS. You want the member COS from the Fortran library, and the member ABS from the C library. Your `LIBRARY` control statement would be similar to the following:

```
LIBRARY DDFORT(COS)
LIBRARY DDCLIB(ABS)
```

If you do not use the `LIBRARY` control statement, you will get both members from the C library or both members from the Fortran library.

NAME control statement

The `NAME` control statement specifies the name of the program object that is output to SYSLMOD. The `NAME` control statement has the following syntax:



member_name A PDS or PDSE library member name, or an HFS file name.

R If you use the option R and the name that you specify already exists, the binder will replace the existing member with the output program object.

The output from the binder can be a single program object, or multiple program objects generated by using multiple NAME control statements.

RENAME Control Statement

The RENAME control statement requests the binder to rename the references to a symbol that remains unresolved at the end of the first pass of final autocall processing of SYSLIB. See “Final Autocall Processing (SYSLIB)” on page 390.

You can use the RENAME control statement to resolve case differences in function names.

The RENAME control statement has the following syntax:

old-name The function to be renamed. Maximum length is 1024.

new-name The name to which old-name may be changed. Maximum length is 1024.

When the binder reads a RENAME control statement, it adds the request to the list of such requests. Nothing else is done until rename processing. See “Rename Processing” on page 391.

Chapter 7. Run-Time Options

This chapter describes how to specify run-time options and #pragma runopts preprocessor directives available to you with z/OS C/C++ and z/OS Language Environment. For a detailed description of the z/OS Language Environment run-time options and information about how to apply them in different environments, refer to the *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, Language 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, Language Environment interprets the entire string as an argument.

If EXECOPS is not in effect, Language 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 z/OS Language Environment, refer to the *z/OS Language Environment Programming Reference*.

Using the #pragma runopts Preprocessor Directive

You can use the #pragma runopts preprocessor directive to specify z/OS Language Environment run-time options. You can also use #pragma runopts to specify the run-time options ARGPARSE, ENV, PLIST, REDIR, and EXECOPS, which having matching compiler options. If you specify the compiler option, it has precedence over the #pragma runopts directive.

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.

The `#pragma runopts` directive can appear in any file: main, include, or source. 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.

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 the *C/C++ Language Reference*.

Part 3. Compiling, Binding, and Running z/OS C/C++ Programs

This part describes how to compile, bind, and run z/OS C/C++ programs using z/OS Language Environment in the following sections:

Chapter 8. Compiling

This chapter describes how to compile your program with the z/OS C/C++ compiler and z/OS Language Environment. For specific information about compiler options see “Chapter 5. Compiler Options” on page 61.

The z/OS C/C++ compiler analyzes the source program and translates the source code into machine instructions that are known as *object code*.

You can perform regular compilations under z/OS batch, TSO, or the z/OS shell.

Input to the z/OS C/C++ Compiler

The following sections describe how to specify input to the z/OS C/C++ compiler for a regular compilation, or the IPA Compile step. For information about input for the IPA Link step, refer to “Chapter 9. Using the IPA Link Step with z/OS C/C++ Programs” on page 339.

If you are compiling a C or C++ program, input for the compiler consists of the following:

- Your z/OS C/C++ source program
- The z/OS C/C++ standard header files including IBM-supplied Class Library header files
- Your header files

When you invoke the z/OS C/C++ compiler, the operating system locates and runs the compiler. To run the compiler, you need these default data sets supplied by IBM:

- `CBC.SCCNCMP`
- `CEE.SCEERUN`

The locations of the compiler and the runtime 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 “Appendix D. Cataloged Procedures and REXX EXECs” on page 551.

HFS file names: Unless they appear in JCL, file names which contain the special characters blank, backslash, and double quote must escape these characters. The escape character is backslash (\).

Primary Input

For a C or C++ program (except for the IPA Link step), the primary input to the compiler is the data set that contains your 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 hierarchical file system (HFS) file
- All HFS files in a directory

Secondary Input

For a C or C++ program (except for the IPA Link step), secondary input to the compiler consists of data sets that contain `#include` files. Use the `LSEARCH` and `SEARCH` compiler options to specify the location of the `#include` files.

For more information on the use of these compiler options, see "LSEARCH | NOLSEARCH" on page 155 and "SEARCH | NOSEARCH" on page 187. For more information on naming `#include` files, see "Specifying Include File Names" on page 327. For information on how the compiler searches for `#include` files, see "Search Sequences for Include Files" on page 334. For more information on include files, refer to "Using Include Files" on page 326.

Output from the Compiler

You can specify compiler output files as one or more of the following:

1. A sequential data set
2. A member of a partitioned data set
3. A partitioned data set
4. A hierarchical file system (HFS) file
5. An HFS directory

For valid combinations of input file types and output file types, refer to Table 38 on page 305.

Specifying Output Files

You can use compile options to specify compilation output files as follows:

Table 36. Compile Options That Provide Output File Names

Output File Type	Compiler Option
Object Module	OBJECT(filename)
Listing File	SOURCE (filename), LIST(filename), INLRPT(filename)
Preprocessor Output	PPONLY(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 the ddnames are not allocated, the output file names are generated based on the name of the source file (except in the case of 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 37 on page 303 shows.

For 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
```

If you compile source from HFS files without specifying output filenames 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 defaults are used:

Table 37. Defaults for Output File Types

Output File Type.	**z/OS File**	**HFS 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 HFS directory if the source resides in the HFS, or to the z/OS data set if the source resides in a data set.
2. If you have specified the `OE` option, see “OE | NOOE” on page 169 for a description of the default naming convention.
3. If you supply inline source in your JCL, you must provide a file name for the output, or route it to the job log. 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. The compiler will not automatically generate file names for output that is created by `#pragma` options.

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 150.

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 166.

Differences in Object Modules under IPA: The object module that a regular compilation generates is different from the object module that the IPA Compile step generates. The IPA Compile step and regular compilation both produce an object module for each source file successfully processed. For the IPA Compile step, however, the output is an IPA-optimized object file, or a combined IPA/conventional object file (if you do not specify the `NOOBJECT` suboption of the `IPA` compiler option). You can use the object file that the `IPA(NOLINK,NOOBJECT)` compiler option creates as input to the IPA Link step only. It contains an external reference to `@@DOIPA`, which remains unresolved until IPA Link step processes the file. If you attempt to bind an IPA object file that was created by using the `IPA(NOLINK,NOOBJECT)` option, the binder issues an error message.

Refer to "Valid Input/Output File Types" 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 179.

Template Instantiation Output

If you specify *location*, which is either an HFS directory or a PDS, 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 instantiation files. See "TEMPLATEREGISTRY | NOTEMPLATEREGISTRY" on page 207 for more information on this default.

If you specify *location*, which is either an HFS 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" on page 205 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 38. 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 HFS File, for example `a/b/c.o`**	**Output Specified as HFS Directory, for example `a/b`**
Sequential Data Set, for example `A.B`	1. If file exists as a sequential data set, overwrites it 2. If file does not exist, creates sequential data set 3. Otherwise compilation fails	1. If PDS does not exist, creates the PDS and adds a member into the data set 2. If PDS exists and member does not exist, adds member in the PDS 3. If 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 file does not exist, creates PDS and member	1. If PDS does not exist, creates PDS and member 2. If PDS exists and member does not exist, adds member 3. If PDS exists and member also exists, overwrites the member	1. If directory does not exist, compilation fails 2. If directory exists and the file with the specified filename does not exist, creates file 3. If the directory exists and the file exists, overwrites file	1. If directory does not exist, compilation fails 2. If directory exists and the file with the filename *MEMBER.ext* does not exist, creates file 3. If directory exists and the file with the filename *MEMBER.ext* also exists, overwrite file
All members of a PDS, for example `A.B`	1. If file exists as a PDS, creates or overwrites members 2. If file does not exist, creates PDS and members 3. Otherwise compilation fails	Not Supported	Not Supported	1. If directory does not exist, compilation fails 2. If directory exists and the files with the filenames *`MEMBER.ext`* do not exist, creates files 3. If directory exists and the files with the filenames *`MEMBER.ext`* exist, overwrites files

Table 38. 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 HFS File, for example a/b/c.o	Output Specified as HFS Directory, for example a/b
HFS file, for example /a/b/d.c	1. If file exists as a sequential data set, overwrites file 2. If file does not exist, creates sequential data set 3. Otherwise compilation fails	1. If PDS does not exist, creates the PDS and stores a member into the data set 2. If PDS exists and member does not exist, then add the member in the PDS 3. If 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
HFS 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 z/OS C/C++ source program under z/OS batch, you can either use cataloged procedures that IBM supplies, or write your own JCL statements.

Using Cataloged Procedures for z/OS C

You can use one of the following IBM-supplied cataloged procedures. Each procedure includes a compilation step to compile your program.

`EDCC` Compile

`EDCCB` Compile and bind

`EDCCBG` Compile, bind, and run

`EDCI` Run the IPA Link step

`EDCCLIB`
Compile and maintain an object library

`EDCCL` Compile and link-edit

`EDCCPLG`
Compile, prelink, link-edit, and run

EDCCLG Compile, link-edit, and run

EDCXCB Compile and bind an XPLINK C program

EDCXCBG
Compile, bind, and run an XPLINK C program

EDCXI Run the IPA Link step for an XPLINK compiled program

IPA Considerations

Only the EDCC procedure applies to the IPA Compile step. Only the EDCI and EDCXI procedures apply to the IPA Link step.

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.

Using Cataloged Procedures for z/OS 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
CBCCB	Compile and bind
CBCCBG	Compile, bind, and run
CBCI	Run the IPA Link step
CBCCL	Compile, prelink, and link
CBCCLG	Compile, prelink, link, and run
CBCXCB	Compile and bind an XPLINK C++ program
CBCXCBG	Compile, bind, and run an XPLINK C++ program
CBCXI	Run the IPA Link step for an XPLINK compiled program

See “Appendix D. Cataloged Procedures and REXX EXECs” on page 551 for more information on cataloged procedures.

IPA Considerations

Only the CBCC procedure applies to the IPA Compile step. Only the CBCI and CBCXI procedures apply to the IPA Link step.

To run the IPA Compile step, use the CBCC 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.

Using Special Characters

When invoking the compiler directly, if a string contains a single quote (') it should be written as two single quotes (") as in:

```
//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 quotes (""), as follows:

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

A backslash need not precede special characters in HFS 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 HFS file names that you use in the PARM statement. For example:

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

Using Your Own JCL

The following example shows sample JCL for compiling a C program:

```
//jobname    JOB   acctno,name...
//COMPILE    EXEC  PGM=CCNDRVR,
// PARM='/SEARCH(''CEE.SCEEH.+'') NOOPT SO OBJ OPTFILE(DD:CPATH)'
//STEPLIB    DD    DSNAME=CEE.SCEERUN,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=...
//SYSUT4     DD    DSN=...
   :

//*
```

Figure 20. JCL for Compiling a C Program (for NOOPT, SOURCE, and OBJ)

The following example shows sample JCL for compiling a 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=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=...
 //SYSUT4   DD    DSN=...
   ⋮

//*
```

Figure 21. JCL for Compiling a C++ Program (for NOOPT, SOURCE, and OBJ)

Use JCL to define your jobs and job steps to the operating system. Describe the steps you want the operating system to perform, and specify the resources that are required by the job. The JCL statements that are essential for running a job are:

- A `JOB` statement that identifies the start of the job
- An `EXEC` statement that identifies a job step and the program to be executed either directly or by a cataloged procedure
- `DD` (data definition) statements, which identify the input/output facilities, that the program executed in the job step requires
- JES control statements that provide information to the Job Entry Subsystem

For more information about JCL, refer to the publications that are listed in the *z/OS Information Roadmap*.

Specifying Source Files

For non-HFS files, use this format of the SYSIN JCL:

```
//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 an HFS directory for your output files.

For HFS files, use this format of the SYSIN JCL:

```
//SYSIN DD PATH='pathname'
```

You can specify compilation for a single file or all source files in an HFS directory, for example:

```
//SYSIN DD PATH='/u/david'
//* All files in the directory /u/david are compiled
```

Note: If you specify an HFS directory as your primary input, you must specify an HFS directory for your output files.

When you place your source code directly in the input stream, use the SYSIN DD statement as follows:

```
//SYSIN DD DATA,DLM=@@
```

rather than:

```
//SYSIN DD *
```

When you use the DD * convention, the first 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 the *z/OS Information Roadmap*.

Specifying Include Files

Use the SEARCH option to specify system include files, and the LSEARCH option to specify your include files. For example:

```
//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 the *z/OS 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 an HFS file as follows:

```
//    CPARM='LIST(./listings/member1.lst)'
```

You can also use DD statements to specify output file names.

To specify non-HFS files, use DD statements with the DSNAME parameter. For example:

```
//SYSLIN DD DSN=USERID.TEST.OBJ(HELLO),DISP=SHR
```

To specify HFS directories or files, use DD statements with the PATH parameter.

```
//SYSLIN DD PATH='/u/david/test.o',PATHOPTS=(OWRONLY,OCREAT,OTRUNC)
```

on `PATH` and `PATHOPTS` parameters.

Note: Use the `PATH` and `PATHOPTs` parameters when specifying HFS 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. These are the two situations in which 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 C/C++ compiler under TSO in any of the following ways:

- Foreground execution from TSO READY
- Foreground execution from the ISPF command line or the ISPF menu option 6
- Foreground execution from ISPF menu option 4
- Foreground execution from an ISPF edit session
- Background execution (batch) from ISPF menu option 5

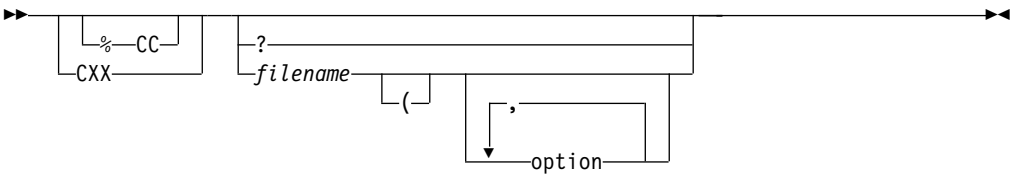
All methods of foreground execution call the `CC` or `CXX` REXX EXECs supplied by IBM.

Note: To run the compiler under TSO, you must have access to the runtime libraries. To ensure that you have access to the runtime library and compiler, 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.
- 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 C compiler, and the `CXX` REXX EXEC to invoke the z/OS C++ compiler. These REXX EXECs share the same syntax:



where

% ensures that the REXX EXEC `CC` is invoked, not the z/OS UNIX System Services `cc` utility.

option is any valid compiler option

filename can be one of the following:

1. A sequential data set
2. A member of a partitioned data set
3. All members of a partitioned data set
4. A hierarchical file system (HFS) file
5. All HFS files in a directory

If *filename* is not immediately recognizable as an HFS 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 an HFS file. For more information on file naming considerations refer to the *z/OS C/C++ Programming Guide*.

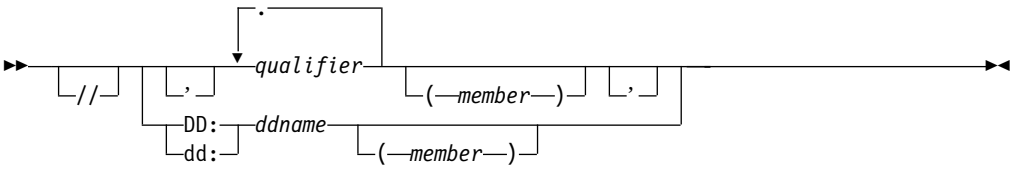
If you invoke either `CC` or `CXX` with no arguments or with only a single question mark, the appropriate preceding syntax diagram is displayed.

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 quote to indicating a fully qualified data set name, you must also use the trailing single quote.

Specifying HFS Files or Directories

You can use the CC or CXX REXX EXECs to compile source code that is stored in HFS files and directories. Use the following syntax when specifying HFS file or directory as your input or output file:

If you specify an HFS 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 quote, 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 HFS source file names. For example, the following input files are HFS 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 filename that does not include pathnames with single slashes, the compiler treats the file as a non-HFS file. The compiler treats the following input files as non-HFS files:

```
CXX hello.c
CC  //hello.c
```

Using Special Characters

When HFS file names contain the special characters blank, backslash, and double quote, you must precede the special character with a backslash(\).

When suboptions contain the special characters left bracket (, right bracket), comma, backslash, blank and double quote, you must precede these characters with a double backslash(\\) to ensure 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 (. 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 68 for more information on compiler options.

Using ISPF to Invoke the Compiler

Under TSO, you can use ISPF foreground and batch compile panels to start the z/OS C/C++ compiler. You can use online help with these panels.

Note: You cannot use ISPF to invoke the IPA Link step.

Foreground Processing

1. Select the `Foreground` option (4) from the `ISPF-PDF PRIMARY OPTION MENU`. The `FOREGROUND SELECTION PANEL` is presented.
2. Select the `C/C++` option (20). The `FOREGROUND UTILITIES` panel is presented, as shown in Figure 22.

```
------------------- FOREGROUND z/OS C/C++ UTILITIES ---------------------
COMMAND ===>


Select a function from the list below.
Enter either the selection number or the command name.

 1  CC        z/OS C Compiler
 2  CXX       z/OS C++ Compiler
```

Figure 22. Foreground IBM z/OS C/C++ Utility Panel

3. Select `1` to get the `FOREGROUND z/OS C COMPILE` panel, or select `2` to get the `FOREGROUND z/OS C++ COMPILE` panel as shown in Figure 23 on page 315.
4. Enter information such as your source data, password, object data set name, compiler options, and additional input libraries (as necessary).

```
------------------------- FOREGROUND z/OS C++ COMPILE ------------------
Command ===>

ISPF Library:          1
   Project ===> USERID
   Group   ===> DEV        ===>           ===>           ===>
   Type    ===> CXX
   Member  ===> *         (Blank or pattern for member selection list)
                          (* for entire PDS)

Other Partitioned or Sequential Data Set:
   Data Set Name  ===>             2

Data Set Password ===>          (If password protected)  3

Compiler Options:
            ===>                    4
            ===> SEARCH('CEE.SCEEH.+','CBC.SCLBH.+')
            ===> USERPATH(/USERID/DEV/INCL)
            ===>
            ===> OPT
            ===>
```

Figure 23. Foreground IBM z/OS C++ Compile Panel

1 The `ISPF Library` field is used if you do not specify a data set in **2**. Input to the compiler is either a member of an ISPF library, a member of a partitioned data set, or a sequential data set. Fill in the `Project`, `Group`, `Type`, and `Member` fields. To compile the entire PDS, place an asterisk (`*`) in the member field.

Note: If the source data set is partitioned and you did not specify a member, you are presented with a member list from which to choose the desired member.

2 Use the `Other Partitioned or Sequential Data Set` field if your input source is one of the following:

- a sequential data set
- a PDS with a number of qualifiers not equivalent to three

If you specify data sets in both **1** and **2**, the data set that you specified in this field is used. To compile an entire PDS of source instead of an individual PDS member, enter the PDS name followed by (*). You can specify a member of a PDS by entering the member name in parentheses after the data set name.

3 If any of your data sources are password protected, you must specify the password in the `Data Set Password` field.

4 Use the `Compiler Options` field, specify the compiler options that you want to use. For a complete list of compiler options, see “Compiler Option Defaults” on page 66.

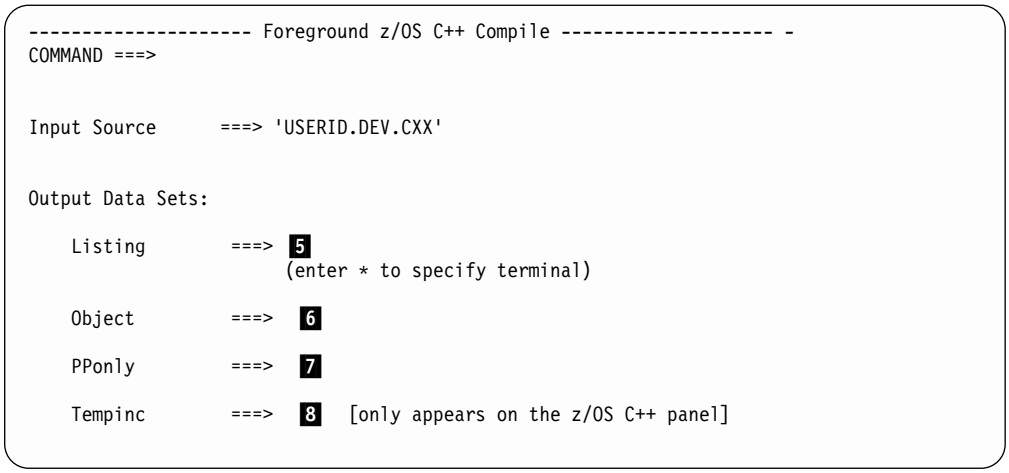


Figure 24. Foreground IBM z/OS C++ Compile Panel (2)

Note for z/OS C: The only difference in the appearance of the panels for z/OS C is in the heading, and the absence of the `Tempinc` option.

5. Enter names of the desired output data sets.

   5 Use the `Listing` field to specify a name for the listing data set. If you leave this field blank, the compiler generates a default name. See "Specifying Output Files" on page 302 for information on the defaults. To generate a listing data set, you must specify the compiler option `SOURCE` or `LIST` under `Compiler Options`.

   6 Use the `Object Data Set` field to specify a name for the object data set. If you leave this field blank, the compiler generates a default name. See "Specifying Output Files" on page 302 for information on the defaults.

   7 Use the `PPonly` field to specify a name for the PPONLY data set. If you leave this field blank, the compiler generates a default name. You can also specify the `LINES` or `COMMENTS` suboptions in this field. See "PPONLY | NOPPONLY" on page 179 for more information.

   8 For z/OS C++, use the `Tempinc` field to specify a PDS name for the template instantiation files. If you leave this field blank, the PDS is given the name `TEMPINC`.

6. Press `Enter` to invoke the foreground processing program.

HFS Note: You cannot use HFS files as input to the ISPF panels, but you can target your output to HFS files through the compiler options.

Batch Processing

Use the `batch` option to invoke the compiler as a batch job. JCL is generated for the job on the basis of the information you enter on the batch processing panels, and the job is submitted for execution.

When you choose the `batch` option from the `ISPF-PDF PRIMARY OPTION MENU`, the `BATCH SELECTION PANEL` is shown. Notice the `SOURCE DATA ONLINE` option and the `JOB STATEMENT INFORMATION` area at the bottom of this panel.

The `SOURCE DATA ONLINE` option specifies whether or not to check if the data set is available. If you specify `YES`, `ISPF` checks to see if the data set exists. If it does not

exist, you receive an ISPF message to indicate that the data set was not cataloged. If you specify NO, ISPF does not check for the data set.

The JOB STATEMENT INFORMATION consists of four lines for JCL card images. These lines are submitted as part of the batch job, so you must follow all the rules of JCL.

Alternatively, choose the IBM z/OS C/C++ Compiler option to show the BATCH IBM z/OS C/C++ COMPILE panels. These panels are similar to the FOREGROUND IBM z/OS C/C++ COMPILE panels. Most of the fields, such as the ISPF library, Other Partitioned, or Sequential Data Set, and Compiler Options behave the same way. See “Foreground Processing” on page 314 for descriptions.

The Batch option does not support passwords. If your input or output data sets are password protected, use the Foreground option. If you submit a job that includes a password-protected data set, the system operator is requested to enter the required password.

Use the Listing Data Set and SYSOUT Class fields to send the compiler list output directly to a SYSOUT queue or into a data set. If you fill in both fields, the value for SYSOUT Class is used.

Enter the source information and other parameters that this panel requires, and press <ENTER>. This generates the JCL, and submits the job.

If your system programmer has not provided a default search option for the C++ compiler, variable CBCCXOPT in CBC.SCCNUTL(CCNCCUST), or you want to modify it, you should enter it under Compiler Options. For example, "SEARCH('CEENEW.SCEEH.+')".

Compiling and Binding in the z/OS Shell

z/OS UNIX C/C++ programs with source code in HFS files or data sets must be compiled to create output object files residing either in HFS files or data sets.

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.

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 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 589.

Use the c89 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 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 utility supports IPA. For information on how to invoke the IPA Compile step from c89, refer to “Invoking IPA from the c89 Utility” on page 321.

You can also use the cc utility to compile a C application program from the z/OS shell. For more information, see “Appendix F. c89 — Compile, link-edit and assemble a z/OS C program and create an executable file” on page 577.

Use the c++ utility 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.

file.o is an object file.

file.a is an archive file.

libname is an archive library.

Another name for the c++ utility is cxx. The cxx utility and the c++ utility are identical. You can use cxx instead of c++ in all the examples that are shown in this section.

For a complete list of c++ options, and for more information on cxx, see “Appendix F. c89 — Compile, link-edit and assemble a z/OS C program and create an executable file” on page 577.

Note: You can compile and bind application program source and objects from within the shell using the c89 or c++ utility. If you use either 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 runs c89 and c++ for you.

For more information on using the make utility, see “Chapter 18. Archive and Make Utilities” on page 477 and *z/OS UNIX System Services Programming Tools*.

Compiling without Binding with c89/CC++

To compile source files without binding them, enter the c89 or c++ command 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 C/C++ source module is an HFS file, the object file is created in the working directory.
- If the z/OS 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 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 C/C++ source is located in a PDS member, you should specify double-quote characters around the qualified data set name. For example:

```
c89 -c "//'OLSEN.IPROGS.C(FILSER)'"
```

If the filename is not bracketed by quotes, 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 C application source to produce only object files. c89 recognizes that a file is a C source file by the .c suffix for HFS files, and the .C low-level qualifier for data sets. c89 recognizes that a file is an object file by the .o suffix for HFS files, and the .OBJ low-level qualifier for data sets.
  - To compile z/OS C source to create the default object file usersource.o in your working HFS directory, specify:

    ```
    c89 -c usersource.c
    ```
  - To compile z/OS 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 C++ application source to produce only object files. c++ recognizes that a file is a C++ source file by the .C suffix for HFS files, and the .CXX low-level qualifier for data sets. c++ recognizes that a file is an object file by the .o suffix for HFS files, and the .OBJ low-level qualifier for data sets.
  - To compile z/OS C++ source to create the default object file usersource.o in your working HFS directory, specify:

    ```
    c++ -c usersource.C
    ```
  - To compile z/OS C++ source to create an object file as a member in the PDS 'JONATHAN.APPROG.OBJ', specify:

    ```
    c++ -c "//'jonathan.approg.CXX(usersrc)'"
    ```
- Compiling and binding application source to produce an application executable file.
  - To compile an application source file to create the object file usersource.o in the HFS working directory and the executable file mymod.out in the /app/bin directory, specify:

    ```
    c89 -o /app/bin/mymod.out usersource.c
    ```
  - To compile the z/OS 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)'"
    ```

z/OS C++ Note:

To use the TSO utility OGET to copy a C++ HFS 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:

```
c++ -cV mypgm.C | awk '/^[^$]/ {print} /^$/ {printf "%s \n", $0}'
  > mypgm.lst
```

Compiling and Binding in One Step with c89 and c++ (or cxx)

To compile and bind a 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 to be created. The `c++` and `cxx` utilities are identical. You can use `cxx` instead of `c++` in all the examples that are shown in this section.

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 the prelinke is used.

- To compile and bind an application source file to create the default executable file `a.out` in the HFS working directory, specify:

```
c89 usersource.c
c++ 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` HFS 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
```

Building an Application with XPLINK from the c89 Utility

To build an application with XPLINK from the c89 utility you must use the XPLINK compiler option (i.e., −Wc,xplink) and the XPLINK link-edit 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.

Invoking IPA from the c89 Utility

You can invoke the IPA Compile step, the IPA Link step, or both from the c89 utility. The step that you invoke depends upon the invocation parameters and type of files specified. To invoke IPA, 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.

If you invoke the c89 utility by specifying the -c compiler option and at least one source file, c89 automatically specifies IPA(NOLINK) and automatically invokes the IPA Compile step. For example, the following command invokes the IPA Compile step for the source file hello.c:

```
c89 -c -WI,noobject hello.c
```

If you invoke c89 with at least one source file for compilation and any number of object files, and do not specify the -c option, c89 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. For example, the following command invokes the IPA Compile step and the bind while creating program foo:

```
c89 -o foo -WI,object foo.c
```

Refer to “Appendix F. c89 — Compile, link-edit and assemble a z/OS C program and create an executable file” on page 577 for more information about the c89 utility.

Specifying Options for the IPA Compile Step

When using the c89 utility, you can pass options to the IPA Compile step, as follows:

- You can pass IPA compiler option suboptions by specifying -WI,, followed by the suboptions.
- You can pass compiler options by specifying -Wc,, followed by the options.

Using IPA(OBJECTONLY) with the c89 Utility

A compilation using IPA(OBJONLY) is simply a standard non-IPA compilation with this option added. Do not use the -WI flag, as this would convert the compilation into an IPA Compile step.

For example, the following command results in an OPT(2) IPA(OBJONLY) compilation for the source file hello.c:

```
c89 -c -Wc,ipa\(objonly\) -2 hello.c
```

Using the make Utility

You can use the make utility to control the build of your z/OS UNIX System Services C/C++ applications. The make utility calls the c89 utility by default to compile and bind the programs that the previously created makefile specifies.

For 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 HFS files. To compile and bind z/OS C/C++ source files that are in data sets, you must use the `c89` utility directly.

See the *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 the *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 C/C++ program across compilation units, and to perform optimizations that are not otherwise available with the z/OS C/C++ compiler. You should refer to the *z/OS 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 batch or `c89` compilation process are noted throughout this chapter.

Figure 25 shows the flow of processing for a regular compilation:

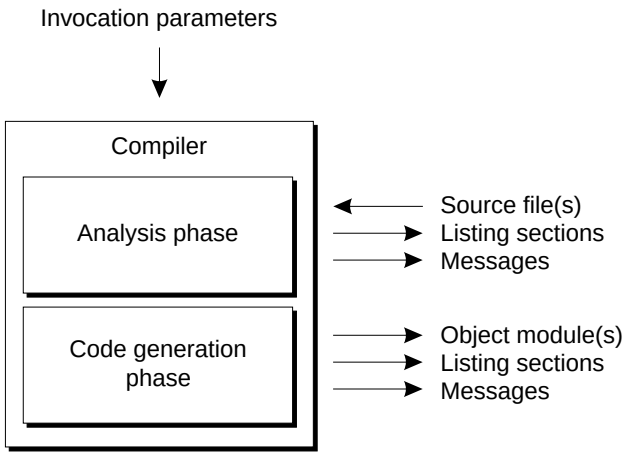


Figure 25. Flow of regular compiler processing

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. 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 26 shows the flow of IPA Compile step processing.

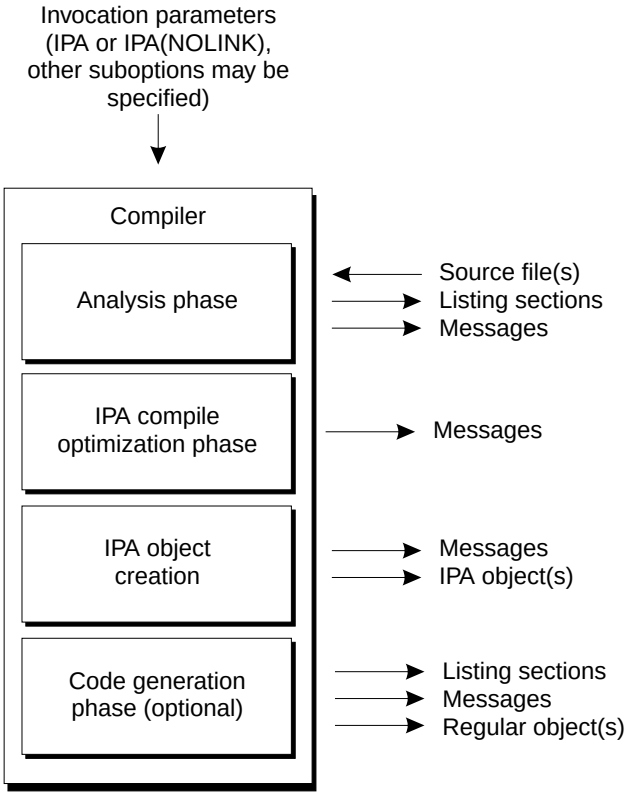


Figure 26. IPA Compile step processing

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. 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 27 on page 324 shows the flow of IPA Link step processing:

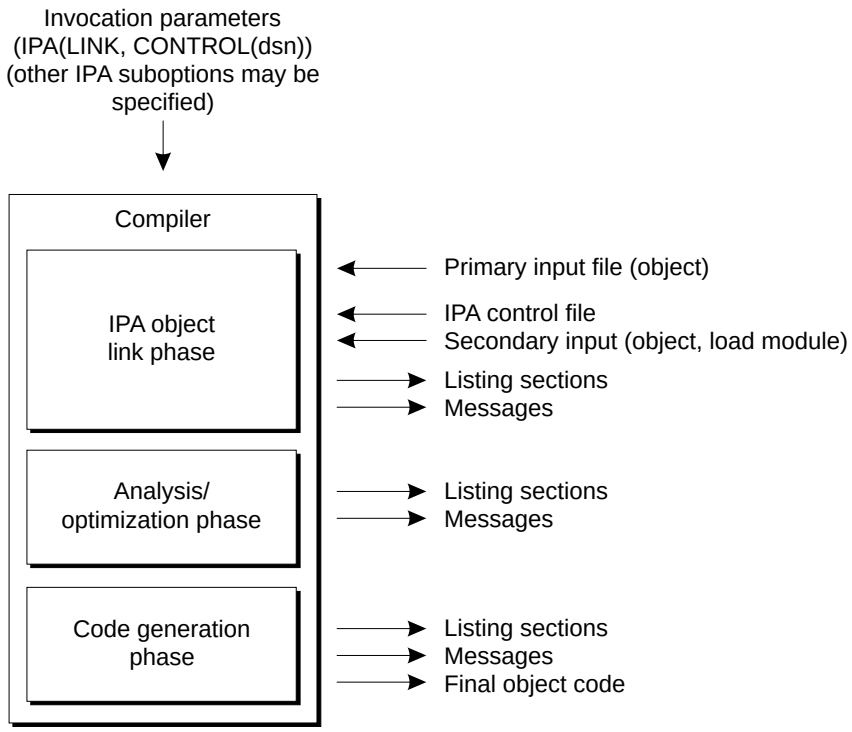


Figure 27. IPA Link step processing

Only `c89`, `c++` and z/OS batch (without the ISPF interface) support the IPA Link step. Refer to “Chapter 9. Using the IPA Link Step with z/OS C/C++ Programs” on page 339 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 28 on page 325 shows the flow of processing for an `IPA(OBJONLY)` compilation.

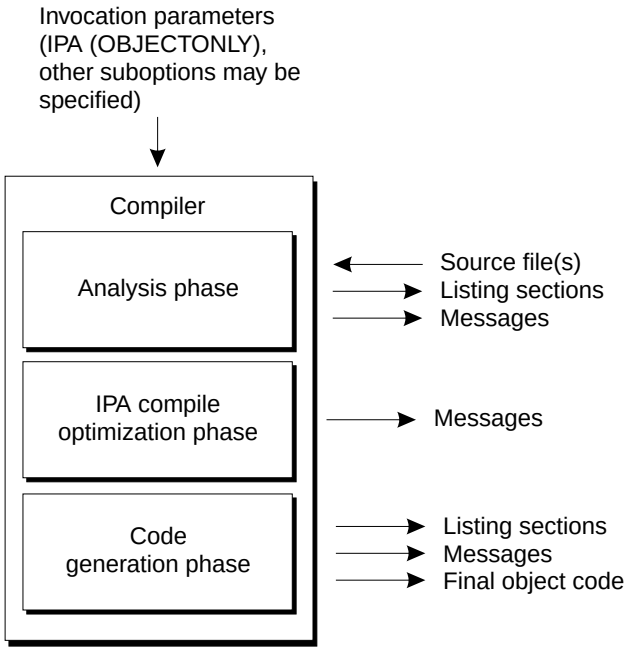


Figure 28. Compiling with IPA(OBJECTONLY)

Working With Object Files

S/390 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. For more information on object files, see “Object File Formats” on page 347.

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 files in an HFS file system, can be browsed using the PDF `obrowse` command. HFS 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.oo`

   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)″.

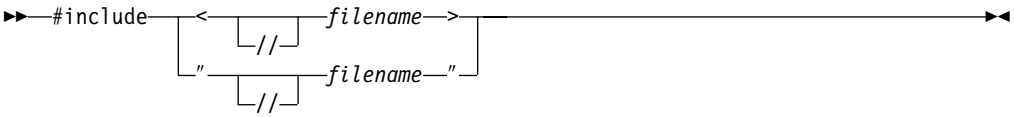
Using Feature Test Macros

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 the *C/C++ Language Reference* for a list of the feature macros). These macros are checked in the system header files.

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.

C/C++ Language Reference describes the `#include` directive. Its syntax is:



The angle brackets specify system include files, and double quotation marks specify user include files.

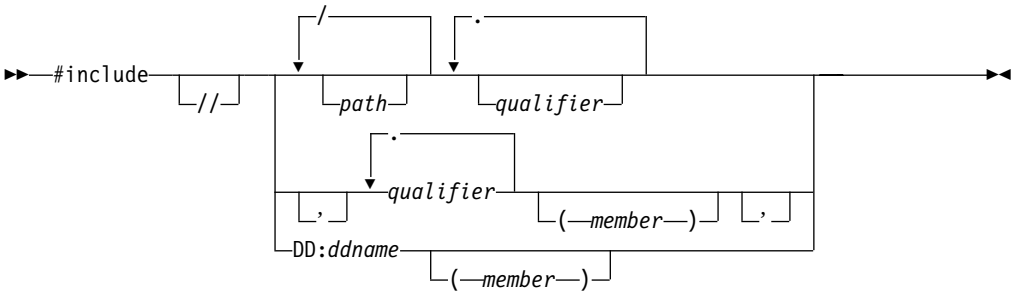
When you use the `#include` directive, you must be aware of the following:

- The *library search sequence*, the search order that C/C++ uses to locate the file. See “Search Sequences for Include Files” on page 334 for more information on the library search sequence.
- The file-naming conversions that C/C++ performs.
- The area of the input record that contains sequence numbers when you are including files with different record formats. See the *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 155 and “SEARCH | NOSEARCH” on page 187.

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-HFS file, hereafter called a data set.

Note:

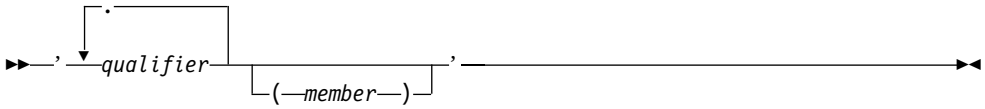
1. *filename* immediately follows the double slashes (//) without spaces.
2. Absolute data set names are specified by putting single quotation marks (') around the name. Refer to the above syntax diagram for this specification.
3. Absolute HFS filenames are specified by putting a leading slash (/) as 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 331 for information on absolute file names. When the compiler performs a library search, it treats *filename* as either an HFS file name or a data set name. This depends on whether the library being searched is HFS or MVS. If the compiler treats *filename* as an HFS 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:

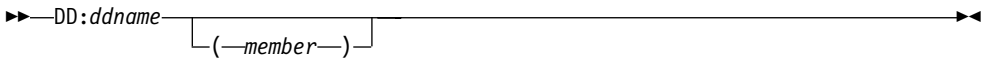
- For the first DSN format:

  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 hex 7c, viewed as an '@' (at sign) in code page 1047
- 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 hex 7c, viewed as an '@' (at sign) in code page 1047

- 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 hex 7c, viewed as an '@' (at sign) in code page 1047

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 *opt* 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

The leftmost qualifier of the *filename* in the #include directive is used when the *filename* is to be a ddname. For example:

Invocation:

```
SEARCH(DD:SYSLIB)
```

Include directive:

```
#include "sys/afile.g.h"
```

Resulting ddname:

```
DD:SYSLIB(AFILE)
```

In the above 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

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 "Specifying Sequential Data Sets and PDSs" on page 158 to understand how to specify sequential data sets. All *qualifier*s and periods (.) in *filename* are used for sequential data set specification. For example:

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

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 “Specifying Sequential Data Sets and PDSs” on page 158 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. For example:

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 +

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. For example:

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 39. 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 39. 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*.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:

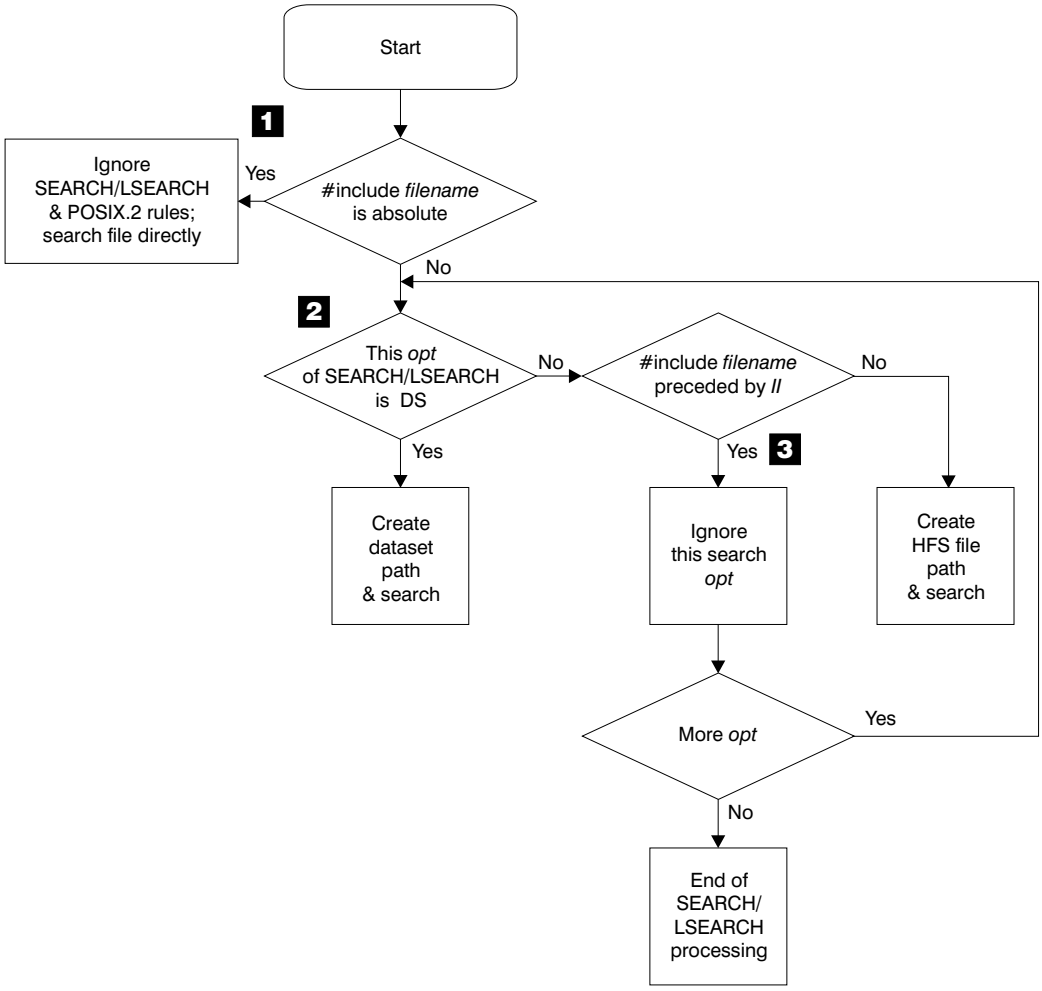


Figure 29. Overview of Include File Searching

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 30 on page 332 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 an HFS file search. The LSEARCH/SEARCH *opt* testing here is described in Figure 31 on page 333.

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 HFS 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:

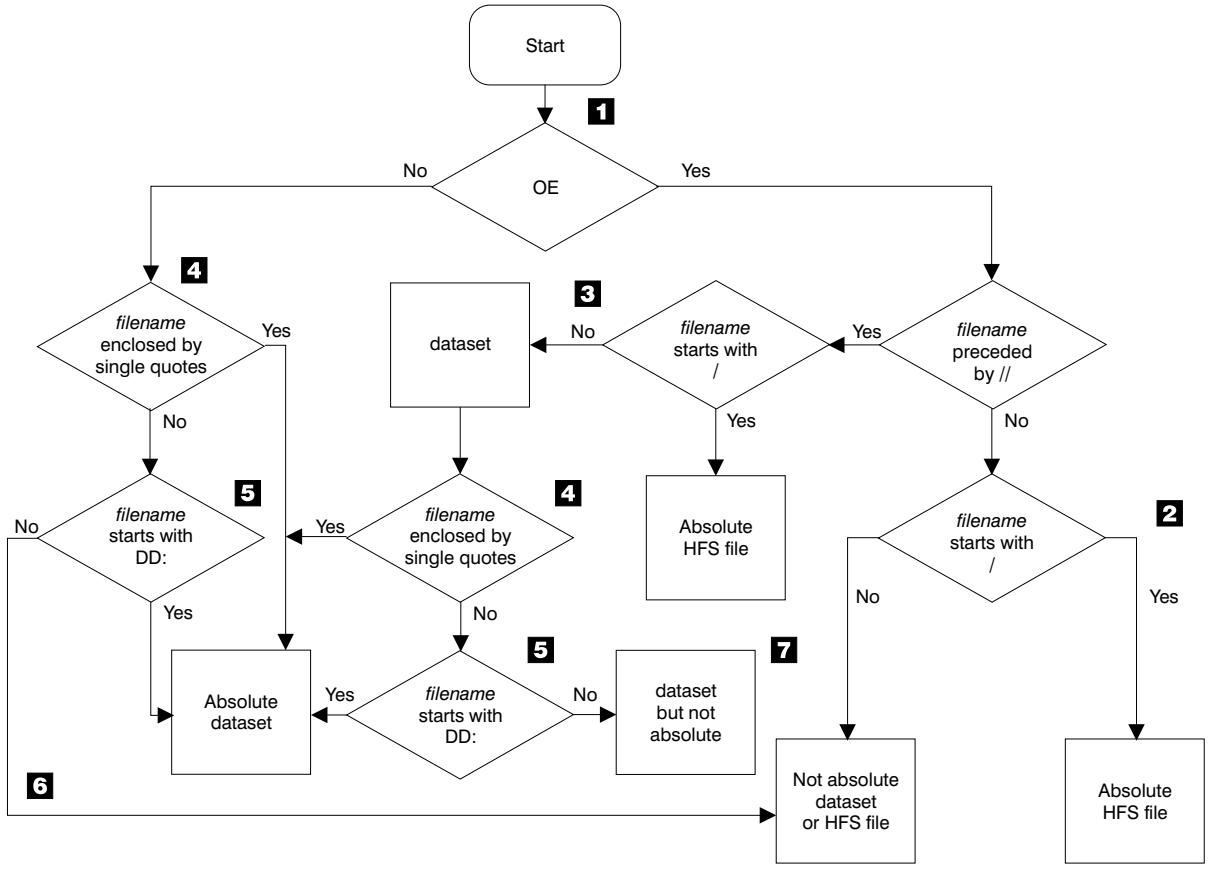


Figure 30. Testing If filename Is In Absolute Form

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 an HFS 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 an HFS 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 an HFS 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.

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 above 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 an HFS or a data set and then searched for the include file.

7 If none of the above 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, HFS/MVS (no starting slash)
  #include "/apath/afile.h"      absolute HFS, (starts with 1 slash)
  #include "//apath/afile.h.c"   NOT absolute, MVS (starts with 2 slashes)
  #include "a.b.c"               NOT absolute, HFS/MVS (no starting slash)
  #include "///apath/afile.h"    absolute HFS, (starts with 3 slashes)
  #include "DD:SYSLIB"           NOT absolute, HFS/MVS (no starting slash)
  #include "//DD:SYSLIB"         absolute, MVS (DD name)
  #include "a.b(c)"              NOT absolute, HFS/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 an HFS path or a data set path:

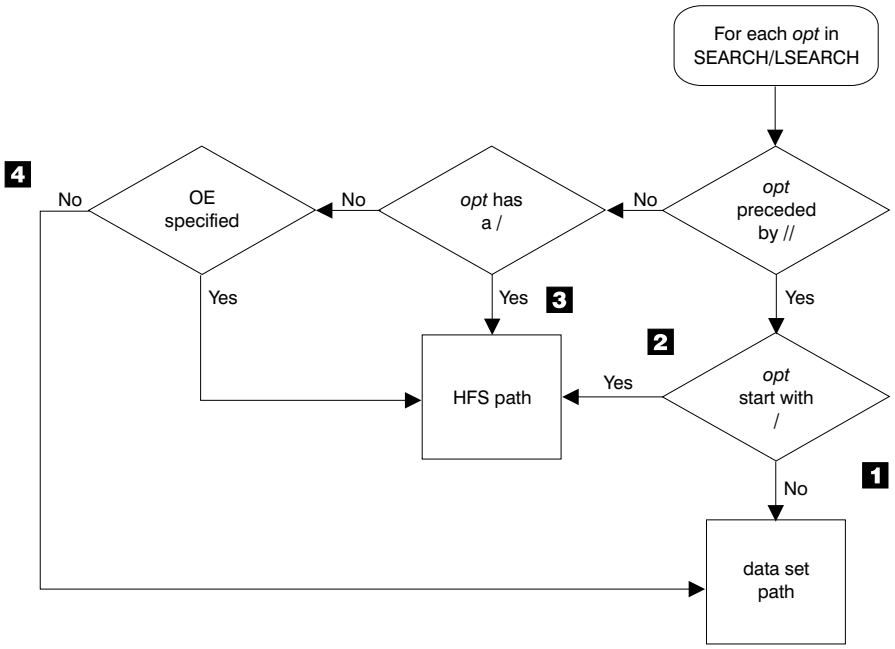


Figure 31. Determining if the SEARCH/LSEARCH opt is an HFS path

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 an HFS path.
3. If *opt* is **not** preceded by double slashes (//) and *opt* starts with a slash (/), then this path is an HFS 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 HFS path
OE SEARCH(PATH)	is treated as an HFS path
NOOE SEARCH(PATH)	is treated as a non-HFS path
NOOE SEARCH(//PATH)	is an explicit non-HFS path.

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. For example:

```
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-HFS so filename is treated as non-HFS)
2. /u/myincs/apath/afile.h   (/u/myincs is HFS so filename is treated as HFS)
3. userid.MAC1.H(AFILE)             (afile.h matches *.h)
```

An HFS 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 (/). For example:

```
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 HFS 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 155 and “SEARCH | NOSEARCH” on page 187 for more information on these compiler options.

Search Sequences for Include Files

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 331 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. The directory of the file that contains the #include directive
  2. When the containing file is HFS, the search order as specified on the `LSEARCH` option, if any
  3. The libraries specified on the `USERLIB DD` statement
  4. The search order for system include files

The example below 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 40. 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
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 331 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 file being processed is an HFS file and a main source file, the directory of the file containing the #include directive
  2. The search order as specified on the `LSEARCH` option, if any
  3. The libraries specified on the `USERLIB DD` statement
  4. The search order for system include files

For example, 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 41. Examples of Search Order for z/OS UNIX

`#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 system 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).	
″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 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 HFS in the 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 and cc z/OS UNIX System Services utilities.

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)

This is supplied for compatibility with previous releases, and will be overridden if SEARCH() is used as described above.

Compiling z/OS C++ Source Code Using the SEARCH option

The following data sets contain the commonly-used system header files for z/OS C++: [3]

- CEE.SCEEH (standard C++ header files)
- 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)
- CEE.SCEEH.T (standard template definitions)
- CBC.SCLBH.H (class library header files)
- CBC.SCLBH.HPP (class library header files)
- CBC.SCLBH.C (class library template definition files)
- CBC.SCLBH.INL (class library inline definition 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 HFS in the directories /usr/include and /usr/lpp/ioclib/include. To specify that the compiler search these directories, code the option:

```
SEARCH(/usr/include/,/usr/lpp/ioclib/include/)
```

This option is the default for the cxx z/OS UNIX System Services utility.

3. The high-level qualifier may be different at your installation.

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 9. Using the IPA Link Step with z/OS C/C++ Programs

This chapter shows how to use the IPA (Interprocedural Analysis) Link step with your z/OS C/C++ program. Before reading this chapter, refer to the *z/OS C/C++ Programming Guide* for an overview of IPA.

The `IPA(LINK)` option triggers IPA Link step processing. For further information on this option, see “IPA Link Step Suboptions” on page 137

IPA Linking Your Program

The IPA Link step combines IPA object files that are created by the IPA Compile step with non-IPA object files and information from load module library members. The IPA Link step optionally performs IPA and code generation optimizations, and generates the final code and data for your program. You must bind the resulting object module to create the executable program.

The entry point of your application must be an IPA object file.

Typically, z/OS C/C++ applications contain references to z/OS Language Environment library functions, as well as interface routines for products such as CICS and DB2. These object module and load module libraries must be available to the IPA Link step for symbol resolution. The IPA Link step extracts all required object information from these libraries to form part of the object module it generates. If external references remain unresolved after the link portion of the IPA Link step has completed, processing terminates before optimization or code generation of the final object code.

IPA Linking with XPLINK

OS/390 Version 2 Release 10 introduced support for XPLINK for applications built without using IPA. The XPLINK version of the LE runtime library uses dynamic linkage (DLLs). The application defines the runtime entry points by including appropriate side-deck members from the SCEELIB library:

- For C applications include CELHS001 and CELHS003
- For C++ applications also include CELHSCPP. If the application is using the ISO C++ Standard Library, also include C128.

A small portion of the LE runtime support (tables and control information) is statically bound into the application module. These are resolved from the SCEEBIND library. This is a program object library which IPA Link cannot read.

z/OS Version 1 Release 2 introduces IPA support for applications built using IPA. The SCEELIB members remain as before, but a new SCEEBND2 library is introduced. This contains the static parts in object format, which must be used if you are using IPA Link.

You should specify the libraries that are described in the previous paragraph in your bind step. During IPA Link step processing with `IPA(NONCAL)` in effect, IPA resolves object information for explicit run-time symbols. The IPA Link step produces additional, implicit references to external run-time symbols during code generation. Although the IPA Link step will search for explicit run-time references, it does not search for implicit run-time references.

To avoid problems with unresolved implicit run-time references, ensure that the run-time object module libraries are available to the binder. Also, check the binder listings and messages to make sure that all your symbols are resolved.

IPA object modules contain longnames, and may be included in object libraries for easy automatic library call resolution.

For information on creating object libraries in z/OS C/C++, refer to “Chapter 13. Object Library Utility” on page 425. For information on binding object modules under z/OS UNIX System Services, refer to “Chapter 10. Binding z/OS C/C++ Programs” on page 365.

IPA Linking with NOXPLINK

OS/390 Version 2 Release 4 has introduced the SCEELKEX library, which is a LONGNAME object version of a large portion of the Language Environment function library. When you IPA Link your application program, place the SCEELKEX library ahead of the SCEELKED library in the search order. This will preserve long run-time function names in the object module and listings that IPA Link generates.

You should specify the libraries that are described in the previous paragraph in your bind step. During IPA Link step processing with `IPA(NONCAL)` in effect, IPA resolves object information for explicit run-time symbols. The IPA Link step produces additional, implicit references to external run-time symbols during code generation. Although the IPA Link step will search for explicit run-time references, it does not search for implicit run-time references.

To avoid problems with unresolved implicit run-time references, ensure that the run-time object module and load module libraries are available to the binder. Also, check the binder listings and messages to make sure that all your symbols are resolved.

If you use the prelinker, make sure that the run-time object module libraries are available to the prelinker, and that the run-time object module and load module libraries are available to the Linkage Editor. The Object Resolution Warnings section of the Prelinker Map and the Linkage Editor Map display unresolved references, as follows:

```
=========================================================
|            Object Resolution Warnings                 |
=========================================================

WARNING EDC4015: Unresolved references are detected:
CEEBETBL CEEROOTA EDCINPL
```

IPA object modules contain longnames, and may be included in object libraries for easy automatic library call resolution.

For information on creating object libraries in z/OS C/C++, refer to “Chapter 13. Object Library Utility” on page 425. For information on binding object modules under z/OS UNIX System Services, refer to “Chapter 10. Binding z/OS C/C++ Programs” on page 365.

Using DD Statements for the Standard Data Sets

The IPA Link step uses certain ddnames. Table 42 lists these ddnames, along with their types and functions. For details on the attributes of specific data sets see “Description of Data Sets Used” on page 555.

Table 42. Data Sets Used by the IPA Link Step

ddname	Type	Function
SYSIN[1]	Input	Primary input
STEPLIB[1],[5]	Utility Library	Location of the z/OS C/C++ compiler (which provides the IPA Link step) and the z/OS Language Environment data sets
SYSLIB	Library	Data set for run-time library: • For XPLINK: SCEEBND2[2] • For NOXPLINK: SCEELKEX[4] and SCEELKED[2] Optional data sets for secondary input [4]
SYSLIN[1]	Output	Output data set for the object module, if the `OBJECT` compiler option is specified
SYSOUT[4]	Output	Destination of diagnostic messages generated by the IPA Link step
SYSCPRT[4]	Output	IPA Link step listing, generated if the `IPA(MAP)`, `LIST`, or `XREF` option is specified.
User-specified[3]	Input	Additional object modules and load modules
SYSUT1, SYSUT4-9, SYSUT14[1]	Output	Work data sets

Notes:

1. Required data set
2. Required for library run-time routines
3. As required by the program:
   - Program parts in object library or load module library format
   - DLL IMPORT side-decks generated by the binder, which define function or variable interfaces of a DLL referenced by the current application
4. Optional data set
5. 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 IPA Link step must be one or more separately compiled object modules or IPA Link control statements. You can specify this input in a sequential data set, a member of a partitioned data set, or an in-line object module (DD *).

Location of Compiler and z/OS Language Environment Library (STEPLIB)

To IPA Link your program, the system must find the data sets that contain the compiler, and the data sets that contain the z/OS Language Environment run-time library. If the run-time library is installed in the LPA or ELPA, it is found

automatically. Otherwise, SCEERUN must be in the JOBLIB or STEPLIB. For information on the search order, see “Chapter 12. Running a C or C++ Application” on page 413.

Secondary Input (SYSLIB)

Secondary input to the IPA Link step consists of object modules, or load modules that are not part of the primary input data set but are to be included in the user executable program. These may be included either:

- Explicitly, as a result of processing an IPA Link control INCLUDE statement.
- Implicitly, as a result of automatic call library processing. This can be due to either
  - Processing a library specified on an IPA Link control LIBRARY statement
  - Searching the libraries that are allocated to SYSLIB (once the IPA Link step has processed all primary input)

The automatic call library is used to resolve external symbols that are currently unresolved.

The call libraries that are used as input to the IPA Link step normally include the z/OS Language Environment libraries. If required, include additional call libraries such as SYS1.LINKLIB, a private program library, or a subroutine library to resolve all external references to your application.

If you are IPA Linking an application that imports symbols from a DLL, you must INCLUDE its definition side-deck on the SYSLIB or other user DD name. The IPA Link step uses the definition side-deck to resolve external symbols for functions and variables that your application imports. If you call more than one DLL, you need to INCLUDE a definition side-deck for each.

You can use the SYSLIB DD statement to concatenate multiple object module libraries and load module libraries. For more information on concatenating data sets, see page 310.

Notes:

1. All secondary input data sets for the IPA Link step must be cataloged.
2. The IPA Link step supports PDS format load module libraries only. It does not support Program Objects that are in PDSE format, or z/OS UNIX System Services HFS executable files.

Output (SYSLIN)

The IPA Link step generates a single object module in the data set that is referenced by the SYSLIN DD name.

Destination of Errors Generated by the IPA Link Step (SYSOUT)

If the IPA Link step encounters problems, it generates diagnostic messages and places them in the SYSOUT data set.

Listing (SYSCPRT)

If you specify the ATTRIBUTE, IPA(MAP), LIST, or XREF compiler option, the IPA Link step writes a listing to the SYSCPRT file name. The options have the following purposes:

ATTRIBUTE Causes IPA Link to generate an External Symbol Cross-Reference

listing section for each partition. The IPA Link step may also generate a Storage Offset Listing if you specified the `XREF`, `IPA(ATTRIBUTE)`, or `IPA(XREF)` option specified during the IPA Compile step.

`IPA(MAP)` Provides information about the object and source files that are included as input to the IPA Link step, and information about the partitions that it generates.

`LIST` Causes IPA Link to generate a Pseudo Assembly listing for each partition, showing the code and data that are generated in each partition.

`XREF` Causes an IPA Link to generate an External Symbol Cross-Reference listing section to each partition. The IPA Link step may also generate a Storage Offset Listing if you specify the `XREF`, `IPA(ATTRIBUTE)`, or `IPA(XREF)` option specified during the IPA Compile step.

Refer to "Using the IPA Link Step Listing" on page 274 for more information about listings that the IPA Link step generates.

Temporary Workspaces for the IPA Link Step (SYSUTx)

The IPA Link step requires data sets for use as temporary workspaces. You define these data sets by DD statements with the names SYSUT1, SYSUT4—9, and SYSUT14. These data sets must be on direct access devices.

IPA Link Step Input

Input to the IPA Link step can be:

- Object records, which can be:
  - One or more IPA object modules
  - IPA Link control statements
  - z/OS Language Environment stub routines
  - Other object libraries and load module libraries
- The IPA Link step control file

Unresolved references or undefined writable static objects often result if you give the IPA Link step object modules produced with a mixture of inconsistent options. For example, `RENT`, `NORENT`, or `DLL`.

Objects processed with the `IPA(OBJONLY)` option are processed the same as objects with the `NOIPA` option.

Note: The IPA Link step will not accept as input a program object that is produced by the binder.

Primary Input

Primary input to the IPA Link step consists of a sequential data set (file) that contains one or more separately compiled object modules or IPA Link control statements. Specify the primary input data set through the `SYSIN` DD name.

Note: If you used the OS/390 Release 2 C/C++ compiler to create an IPA or combined IPA/conventional object module, and specified the `OPTIMIZE(0)` and `IPA(NOOPTIMIZE)` compiler options, your object module is incompatible

with a later release of the z/OS C/C++ IPA Link step. You must recompile your source code with a later release of the C/C++ IPA Compile step before attempting to use the current release of the z/OS C/C++ IPA Link step.

Refer to “Object Record Formats” on page 347 for more information about the different types of object records.

IPA Linking Multiple Object Modules

z/OS C/C++ generates a CEESTART CSECT at the beginning of the object module in two situations:

- For a source program that contains the `main()` function, as long as you have not specified the `NOSTART` compiler option.
- For a source program containing a function for which a `#pragma linkage (`*name*`, FETCHABLE)` preprocessor directive applies.

When you IPA Link multiple object modules into a single object module, the binder resolves the entry point of the resulting object module to the external symbol CEESTART. If you want to control the entry point of the object module, use the `ENTRY` binder control statement or the c89 ″-e″ option.

For the IPA Link step, object modules containing the `main()` function or `#pragma fetchable` function must be IPA object files. If these object files are IPA Linked with other object modules produced by C, assembler, or other languages, the IPA object file containing the `main()` or `#pragma fetchable` function 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, you can include the following binder `ENTRY` control statement in the input to the binder:

```
ENTRY CEESTART
```

If you are building a DLL with IPA, you must use the `ENTRY` control statement as described above.

Secondary Input

Secondary input to the IPA Link step consists of object modules, or load modules that are not part of the primary input data set but are to be included in the object module. They may be included either:

- Explicitly, as a result of processing an IPA Link control INCLUDE statement.
- Implicitly, as a result of automatic call library processing. This can be due to either
  - Processing a library specified on an IPA Link control LIBRARY statement
  - Searching the libraries that are allocated to SYSLIB (once the IPA Link step has processed all primary input)

The automatic call library is used to resolve external symbols that are currently unresolved. The IPA Link step locates the library member in which the external symbols are defined, extracts the corresponding object information, and incorporates it in the output object module.

The automatic call library may include:

- Object module libraries. These may contain IPA object files or non-IPA object modules, and may contain the records of IPA Link control statements.

  These libraries may be:
  - PDS libraries

  - PDSE libraries
  - archive libraries

  Note: You do not normally use control statement records within secondary input with the `c89` utility. The `c89` utility allocates libraries that are passed in the `c89` invocation. You cannot allocate additional user autocall libraries with user-specified DD names.
- Load module libraries
- z/OS Language Environment libraries, if any of the z/OS Language Environment library functions are needed to resolve external references

Refer to “Object Record Formats” on page 347 for more information about the different types of object records.

Note: You can concatenate PDS, PDSE, and load module libraries together. However, you cannot concatenate archive libraries to other library types.

Specify the standard secondary input data sets with a SYSLIB DD statement. You can also explicitly reference secondary input, through IPA Link control statements.

Additional Object Modules and Load Modules as Input

You can explicitly reference secondary input through `INCLUDE` or `LIBRARY` control statements.

Use the `INCLUDE` statement to specify additional object information from object modules or load modules that you want included in the final object module.

Use the `LIBRARY` statement to specify additional libraries to be searched for object information from object modules or load modules to be included in the final object module. The IPA Link step only uses data sets that are specified by the `LIBRARY` statement if there are unresolved references once it has processed all other input.

When the IPA Link step encounters an `INCLUDE` statement, it incorporates the data sets that the statement specifies. If you specify the `IPA(NONCAL)` option, the IPA Link step performs a library search for currently unresolved symbols when it encounters a `LIBRARY` statement. If the processing of subsequent `INCLUDE` or `LIBRARY` statements results in new or unresolved symbols, the IPA Link step does not search a previously encountered library again. You need to specify another `LIBRARY` statement that points to the same library so that IPA Link searches it again.

Uppercase Name Resolution with the IPA(UPCASE) Option

If you specify the `IPA(UPCASE)` option, the IPA Link step makes an additional automatic library call pass against the SYSLIB DD statement. In this situation, symbol matching is case-insensitive. The purpose of this `IPA(UPCASE)` option is to provide support for linking assembler object routines without source changes. It is preferable to add `#pragma map` definitions for these symbols, so that IPA Link finds the correct symbols during normal automatic library call processing.

Processing the IPA Link Automatic Library Call

The IPA Link step uses the following process to resolve a referenced and currently undefined symbol, if you have specified the `IPA(NONCAL)` compiler option:

- If the data set contains a `C370LIB` directory created using the z/OS C/C++ Object Library Utility, and the `C370LIB` directory shows that a defined symbol by that name exists (with a case-insensitive exact match), the IPA Link step reads the PDS member containing that symbol.

- If the data set does not contain a C370LIB directory created using the z/OS C/C++ Object Library Utility and the reference is not to static external data, the IPA Link step reads the member or alias with the same name (with a case-sensitive exact match).

The z/OS V1R2 release introduces support for object libraries, which contain members that define different linkage conventions for the same symbols. This is accomplished by creating separate members for each convention and processing with the appropriate library utility (C370LIB for PDS and PDSE libraries, ar for HFS archive libraries). The IPA Link step and the binder will select the member with the convention that matches the symbol reference.

If unresolved symbols remain after IPA Link step has processed user input, and you specified the NONCAL option, the IPA Link step searches the files allocated to the SYSLIB DD name, as follows:

1. It searches for a case-insensitive exact match in the C370LIB and non-C370LIB libraries that are concatenated to the SYSLIB DD name, as described above.
2. If the symbol remains unresolved, IPA searches z/OS Language Environment for a library function with the same name as the symbol. (You must include the Language Environment stub library in the SYSLIB concatenation).
3. If the symbol is still unresolved, and you have specified the IPA(UPCASE) option, IPA searches using the uppercased name.

For more information about the z/OS C/C++ Object Library Utility, see “Chapter 13. Object Library Utility” on page 425.

References to Currently Undefined Symbols (External References)

If the IPA Link step finds unresolved references to external symbols after it has completed the link portion of its processing, it issues a diagnostic message and terminates processing.

Symbol Attribute Match Checking

As IPA Link processes object files, checks are made to ensure that the attributes of symbol references and symbol definitions are compatible. This occurs initially for external symbols as the object files are read. As the program information in IPA object files is linked together and the call graph is constructed, further checks are made.

If the IPA Link step finds incompatible attributes, it issues a diagnostic message and terminates processing.

Library Routine Considerations

z/OS Language Environment contains run-time libraries for all Language Environment-enabled languages: C, C++, COBOL, FORTRAN, and PL/I. For detailed instructions on linking and running z/OS C/C++ programs under z/OS Language Environment, refer to the *z/OS Language Environment Programming Guide*.

z/OS Language Environment is dynamic. That means that many of the functions, such as library functions, are not physically stored as a part of your executable program.

- For XPLINK, the run-time environment uses dynamic linkage (DLLs). The side-decks are in the file CEE.SCEELIB. The DLLs are in the file CEE.SCEERUN2.

If the application is using the ISO C++ Standard Library, additional side-decks and DLLs from the same libraries are used.

- For NOXPLINK static linkage, only a small portion of code, known as a stub routine, is stored with your executable program. This results in a smaller executable module. There is a stub routine for each library function. Each stub routine has:
  - The same name as the library function that it represents
  - Enough code to locate the actual library function at run time

  The C stub routines are in the file `CEE.SCEELKED`, or `CEE.SCEELKEX`.

Using DLLs

If you are building an application that imports symbols from a DLL, your input to the IPA Link step must include the definition side-deck that the binder produced when the DLL was built.

The IPA Link step uses longnames to resolve exported and imported symbols when it generates an object module for an application that is compiled with the `DLL` compiler option.

When IPA Link is used for XPLINK applications, the run time will always use dynamic linkages (DLLs).

For information on how to create a DLL or an application that uses DLLs, see the *z/OS C/C++ Programming Guide*.

Object File Formats

The High Level Assembler (HLASM) and other z/OS compilers and language translators generate two object file formats:

Object File Format
: The standard S/370 "TEXT" object format, packaged as fixed-length 80 byte records. Extensions to the basic format support long external symbols when the z/OS C/C++ compiler `LONGNAME` option is in effect. IPA Link accepts input in object file format. The z/OS C/C++ compiler only produces files that are in object file format.

Generalized Object File Format (GOFF)
: A hierarchical object file format that was introduced with HLASM R2, and the z/OS Binder. IPA Link accepts this format as input, whether the object file was compiled XPLINK or NOXPLINK, IPA or NOIPA.

Refer to *z/OS DFSMS Program Management* for more information on object file formats.

Object Record Formats

There are two basic types of object records which may be present in a file of object file format.

Binary Object Records

Binary object records provide information about your program. The records may include IPA object information, or code and data generated through the `OBJECT` suboption of the `IPA` compiler option during the IPA Compile step.

The records include the following types:

- ESD

- XSD
- TXT
- END
- RLD
- GOFF object records

The z/OS C/C++ compiler or an equivalent language translator may generate these object records.

IPA Link Control Statements

You can also specify control statement records as input. These statements can include the following types:

- `INCLUDE`
- `LIBRARY`
- `RENAME`
- `IMPORT`
- `ALIAS`
- `ENTRY`
- `NAME`

The `INCLUDE` and `LIBRARY` control statements explicitly identify secondary input files.

IPA Link control statements are contiguous records that you can specify in an object file or in a DD * stream. The syntax and format of these control statements are similar to those that the binder uses. The logical records can span multiple fixed-block, 80 column wide physical records.

You can specify blank records and comment control statements (those starting with an asterisk in column 1), but the IPA Link step ignores them.

The following table shows the format of records:

Table 43. IPA Link Control Statements

Start Column	End Column	Field Length	Description
1	1	1	Record type indicator • blank-Control • 0X02-Binary • ″*″-Comment
2	71	70	Record data
72	72	1	Control statement continuation indicator • EBCDIC blank character-No continuation (required for last record) • non-blank EBCDIC character-Continuation
73	80	8	Record sequence number (optional field, contents not verified)

You can delimit character strings with blanks, commas, or parentheses. If character strings contain embedded blanks, you must enclose the strings in single quotes. If

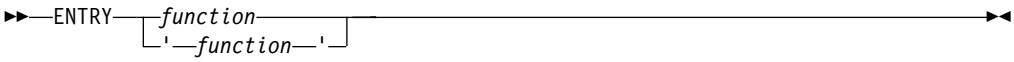
you want to enclose a name in single quotation marks, and it contains a single quotation mark, replace the single quotation mark with two adjacent ones. For example, if you want the name `SymbolNameWithAQuote'InTheMiddle`, specify it as follows: `'SymbolNameWithAQuote''InTheMiddle'`.

All 70 data characters of a control statement are significant. Control statements continue in column 2 (IPA conforms to the same convention as the Program Management Binder).

The IPA Link step performs syntax checking on the object records. If it finds an error, it issues a diagnostic message and indicates the location of the error. Records cannot continue past the end of an object file.

The following sections describe the IPA Link control statements.

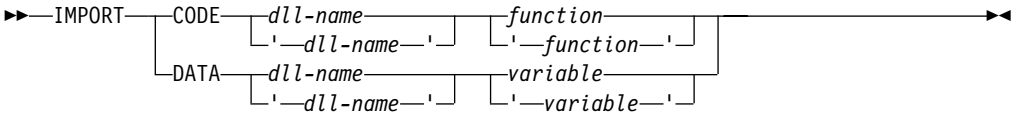
ENTRY Control Statement: The `ENTRY` control statement has the following syntax:



function An external symbol that will be used as the program entry point. Mixed-case longnames must be enclosed in quotes.

The IPA Link step processes `ENTRY` statements. It passes the statement to the binder.

IMPORT Control Statement: The `IMPORT` control statement has the following syntax:



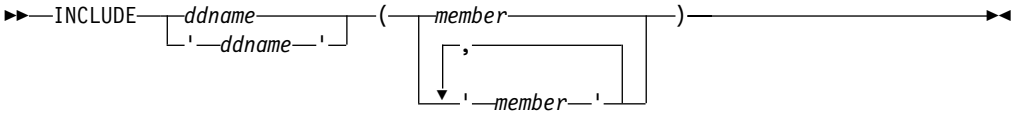
dll-name The directory name (primary member or alias) or HFS filename of the load module or program object that contains the imported function or variable. The maximum length of a dll-name is 1024 characters. The maximum length of an HFS filename is 255 bytes.

variable An exported variable name. It is a mixed-case longname.

function An exported function name. It is a mixed-case longname.

The IPA Link step processes `IMPORT` statements. It passes the binder the statements that represent entry points that are present within the DLL.

INCLUDE Control Statement: The `INCLUDE` control statement has the following syntax:



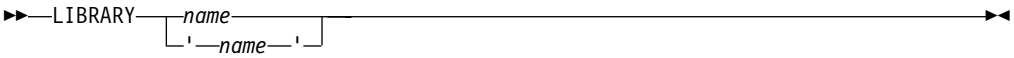
ddname a ddname associated with a file to be included.

member the member of the DD to be included.

The IPA Link step attempts to read the DD or member of the DD (whichever you specify), and if successful, resolves the INCLUDE request.

Note: The IPA Link step removes the INCLUDE control statement and does not place it in the IPA Link output object module.

LIBRARY Control Statement: The LIBRARY control statement has the following syntax:



name The name of a DD that defines a library. This could be a concatenation of one or more libraries that were created with or without the Object Library Utility.

Note: The IPA Link step removes the LIBRARY control statement and does not place it in the IPA Link output object module.

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. Note that in the listed directives, *name* can be a regular expression. Thus, *name* can match multiple symbols in your application through pattern matching. For more information on regular expressions, see “Using Regular Expressions” on page 354.

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.

export=*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 indirectly 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, above.

isolated Specifies that the missing functions are "isolated". See the description for the `isolated` directive, above.

pure Specifies that the missing functions are "pure". See the description for the `pure` directive, above.

unknown Specifies that the missing functions are "unknown". See the description for the `unknown` directive, above. 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.

LIBANSI Option and Symbol Attributes

The IPA Link step has attribute information for the Language Environment run-time entry points. When the `LIBANSI` option is active and Level (2) IPA optimization is selected during the IPA Link step, this information will be used to improve the generated code.

Using Regular Expressions

You can specify various attributes about functions such as controlling whether a function is inline or not through the IPA control file. For example, the following directives directs IPA not to inline functions `foo1()`, `foo2()`, or `foo3()`. It directs IPA to inline functions `bar1()`, `bar2()`, and `bar3()`. It also declares that `trace1()`, `trace2()`, and `trace3()` are low frequency functions:

```
noinline=foo1,foo2,foo3
inline=bar1,bar2,bar3
lowfreq=trace1,trace2,trace3
```

Regular expressions are a common form of expressing pattern matching. The most common forms of regular expressions are listed below. Using regular expressions, you can describe the list of functions to which these attributes and inline directives refer. For example, you could rewrite the above directives as:

```
noinline=foo[123]
inline=bar[123]
lowfreq=trace[123]
```

The directives for which regular expressions are supported in the IPA control file are:

- inline
- noinline
- unknown
- lowfreq
- exits
- pure
- safe

You can also find additional information on using regular expressions in the z/OS UNIX shell in the *z/OS UNIX System Services Command Reference*.

`string` A regular string of characters matches the same string of characters in the item you are searching. Thus you can search for all occurrences of the string `test` by using the regular expression `test`. This will also match lines containing the strings `testimony`, `latest` and `intestine`.

`start (^)` Indicates a *beginning of line* in a match. For example `^test` matches all lines that begin with `test`. Note that this must appear as the left most character in the expression.

`end ($)` Indicates an *end of line* in a match. The regular expression `test$` will match those lines that end with `test`, and `^test$` will match those lines that contain only `test`. Note that to work as the end of line, the `$` must be the last character in the expression.

`single character (.)`
The period matches any character. Matches for `t.st` includes `test`, `tast`, `tZst`, and `t1st`.

`escape (\)` Use the back slash character to *escape* special characters if you want to search for them in the expression. For example, if you wanted to find those lines ending with a period, the expression `.$` shows all lines that have at least one character in them. You need to escape the `.` as `\.$`. Similarly to find those places with the two

characters period and dollar sign next to each other, use \.\. You also have to escape the back slash character to match it by using \\.

`character set ([ ])`
Represents a set of characters to compare against a single character. Ranges of characters can be represented by the first character in the series, followed by a hyphen, then the last character in the series. For example, [abcdefg] is the same as [a–g]. The pattern `t[a–g123]st` matches `tast`, `test`, and `t2st`, but not `t–st`, `taast` nor `tAst`. Note that the special characters \,], –, and ^, must be escaped. See the escape character, (\), above. The special character] can appear as the first character in the range without being escaped.

`complemented character set ([^ ])`
The special character ^ placed at the beginning of a character set indicates that the character set must not match the single character. For example, `t[^a-zA-Z]st` matches `t1st`, `t–st`, or `t,st`, but not `test` or `tYst`.

`grouping (( ))`
You can use parentheses to group expressions. This is useful when you want to use a string as basis for a pattern. For example, `(test)+` matches multiple instances of `test`.

`alternation (|)`
This is an *or* operator. It allows the extension of a pattern to include alternative matches. For example, to match those lines beginning with `Page` or ending with `Foot`, you could use the expression `^Page|Foot$`.

`replication (*)`
When a character pattern is followed by a `*`, the pattern will match zero or more instances of that pattern. For example, the pattern `te*st` matches `tst`, `test`, `teeeeeest`, and so on.

`replication (+)`
This modifier works the same way as `*`, but it matches at least one instance of the pattern. For example, `t(es)+t` will match `test`, `tesest`, and so on, but not `tt`.

`replication (?)`
The question mark is used in a fashion similar to the two previous patterns, but matches exactly one or zero matches. Thus, `te?st` will matter either `tst` or `test`.

`replication ({m,n})`
For more rigid replicated pattern matches, the braces can be used to indicate an exact number, a minimum number, or a range of numbers of replications. To match exactly `m` copies, use the form {*m*}, where *m* is a number between 1 and 255. To match a minimum of *m* copies, use the form {*m*}. To match between *m* and *n* copies, use the form {*m*,*n*}} where *n* must be between *m* and 255. For example, `a{2}` matches `aa`, `b{1,4}` matches `b`, `bb`, `bbb` and `bbbb`.

Output from the IPA Link Step

You can specify output from the IPA Link step as one of the following:
1. A sequential data set
2. A member of a partitioned data set
3. A partitioned data set
4. A hierarchical file system (HFS) file
5. An HFS directory

Output may be either an object module or a listing.

For valid combinations of input and output file types, refer to Table 38 on page 305.

Specifying Output Files

You can use compiler options to specify the output files for IPA Link, as follows:

Table 44. Compiler Options That Provide Output File Names

Output File Type	Compiler Option
Object Module	OBJECT(filename)
Listing File	LIST(filename)

If you specify compiler options that generate output files but do not specify the suboptions to identify the output files or allocate the ddnames, the IPA Link step generates the output file names based on the input file name. For data sets, the IPA Link step uses the userid under which the compiler is running as the high-level qualifier. It generates the low-level qualifier by appending a suffix, as shown in Table 45. z/OS creates HFS files in the current working directory.

The IPA Link step uses the following default suffixes:

Table 45. Default Suffixes for Output File Types

Output File Type.	MVS File	HFS File
Output from IPA Compile Step	OBJ	`o`
Listing File	LIST	`lst`
Output from IPA Link Step	IPA	`I` (for c89 batch) or `o` (otherwise)

Refer to the *z/OS UNIX System Services Command Reference* for more information about default suffixes.

Note: Output files default to the HFS directory if the input resides in the HFS, or to the MVS file if the input resides in a data set.

If you use the c89 utility to compile HFS source files and perform an IPA Link in one invocation, and do not specify output filenames in the compiler options, the compiler writes output files to the current working directory. It generates output file names by:
- Appending a suffix, if it does not exist
- Replacing the suffix, if it exists

as shown in Table 45. For example, the following command generates the IPA Compile step object file `./hello.o` and the IPA Link step object file `./hello.I` :

```
cc  /user/tullio/hello.c
```

The IPA Link step object file `./hello.I` is temporary, but you can use environment variables to make it permanent. Refer to the z/OS Shells and Utilities manual for more information.

Notes:

1. If you have specified the `OE` option, see “OE | NOOE” on page 169 for a description of the default naming convention.
2. If you supply the primary input file inline in your JCL, you must provide a file name for the output, or route it to the job log. 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.

Listing Output

To create a listing file that contains source, object or inline reports, use one of the following:

- the `MAP` suboption of the `IPA` option
- the `AGGREGATE` option
- the `LIST` option
- the `INLINE(,REPORT,,)` option
- the `XREF` option

The IPA Link Step listing contains several individual listing sections that are only generated if required. Unresolved requests generate error or warning messages in the listing.

The listing includes the results of the default or specified options of the `IPARM` parameter (that is, the diagnostic messages and the object code listing). If you specified *filename* with two or more of these compile options, the IPA Link step combines the listings and writes them to the last file named. If you did not specify any suboptions, the IPA Link step writes the listing to the `SYSCPRT` DD name, if you have allocated it. Otherwise, the IPA Link step generates a default file name as described in “LIST | NOLIST” on page 150.

Object Module Output

To create an object module and store it on disk or tape, you can use either the `OBJECT` compiler option.

If you do not specify a *filename* with the `OBJECT` compiler option, the IPA Link step stores the object code in the file that you defined in the `SYSLIN DD` statement. If you did not specify any suboptions, and did not allocate SYSLIN, the IPA Link step generates a default file name as described in “OBJECT | NOOBJECT” on page 166.

You must use the binder (or the prelinker, followed by the linkage-editor) to process the object module from the IPA Link step. You should not use the output object module from one IPA Link step as input to another IPA Link step.

Mapping Static Symbol Names

Static symbols (such as C static functions) within a compilation unit are not exposed as external symbols if an application program is built using the non-IPA compilation process.

The IPA Link merges and optimizes the IPA object information, and splits it into partitions for final code and data generation. The partitioning process must flexibly assign the code and data from the original Compilation Units to their final partition based on how they are used within the application.

As the IPA Link step reads IPA object modules, it assigns each static static symbol a unique name and promotes it to an external symbol. This prevents static symbols from constraining the partitioning. IPA Link generates the unique name by adding a prefix to the original static name, as follows:

@n@original_name
where *n* is the object file id number. Refer to the Object File Map section of the "Using the IPA Link Step Listing" on page 274 for details.

If an object file defines multiple static symbols with the same name, IPA Link generates the unique name for the subsequent symbols as follows:

@n@m@original_name
where:

n is the object file id number.

m is the collision counter, starting with 1.

Running the IPA Link Step Under z/OS Batch

The following diagram shows the basic IPA Link step process for your C/C++ application.

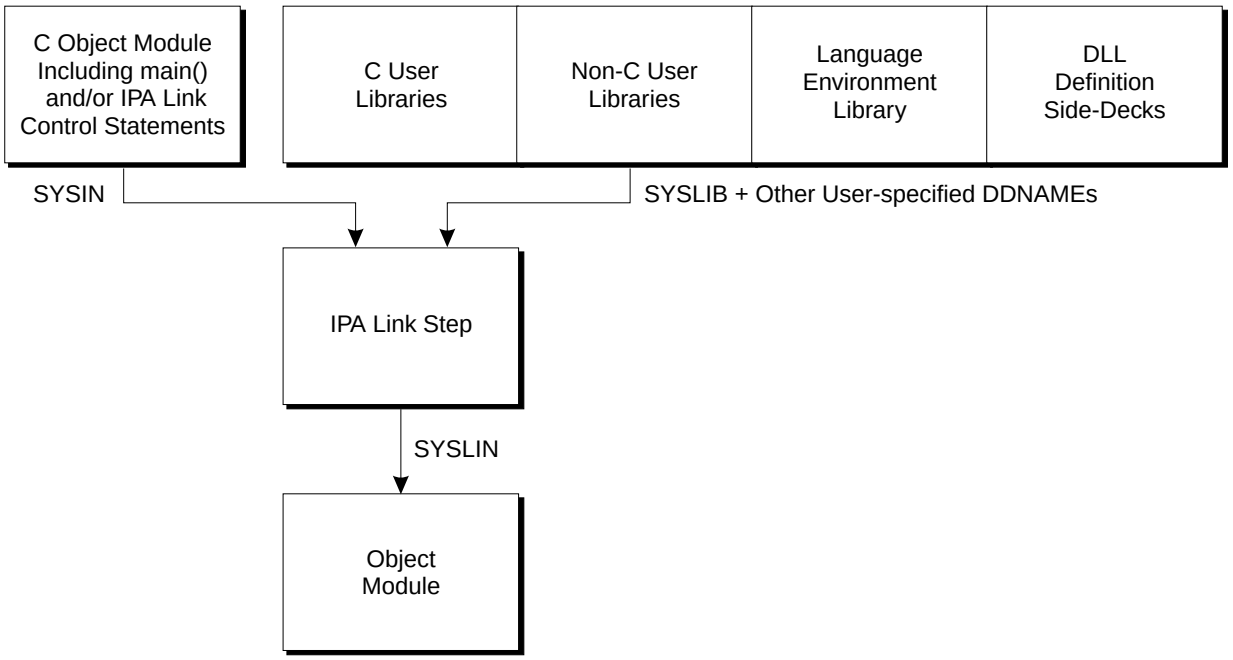


Figure 32. Basic IPA Link Step Processing

Use the SYSIN DD statement to specify your primary input. This may be object modules or IPA Link step control statements.

Use the SYSLIB DD statement to specify your secondary input. Your secondary input may be C/C++ user libraries, non-C/C++ user libraries, or the Language Environment library. Also, if you are creating an application that imports symbols from DLLs, you must INCLUDE the definition side-deck for each DLL from the SYSLIB DD statement.

You can specify additional secondary input through user-specified ddnames.

The IPA Link step stores the final object module that it generates in the data set that is referenced by the SYSLIN DD name.

Using the EDCI, CBCI, EDCXI and CBCXI Cataloged Procedures

You can use the IBM-supplied cataloged procedures EDCI and CBCI to perform IPA Link step processing on your non-XPLINK program. The two procedures are the same. IBM provides CBCI to conform to the procedure naming conventions of C++, and CBCI is aliased to EDCI. Note that by default, the EDCI procedure does not save the generated object module.

The EDCI procedure specifies the `IPA(LINK)` option for you. You can use the IBM-supplied cataloged procedures EDCXI and CBCXI to perform IPA Link step processing on your XPLINK program. The two procedures are the same.

The following example shows the general job control procedure for IPA linking a program under z/OS batch:

```
// jobcard
//*
//* IN THE FOLLOWING STEP, THE MEMBERS TESTFILE AND DECODE FROM
//* THE LIBRARIES USERID.WORK.OBJECT AND USERID.LIBRARY.OBJECT ARE
//* IPA LINKED, AND THE GENERATED OBJECT MODULE IS PLACED
//* IN USERID.WORK.IPAOBJ(TEST).
//* AN IPA LINK LISTING IS GENERATED AND DIRECTED TO SYSOUT=*.
//*
//IPALINK EXEC EDCXI,
//     INFILE='SEE.SYSIN.OVERRIDE',
//     OUTFILE='USERID.WORK.IPAOBJ(TEST),DISP=SHR',
//     IPARM='IPA(MAP,LIST,DUP,ER,NONCAL)',
//     IREGSIZ=64M
//OBJECT   DD  DSNAME=USERID.WORK.OBJECT,DISP=SHR
//LIBRARY  DD  DSNAME=USERID.LIBRARY.OBJECT,DISP=SHR
//DECKLIB  DD  DSNAME=CE.SCEELIB, DISP=SHR
//SYSOUT   DD  SYSOUT=*
//SYSCPRT  DD  SYSOUT=*
//SYSIN DD DATA,DLM=@@
 INCLUDE OBJECT(TESTFILE)
 INCLUDE LIBRARY(DECODE)
 INCLUDE DECKLIB(CELHS003)
 INCLUDE DECKLIB(CELHS001)
@@
```

Figure 33. IPA Linking an XPLINK C Program under z/OS Batch

```
// jobcard
//*
//* IN THE FOLLOWING STEP, THE MEMBERS TESTFILE AND DECODE FROM
//* THE LIBRARIES USERID.WORK.OBJECT AND USERID.LIBRARY.OBJECT ARE
//* IPA LINKED, AND THE GENERATED OBJECT MODULE IS PLACED
//* IN USERID.WORK.IPAOBJ(TEST).
//* AN IPA LINK LISTING IS GENERATED AND DIRECTED TO SYSOUT=*.
//*
//IPALINK EXEC EDCI,
//     INFILE='SEE.SYSIN.OVERRIDE',
//     OUTFILE='USERID.WORK.IPAOBJ(TEST),DISP=SHR',
//     IPARM='IPA(MAP,LIST,DUP,ER,NONCAL)',
//     IREGSIZ=64M
//OBJECT   DD  DSNAME=USERID.WORK.OBJECT,DISP=SHR
//LIBRARY  DD  DSNAME=USERID.LIBRARY.OBJECT,DISP=SHR
//DECKLIB  DD  DSNAME=CEE.SCEELIB,DISP=SHR
//SYSOUT   DD  SYSOUT=*
//SYSCPRT  DD  SYSOUT=*
//SYSIN DD DATA,DLM=@@
 INCLUDE OBJECT(TESTFILE)
 INCLUDE LIBRARY(DECODE)
 INCLUDE DECKLIB(CELHS003)
 INCLUDE DECKLIB(CELHS001)
 INCLUDE DECKLIB(CELSCPP)
 INCLUDE DECKLIB(C128)
@@
```

Figure 34. IPA Linking an XPLINK C++ Program under z/OS Batch

```
// jobcard
//*
//* IN THE FOLLOWING STEP, THE MEMBERS TESTFILE AND DECODE FROM
//* THE LIBRARIES USERID.WORK.OBJECT AND USERID.LIBRARY.OBJECT ARE
//* IPA LINKED, AND THE GENERATED OBJECT MODULE IS PLACED
//* IN USERID.WORK.IPAOBJ(TEST).
//* AN IPA LINK LISTING IS GENERATED AND DIRECTED TO SYSOUT=*.
//*
//IPALINK EXEC EDCI,
//     INFILE='SEE.SYSIN.OVERRIDE',
//     OUTFILE='USERID.WORK.IPAOBJ(TEST),DISP=SHR',
//     IPARM='IPA(MAP,LIST,DUP,ER,NONCAL)',
//     IREGSIZ=64M
//OBJECT   DD  DSNAME=USERID.WORK.OBJECT,DISP=SHR
//LIBRARY  DD  DSNAME=USERID.LIBRARY.OBJECT,DISP=SHR
//SYSOUT   DD  SYSOUT=*
//SYSCPRT  DD  SYSOUT=*
//SYSIN DD DATA,DLM=@@
 INCLUDE OBJECT(TESTFILE)
 INCLUDE LIBRARY(DECODE)
@@
```

Figure 35. IPA Linking a Non-XPLINK Program under z/OS Batch

Specifying IPA Link Options

Use the IPARM parameter to specify the IPA Link options. The format of the parameter is:

```
IPARM='"ipa-link-options" '
```

where *ipa-link-options* is a list of IPA Link options, separated by commas.

Specifying Region Size

Use the IREGSIZ parameter to specify the IPA Link step region. The format of the parameter is:

```
IREGSIZ=region-size
```

Specifying Secondary Input under z/OS Batch

Specify the secondary input data sets with the SYSLIB DD statement.

- XPLINK with SCEEBND2

  Run-time symbols will be resolved from longname IMPORT statements in previously INCLUDEd side-decks from SCEELIB where available, otherwise they will be resolved from members of SCEEBND2.

  Add `LIBRARY` and `INCLUDE` control statements to reference object data sets. If you have multiple secondary input data sets, concatenate them as shown in the following example:

  ```
  //SYSLIB DD DSNAME=CEE.SCEEBND2,DISP=SHR
  //       DD DSNAME=AREA.SALESLIB,DISP=SHR
  ```

- NOXPLINK with SCEELKEX and SCEELKED

  Run-time symbols will be resolved from longname definitions in SCEELKEX where available, otherwise resolved from load module stubs in SCEELKED.

  Add `LIBRARY` and `INCLUDE` control statements to reference object and load module library data sets. If you have multiple secondary input data sets, concatenate them as shown in the following example:

  ```
  //SYSLIB DD DSNAME=CEE.SCEELKEX,DISP=SHR
  //       DD DSNAME=CEE.SCEELKED,DISP=SHR
  //       DD DSNAME=AREA.SALESLIB,DISP=SHR
  ```

- NOXPLINK with SCEELKED

  Run-time symbols will be resolved from load module stubs in SCEELKED.

  Add `LIBRARY` and `INCLUDE` control statements to reference object and load module library data sets. If you have multiple secondary input data sets, concatenate them as shown in the following example:

  ```
  //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, as follows:

```
   ⋮

//SYSIN   DD DSNAME=myid.IPAOBJ,DISP=SHR
//        DD DSNAME=...
   ⋮

//        DD *
  INCLUDE ddname(member)
     LIBRARY ADDLIB(CPGM10)
/*
```

Figure 36. IPA Link Control Statements

Running the IPA Link Step in z/OS UNIX

Processing your application under z/OS UNIX System Services is the same as processing it under z/OS batch.

Using JCL

The example JCL, that follows, uses archive libraries and data sets. INCLUDE files may be PDS members, sequential files, or HFS files. Libraries may be partitioned data sets or archive libraries.

```
// jobcard
//*
//* THE FOLLOWING STEP IPA LINKS THE OBJECT FILES DEFINED BY DDOBJ1,
//* AND DDOBJ2 AND PLACES THE GENERATED OBJECT MODULE IN
//* USERID.WORK.IPAOBJ(TEST). AN IPA LINK LISTING IS GENERATED AND
//* DIRECTED TO SYSOUT=*.
//*
//IPALINK EXEC EDCI,
//     INFILE='SEE.SYSIN.OVERRIDE',
//     OUTFILE='USERID.WORK.IPAOBJ(TEST),DISP=SHR',
//     IPARM='IPA(MAP,LIST,DUP,ER,NONCAL)',
//     IREGSIZ=64M
//SYSLIB   DD  DSNAME=CEE.SCEELKED,DISP=SHR
//* object file
//DDOBJ1   DD PATH='/u/myuserid/callfoogoohoo.o',
//            PATHOPTS=(ORDONLY),
//            PATHDISP=(KEEP,KEEP)
//* PDS member
//DDOBJ2   DD DISP=SHR,DSN=MYUSERID.QAPARTNR.OBJ(MEM1)
//* archive library
//DDLIB3   DD PATH='/u/myuserid/mylibrary.a',
//            PATHOPTS=(ORDONLY),
//            PATHDISP=(KEEP,KEEP)
//* PDS Library
//DDLIB4   DD DISP=SHR,DSN=MYUSERID.QAPARTNR.OBJ
//SYSLIN   DD DISP=SHR,DSN=USERID.WORK.IPAOBJ(TEST)
//SYSOUT   DD  SYSOUT=*
//SYSCPRT  DD  SYSOUT=*
//SYSIN    DD *
  INCLUDE  DDOBJ1
  INCLUDE  DDOBJ2
  LIBRARY  DDLIB3
  LIBRARY  DDLIB4
/*
```

Figure 37. Using JCL for IPA Linking z/OS UNIX Non-XPLINK Applications

Note: For XPLINK, follow the example shown in Figure 33 on page 359 and Figure 34 on page 360.

Invoking IPA from the c89 Utility

The `c89` utility supports IPA. 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. You can specify `IPA` suboptions as comma-separated keywords.

If you invoke `c89` with a source file and the `-c` 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 source file hello.c:

```
c89 -c -WI hello.c
```

If you invoke `c89` with an object file, do not specify the `-c` option and do not specify any source files, `c89` automatically specifies `IPA(LINK)` and 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
```

If you invoke c89 with at least one source file and any number of object files, and do not specify the -c option, c89 automatically invokes the IPA Compile step once for each compilation unit and the IPA Link step once for the entire program. For example, the following command invokes the IPA Compile step, the IPA Link step, and the binder while creating program foo:

```
c89 -o foo -WI,object foo.c
```

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.
- If you specify -Wc, followed by compiler options, c89 passes those options only to the IPA Compile step.
- If you specify -Wl,I, followed by compiler options, c89 passes those options only to the IPA Link step.

The following is an example of passing options:

```
c89 -2 -WI,noobject -Wc,source -Wl,I,"maxmem(2048)" file.c
```

In this example, you pass the IPA(NOOBJECT) option to both the IPA Compile and IPA Link steps, the SOURCE option only to the IPA Compile step, and the MAXMEM(2048) option only to the IPA Link step.

When the XPLINK option is used, this option must be specified on all IPA Compile, non-IPA Compile, IPA Link, and bind steps. This ensures that appropriate input files are supplied to each step.

Using IPA Link with Archive Files

The IPA Link step supports all archive files, including those which are empty. Refer to “Processing the IPA Link Automatic Library Call” on page 345 for more information about the multiple attribute support introduced with z/OS V1R2.

Other Considerations

The compiler (which includes IPA) is packaged in MVS program object format, not in HFS executable format.

Refer to “Appendix F. c89 — Compile, link-edit and assemble a z/OS C program and create an executable file” on page 577 for more information about the c89 utility.

Chapter 10. Binding z/OS C/C++ Programs

This chapter describes how to bind your programs using the binder (the DFSMS/MVS 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 an HFS file. Depending on the environment you use, you can produce binder program objects as follows:

- For c89:

  If the targets of your executables are HFS 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` and/or `XPLINK` compiler options, you must use the binder.

For more information on the prelinker, see “Appendix A. Prelinking and Linking z/OS C/C++ Programs” on page 485.

When You Cannot Use the Binder

The following are the restrictions to 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 1.3, PDSEs are not supported. From CICS Transaction Server 1.3 onwards, there is support in CICS for PDSEs. Please refer to *CICS Transaction Server for OS/390 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 compiler option `IPA(NOLINK,OBJECT)` 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 compiler option `IPA(NOLINK,NOOBJECT)` 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 section shows you how to use 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 DLLs
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 38 on page 367 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.

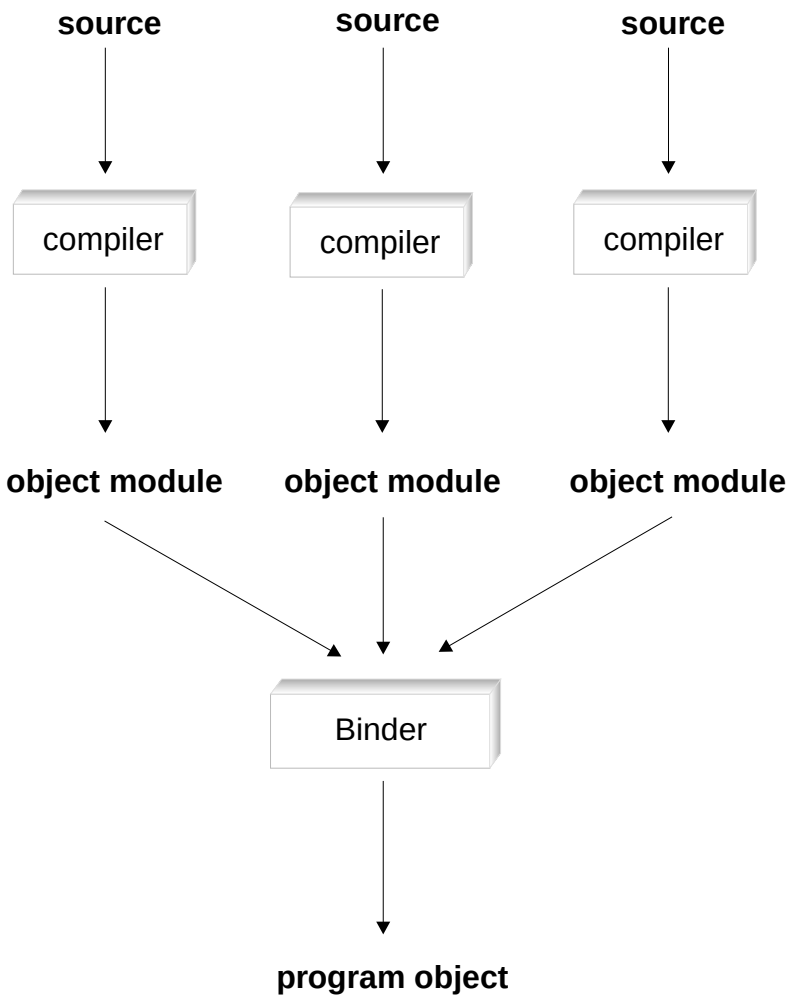


Figure 38. Single Final Bind

Bind Each Compile Unit

If you have changed the source in a compile unit, you can use the method that is shown in Figure 39 on page 368. 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.

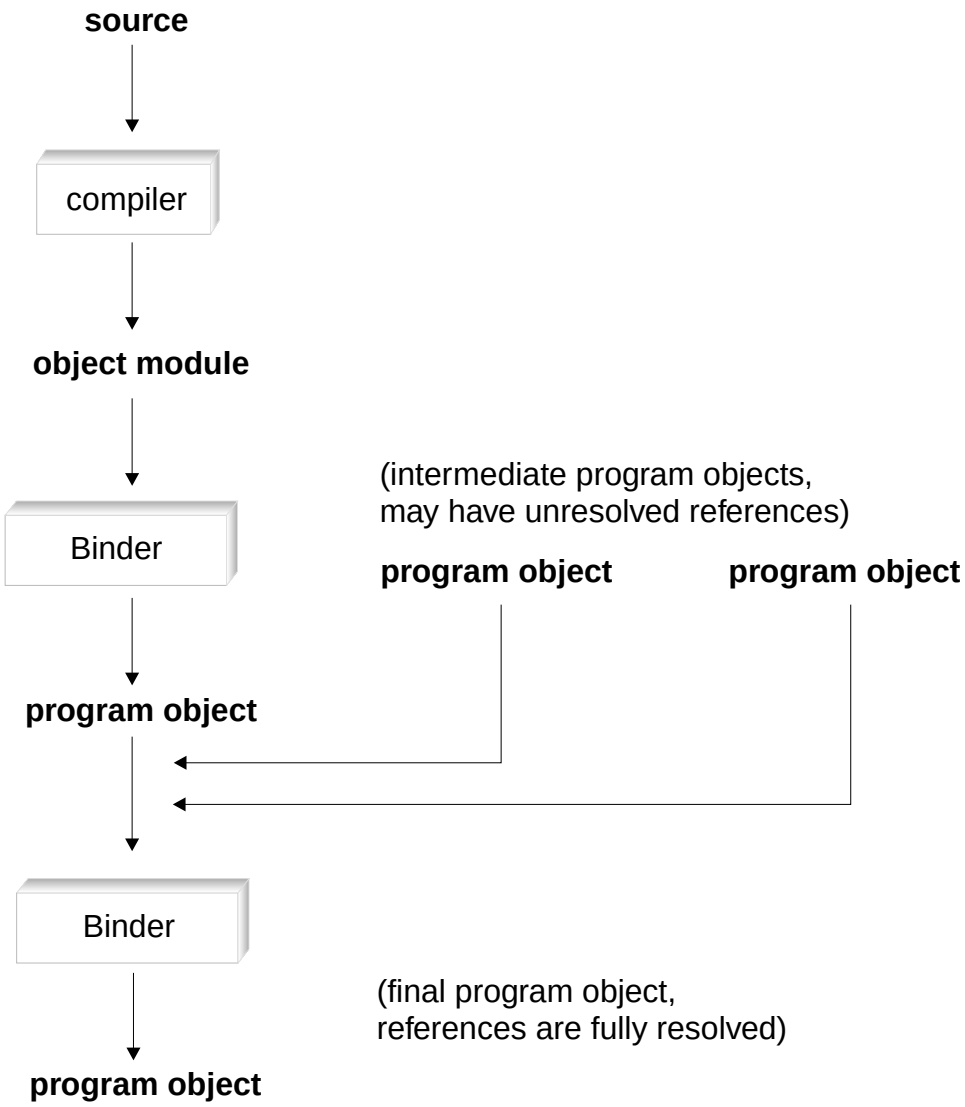


Figure 39. Bind Each Compile Unit

Build and Use a DLL

You can use the method that is shown in Figure 40 on page 369 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.

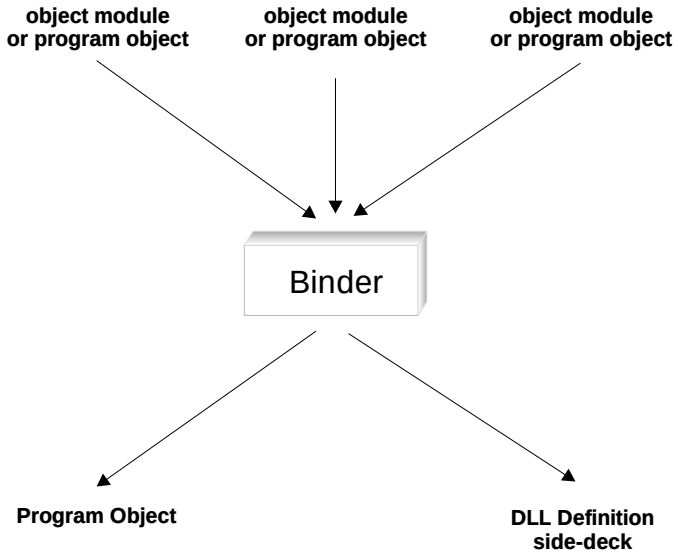


Figure 40. Build a DLL

You can use the method that is shown in Figure 41 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.

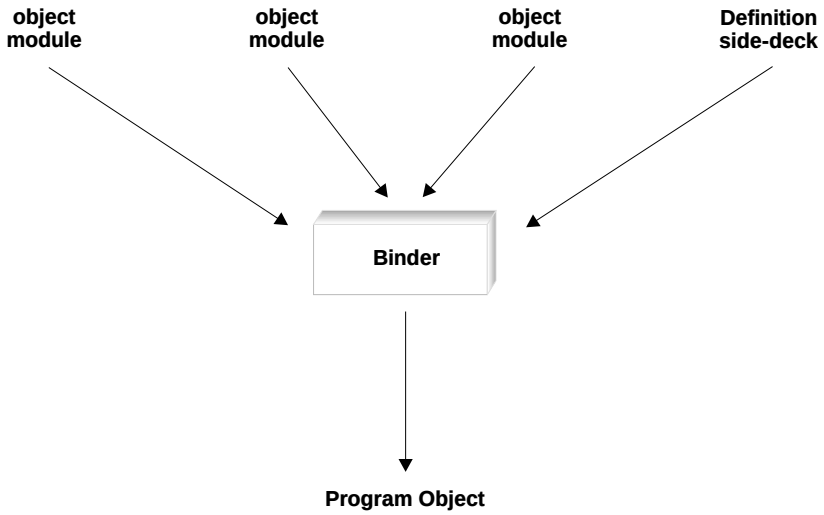


Figure 41. Build an application that uses a DLL

Rebind a Changed Compile Unit

You can use the method shown in Figure 42 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.

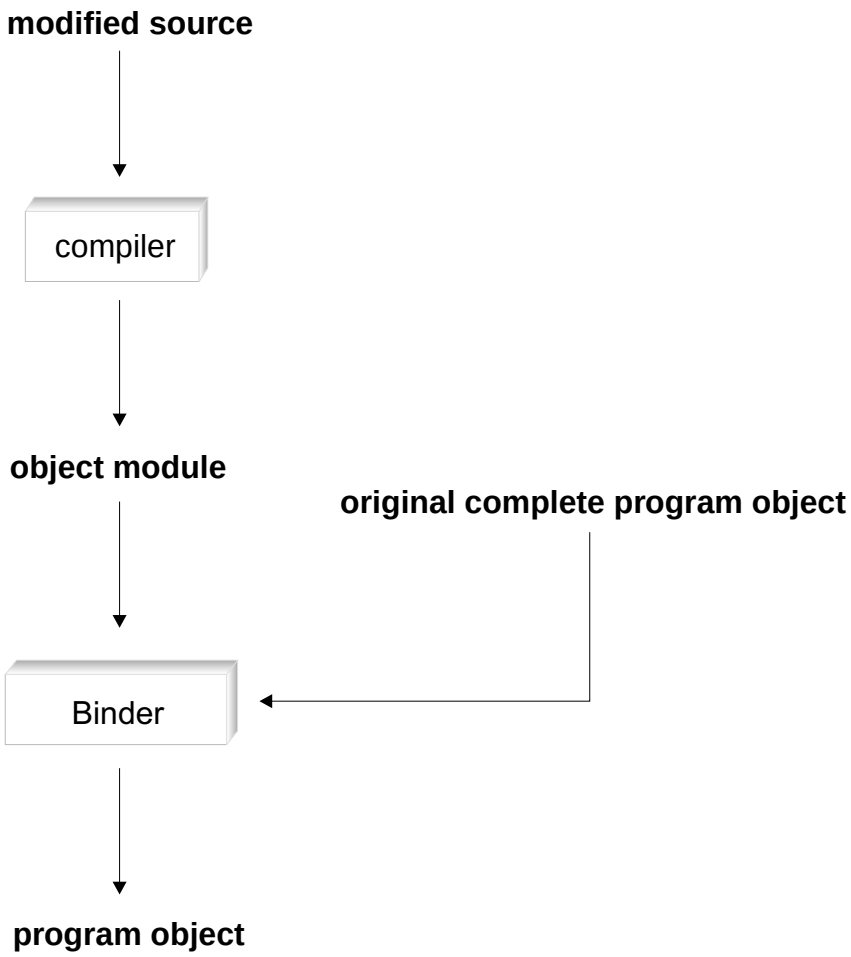


Figure 42. Rebinding a Changed Compile Unit

Binding Under z/OS UNIX

The `c89` and `c++` utilities are the interface to the compiler and the binder for z/OS UNIX System Services C/C++ applications. You can use `c89` and `c++`, to compile and bind a program in one step, or to bind application object modules after compilation.

Since OS/390 V2R4, the default, for the above utilities, is to invoke the binder alone, without first invoking the prelinker. That is, since the OS/390 V2R4 level of

OS/390 Language Environment 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 **{_STEPS}** environment variable in “Environment Variables” on page 589.

Typically, you invoke the c89 and c++ utilities from the z/OS shell. For more information on these utilities, see “Appendix F. c89 — Compile, link-edit and assemble a z/OS C program and create an executable file” on page 577 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 Example

The example source files unit0.c, unit1.c, and unit2.c that are shown in Figure 43, 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 43. Example Source Files

Single Final Bind Using c89

If you want the static IBM Open Class libraries, you must explicitly specify them (in other words, the default is the dynamic libraries) for the link-edit phase. 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 sets, as follows:

```
c++ ... -l "//'cbc.sclbcpp2'"
```

The OS/390 V2R10 version of the static library is in CBC.SCLBCPP. The z/OS V1R2 version of the static library is in CBC.SCLBCPP2, which is an upgrade from the OS/390 V2R10 version. For more information, refer to “Prelinking and Linking Under z/OS Batch” on page 504.

Compile each source file, then perform a final single bind of everything as follows:

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.

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.

Bind Each Compile Unit Using c89

Compile each source file and also bind it, then perform a final bind of all the partially bound units as follows:

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
   ```

The following is an example of a makefile to perform 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.

Build and Use a DLL Using c89

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.

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.

Rebind a Changed Compile Unit Using c89

Rebuild an application after making a change to a single source file. 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 compile time option `CSECT` to ensure 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 to perform 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 non-conventional filenames

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 ensures that such parts are not 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 “Appendix F. c89 — Compile, link-edit and assemble a z/OS C program and create an executable file” on page 577. 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, and none of the intermediate object modules are retained.

Binding under z/OS Batch

You can use the following procedures, which the z/OS C/C++ compiler supplies, to invoke the binder:

Procedure name	Description
CEEXL	C Bind an XPLINK Program
CEEXLR	C Bind and run an XPLINK program
EDCCB	C Compile and Bind steps
EDCCBG	C Compile, Bind, and Go steps
EDCXCB	C Compile and Bind an XPLINK Program
EDCXCBG	C Compile, Bind, and Go steps for an XPLINK Program
EDCXLDEF	Create C Source from a Locale, Compile, and Bind the XPLINK Program
CBCB	C++ Bind step
CBCBG	C++ Bind and Go steps
CBCCB	C++ Compile and Bind steps
CBCCBG	C++ Compile, Bind, and Go steps
CBCXB	C++ Bind an XPLINK Program
CBCXBG	C++ Bind and Go steps for an XPLINK Program
CBCXCB	C++ Compile and Bind an XPLINK Program
CBCXCBG	C++ Compile, Bind, and Go steps for an XPLINK Program

If you want to generate DLL code, you must use the binder DYNAM(DLL) option. All the z/OS 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 44 on page 376 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 44. Example Source Files

Single Final Bind under z/OS Batch

Compile each source file, then perform a final single bind of everything as follows:

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.

Bind Each Compile Unit under z/OS Batch

Compile each source file and also bind it, then perform a final bind of all the partially bound units as follows:

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.SCEEBIND. 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. Developers can then use these program objects and their changed object modules to create a development program object.

Build and Use a DLL under z/OS Batch

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.

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.

Rebind a Changed Compile Unit under z/OS Batch

Rebuild an application after making a change to a single source file. 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 compile time option `CSECT` to ensure 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. For example, you can specify the OPTIONS option on the PARM parameter to read binder options from the DD name 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 the example above, object module P1, which was compiled NOXPLINK, is bound using the IOSTREAM DLL definition sidedeck. The Language Environment non-XPLINK run-time libraries SCEELKED, SCEELKEX, and SCEECPP are statically bound to produce the program object PROG1. 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.SCEEBIND,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=*
```

The binder always requires three standard data sets. You must define these data sets on DD statements with the DDnames SYSLIN, SYSLMOD, and SYSPRINT.

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 is done above. 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.

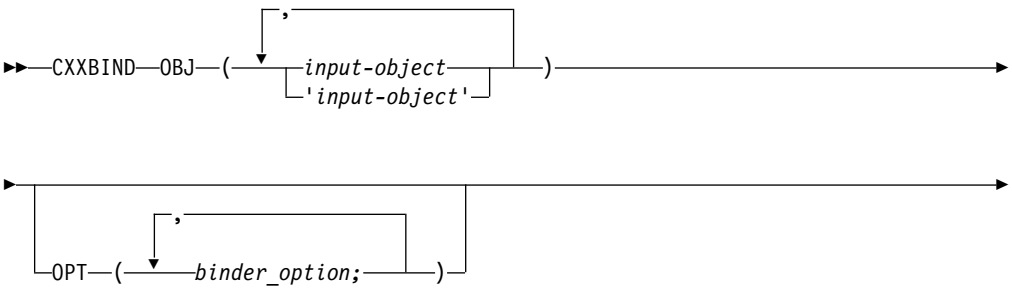
Binding Under TSO Using CXXBIND

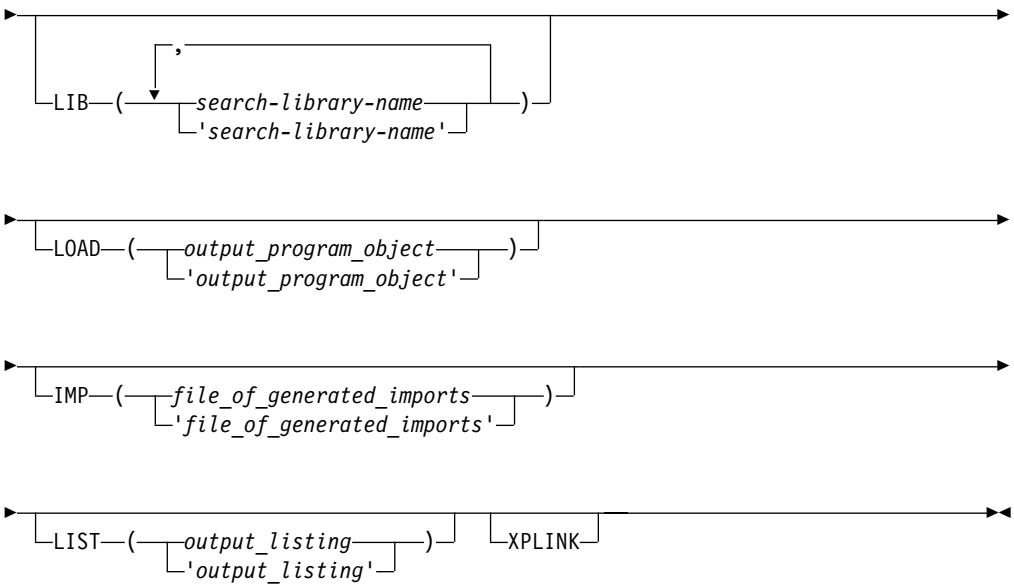
This section describes how to bind your z/OS C++ or z/OS C program in TSO by invoking the CXXBIND REXX EXEC. This REXX EXEC invokes the binder and creates an executable program object.

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 an HFS filename in an option, it must be an absolute filename; it must begin with a slash (/). You can include commas and special characters in filenames, but you must enclose filenames that contain special characters or commas in single quotes. If a single quote is part of the filename, you must specify the quote twice.

The syntax for the CXXBIND EXEC is:





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 an HFS file
  - A load module that is a PDS member
  - A program object that can be a PDSE member or an HFS file
  - A text file that contains binder statements. The file can be a PDS member, a sequential data set, or an HFS 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 C/C++ libraries CEE.SCEELKED, CEE.SCEELKEX, CEE.SCEECPP and the C++ class library CBC.SCLBSID. The default libraries that are used when the XPLINK option is specified are the C/C++ libraries CEE.SCEEBIND, CEE.SCEELIB and the C++ class library CBC.SCLBSID. 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 an HFS 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 45 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 45. Example Source Files

Single Final Bind Under TSO

Compile each source file, then perform a final single bind of everything as follows:

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.

Bind Each Compile Unit Under TSO

Compile and bind each source file, then perform a final bind of all the partially bound units as follows:

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. Developers can then use these program objects and their changed object modules to create a development program object.

Build and Use a DLL under TSO

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.

Rebind a Changed Compile Unit Under TSO

Rebuild an application after making a change to a single source file. 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 compile time option `CSECT` 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 11. Binder Processing

You can bind any z/OS C/C++ object module or program object.

Object files with longname 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 25.

Various limits have been increased from the linkage-editor. For example, the binder supports variable and function names up to 1024 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 runtime 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 runtime 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 using the C++ shell utility, use −Wl,DLL.

z/OS UNIX System Services HFS support allows library search of archive libraries that were created with the `ar` utility. HFS files can be specified on binder control statements.

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 99.

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 an HFS 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 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.

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 C or 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 411 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 370 and “Rebindability” on page 405.

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 for short. 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 “AUTOCALL Control Statement” on page 293. 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.
’PRINTF’.CEE.SCEELKEX has the equivalent case-sensitive longnamed 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.

XPLINK Libraries

The libraries described here are to be used only for binding XPLINK program modules.

For C and C++, you must include CEE.SCEEBIND 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 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 C++, 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.

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 "RENAME Control Statement" on page 296.
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 397.

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 section assumes that you specified the `DYNAM(DLL)` option. See “DYNAM(DLL | NO)” on page 289 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 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 initialize 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)
- An HFS 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 #pragmaexport directive or the EXPORTALL compiler option, or the C++ _Export keyword.

Output Listing

This section 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 section, the files USERID.PLAN9.OBJ(CU1) and /u/userid/plan9/cu2.o were bound together using the JCL shown in Figure 47 on page 395. Figure 46 on page 395 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 46. 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 47. 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.

The following example header was produced using the batch emulator:

```
OS/390 V2 R10 BINDER         09:08:20 WEDNESDAY MAY 10, 2000
BATCH EMULATOR  JOB(USERIDXX) STEP(BIND1   ) PGM= IEWL      PROCEDURE(BIND    )
```

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 “LIST(OFF | STMT | SUMMARY | NOIMP | ALL)” on page 290 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. 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 HFS file when it processes HFS pathnames 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 HFS filename.

```
          ***  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 391. 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

REFERENCE				TARGET			
CLASS OFFSET	SECT/PART(ABBREV)	ELEMENT OFFSET	TYPE	SYMBOL(ABBREV)	SECTION (ABBREV)	ELEMENT 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 HFS filename 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 402.

In the example above, you can see that c2 and c3 are to be dynamically linked

```
  ***  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 ***
```

from a DLL named DLLC. Also, this program exports variables Ax and ALongNamedThingVVWhichIsExported, and functions A3 and ALongNamedThingFFWhichIsExported.

Mangled to Demangled Symbol Cross Reference

The mangled to demangled name 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 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 HFS pathname 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 HFS 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 pathnames on control statements.

The binder creates a dummy `DDname` for each unique HFS file when it processes HFS pathnames 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 HFS filename.

The `DDname` vs pathname cross reference table shows the correspondence between the dummy `DDname` and its corresponding HFS filename. The table appears only if there is a generated `DDname`. Pathnames that you specified on JCL have user-assigned `DDname`s, 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 XOBJ object modules and is not applicable to `GOFF`.

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.

For example, suppose that our DLL consists of two compile units, `cu3.c` and `cu4.c`, that are bound using the JCL in Figure 48 on page 406:

```
/* 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 48. 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.

Error recovery

This section 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 391. Such inconsistencies arise from inconsistent usage of the LONGNAME and NOLONGNAME compiler options, and from multi-language programs that make symbol names uppercase. For 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. For 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 “LIBRARY Control Statement” on page 294.

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.

For 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. For 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). 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.

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)
```

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 option `REUS(NONE)`, 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 12. Running a C or C++ Application

This chapter gives an overview of how to run z/OS C/C++ programs under z/OS batch, TSO, and the z/OS Shell.

z/OS Language Environment provides a common runtime environment for C, C++, COBOL, PL/I, and FORTRAN. For detailed instructions on running existing and new z/OS C/C++ programs under z/OS Language Environment, refer to *z/OS Language Environment Programming Guide*. *z/OS C/C++ Programming Guide* also describes how to run z/OS C/C++ programs in a CICS environment.

Setting the Region Size for z/OS 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 current compiler default region size is 48M, but depending on your program and the degree of optimization you are using (i.e., OPT(2) and/or 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.

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 SCLBDLL and/or SCLBDLL2 at run time. The OS/390 V2R10 version of the DLL library is in CBC.SCLBDLL. The z/OS V1R2 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 STEPLIB 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 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 415.
- On the GPARM parameter of the cataloged procedures that are supplied by IBM. Refer to “Using Cataloged Procedures” on page 415.
- The #pragma runopts statement in your source code.
- The CEEUOPT facility that is provided by z/OS Language Environment.
- In the assembler user exit. For more information, refer to the *z/OS 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 interprets 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 interprets 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 297.

You should establish the required settings of the options for all z/OS 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 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

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 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 49. Running your program under z/OS Batch

Specifying Run-time Options in the EXEC Statement

You can specify run-time options in the `PARM` parameter of the `EXEC` statement as follows:

```
//[stepname] EXEC PGM=program_name,
//           PARM='[runtime options/][program parameters]'
```

For 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 C/C++ compiler to run your program. Each procedure listed below includes an execution step:

For z/OS C programs:

`EDCCBG`	Compile, bind, and run
`EDCXCBG`	Compile, bind, and execute an XPLINK C Program

For z/OS C++ programs:

`CBCBG`	Bind and run
`CBCCBG`	Compile, bind, and run
`CBCG`	Run
`CBCXBG`	Bind and run an `XPLINK` z/OS C++ program
`CBCXCBG`	Compile, bind, and run an `XPLINK` z/OS C++ program
`CBCXG`	Run an `XPLINK` z/OS C++ program

For more information on these cataloged procedures, see “Appendix D. Cataloged Procedures and REXX EXECs” on page 551.

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. For example:

```
//STEP   EXEC EDCCBG,INFILE='...',
//            GPARM='STACK(10K)/'
```

You can also use the GPARM parameter to pass arguments to the z/OS 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'
```

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  EDCCBG,...GPARM='/HFS file:/u/mike/cloudy.C'
```

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='/HFS file:/u/mike/cloudy.C'
```

See also “Specifying Run-Time Options” on page 297.

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.
- If you are using IBM Open Class Libraries, concatenate the SCLBDLL and/or SCLBDLL2 data set(s) to C++ STEPLIB, or add it to the LPA. The OS/390 V2R10 version of the DLL library is in CBC.SCLBDLL. The z/OS V1R2 version of the DLL library is in CBC.SCLBDLL2.

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 DD name 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 manual 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:

```
'[runtime 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 297.

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 C/C++ Application” for more information on passing mixed case parameters.

Passing Arguments to the z/OS 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 88 or the *C/C++ Language Reference*.

The case of the characters in argv depends on you invoked how your z/OS C/C++ program, as shown in the following table.

Table 46. Case sensitivity of arguments under TSO

How the z/OS 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

This section discusses how to run your z/OS UNIX System Services C/C++ application.

You must have the Language Environment Library SCEERUN available before you try to run your application under z/OS UNIX. 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. If your application was bound with the DLL Class Libraries, you must supply SCLBDLL and/or SCLBDLL2 at run time. The OS/390 V2R10 version of the DLL library is in CBC.SCLBDLL. The z/OS V1R2 version of the DLL library is in CBC.SCLBDLL2.

z/OS UNIX Application Environments

You can run your z/OS UNIX System Services 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 an HFS file from the TSO/E `READY` prompt, you must use the `BPXBATCH` utility.
- z/OS batch

  To run an application program that resides in an HFS 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 an HFS 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

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 C/C++ application program that resides in an HFS file. Any programs you intend to run from the file system must be 31-bit AMODE, problem program state, PSW key 8 programs. If you plan to run a 24-bit AMODE z/OS C/C++ program from within an application, ensure that the executable resides in a PDS or PDSE member.

Any new z/OS UNIX System Services z/OS C/C++ applications you develop should be 31-bit AMODE.

Copying Applications between a PDS and HFS

If you have a C/C++ application as a PDS member and want to place it in the HFS, you can use the z/OS UNIX System Services TSO/E command `OPUTX` to copy the member into an HFS file.

If you have a C/C++ application as an HFS file and want to place it in a PDS, you can use the z/OS UNIX System Services TSO/E command `OGETX` to copy the HFS 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 “Appendix F. c89 — Compile, link-edit and assemble a z/OS C program and create an executable file” on page 577. For examples of using these commands to copy data sets to HFS 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 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 under 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 Applications under z/OS Batch

Using the BPXBATCH Utility

Use the IBM-supplied `BPXBATCH` program to run a C/C++ application under z/OS batch from an HFS 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 HFS files. You should also allocate stdout and stderr HFS files for writing program output such as error messages. Allocate the standard files using the PATH options on TSO/E ALLOCATE command or the JCL DD statement.

For more information on the BPXBATCH program, refer to “Chapter 19. BPXBATCH Utility” on page 479.

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 50 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 50. 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 stdout and stderr allocated files 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, and write error messages to the file /myap/std/my.err, 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-HFS z/OS UNIX Executable to Run under z/OS 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-HFS program object (PDSE

member), it may 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)/'`.

Part 4. Utilities and Tools

This section contains information about the utilities and tools that you can use under z/OS.

Chapter 13. Object Library Utility

This chapter describes how to use the Object Library Utility to update libraries of object modules. On z/OS, a library is a PDS or PDSE with object modules as members.

Object libraries provide convenient packaging of object modules. With the Object Library Utility, a library can contain objects modules compiled with long names, short names, writable static data, XPLINK, or IPA. The Object Library Utility stores source member symbol information with different attributes. This information is stored in two special members of the library, the Basic Directory Member (@@DC370$) and the Enhanced Directory Member(@@DC390$). Both could be referred to as the EDCALIAS directory. This information is stored in a special member of the library that this chapter refers to as the C370LIB or EDCALIAS directory.

Commands are available to add object modules to a library, to delete object modules from a library, or to build the C370LIB directory for a library. Use the DIR command to build the C370LIB directory for a library of object modules. Use the MAP command to list the contents of the C370LIB directory.

You can create an object library under z/OS batch and TSO.

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)

For more information on the data sets that you use with the Object Library Utility, see “Description of Data Sets Used” on page 555.

To compile the z/OS C program WALTER.SOURCE(SUB1) for long names and add to WALTER.SOURCE.OBJ(SUB1), use the following JCL. The Object Library Utility directory for the library, WALTER.SOURCE.OBJ, is updated in the process.

```
//COMPILE EXEC EDCCLIB,INFILE='WALTER.SOURCE(SUB1)',CPARM='LO',
//      LIBRARY='WALTER.SOURCE.OBJ',MEMBER='SUB1'
```

If you request a map for the library WALTER.SOURCE.OBJ, use the following:

```
//OBJLIB EXEC EDCLIB,OPARM='MAP',LIBRARY='WALTER.SOURCE.OBJ'
```

For z/OS C++, use the EDCLIB cataloged procedure. 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 section shows you how to specify these options under z/OS batch.

The following example creates a new C370LIB directory. If the directory already exists, it is updated.

```
//DIRDIR   EXEC EDCLIB,
//         LIBRARY='LUCKY13.CXX.OBJMATH',
//         OPARM='DIR'
```

To create a map:

```
//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. For example, to add a new member named MA191:

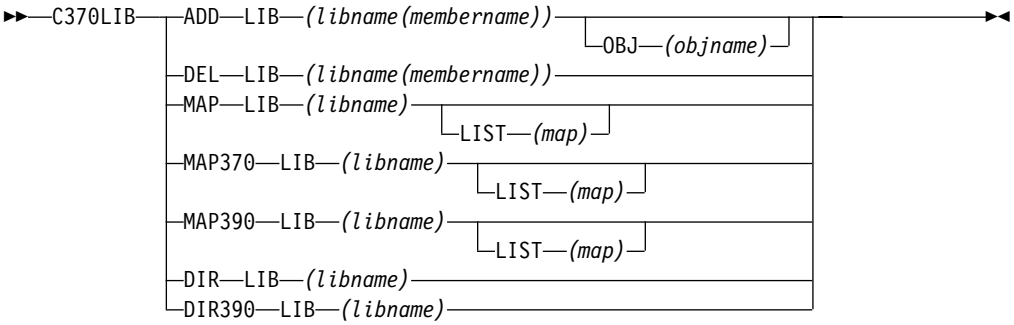
```
//ADDDIR   EXEC EDCLIB,
//         LIBRARY='LUCKY13.CXX.OBJMATH',
//         OPARM='ADD MA191',
//         OBJECT='DSNAME=LUCKY13.CXX.OBJ(OBJ191),DISP=SHR'
```

To delete a member from an object library, use the DEL option to keep the directory up to date. For 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:



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 the Basic Directory Member.

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 C370LIB.
DIR390	Starting in z/OS V1R2, DIR and DIR390 are the same, which means that both force the generation of the Enhanced Directory Member (@@DC390$).
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, 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 a Object Library Utility-directory is to be built.
LIST(*map*)	Specifies the data set that is to contain the library map. If you specified an asterisk (*), the library map 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 map is FRANK.TEST.OBJ.MAP.

Under TSO, for z/OS 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 or MAP390 command. The listing contains information on each member of the library.

```
================================================================================
| 1                      Object Library Utility Map                            |
|                                                                              |
|C370LIB:5647A01 V2 R10 M0 IBM Language Environment 2001/01/03 15:10:51        |
================================================================================

 2  Library Name: FRANK.A.OBJLIB

*------------------------------------------------------------------------------*
* 3  Member Name:    CGOFF                        (P) 2001/00/00 13:51:23 *
*                                                   5647A01    V2 R10   *
*------------------------------------------------------------------------------*

   User Comment:
AGGRCOPY(NOOVERLAP) NOALIAS ANSIALIAS ARCH(2) ARGPARSE NOCOMPACT
NOCOMPRESS NOCONVLIT CSECT() NODLL(NOCALLBACKANY) EXECOPS NOEXPORTALL
FLOAT(HEX, FOLD, NOAFP) GOFF NOGONUMBER NOHWOPTS NOIGNERRNO NOINITAUTO
NOINLINE NOIPA LANGLVL(*EXTENDED) NOLIBANSI NOLOCALE LONGNAME
MAXMEM(2097152) OPTIMIZE(0) PLIST(HOST) REDIR RENT NOROCONST NOROSTRING
NOSERVICE SPILL(128) START STRICT NOSTRICT_INDUCTION
TARGET(LE, OSV2R10) NOTEST TUNE(3) NOUPCONV NOXPLINK COMPILED_ON_MVS

 4   (   L) Function Name: CSTUFF#C
    (  WL) External Name: this_int_is_in_writable_static_and_its_name_wi
                          ll_warp_because_it_is_too_long
    (   L) Function Name: foo
    (  WL) External Name: CSTUFF#T
    (  WL) External Name: CSTUFF#S

*------------------------------------------------------------------------------*
*  Member Name: CPPIPANO                      (P) 2001/01/03 15:10:51 *
*                                                 5647A01    V2 R10   *
*------------------------------------------------------------------------------*

   User Comment:
AGGRCOPY(NOOVERLAP) ANSIALIAS ARCH(2) ARGPARSE NOCOMPACT NOCOMPRESS
NOCONVLIT NOCSECT CVFT DLL(NOCALLBACKANY) EXECOPS NOEXPORTALL
FLOAT(HEX, FOLD, NOAFP) NOGOFF NOGONUMBER NOIGNERRNO NOINITAUTO
IPA(NOLINK, NOOBJECT, COMPRESS, OPTIMIZE, NOGONUM) LANGLVL(EXTENDED)
NOLIBANSI NOLOCALE LONGNAME MAXMEM(2097152) NOOPTIMIZE PLIST(HOST)
REDIR NOROCONST ROSTRING ROUND(Z) NOSERVICE SPILL(128) START STRICT
NOSTRICT_INDUCTION TARGET(LE, OSV2R10) NOTEST(HOOK) TUNE(3) NOXPLINK
COMPILED_ON_MVS

    (I  L) Function Name: testeh()
    (I  L) Function Name: f1()
    (I  L) Function Name: a()
    (I  L) Function Name: A::areallyreallyreallyreallyreallyreallyreally
                          longnamefunction()
    (I  L) Function Name: A::operator+=(int)
    (I  L) Function Name: A::x()
    (I  L) External Name: i1
    (I  L) External Name: i2

*------------------------------------------------------------------------------*
*  Member Name:  CPPIPAO                      (P) 2001/01/03 15:10:51 *
*                                                 5647A01    V2 R10   *
*------------------------------------------------------------------------------*
```

```
   User Comment:
of IPA OBJECT AGGRCOPY(NOOVERLAP) ANSIALIAS ARCH(2) ARGPARSE NOCOMPACT
NOCOMPRESS NOCONVLIT NOCSECT CVFT DLL(NOCALLBACKANY) EXECOPS
NOEXPORTALL FLOAT(HEX, FOLD, NOAFP) NOGOFF NOGONUMBER NOIGNERRNO
NOINITAUTO IPA(NOLINK, OBJECT, COMPRESS, OPTIMIZE, NOGONUM)
LANGLVL(EXTENDED) NOLIBANSI NOLOCALE LONGNAME MAXMEM(2097152)
NOOPTIMIZE PLIST(HOST) REDIR NOROCONST ROSTRING ROUND(Z) NOSERVICE
SPILL(128) START STRICT NOSTRICT_INDUCTION TARGET(LE, OSV2R10)
NOTEST(HOOK) TUNE(3) NOXPLINK COMPILED_ON_MVS of OBJECT

    (   L) Function Name: testeh()
    (   L) Function Name: f1()
    (   L) Function Name: a()
    (   L) Function Name: A::areallyreallyreallyreallyreallyreallyreally
                          longnamefunction()
    (   L) Function Name: A::operator+=(int)
    (   L) Function Name: A::x()
    (  WL) External Name: i1
    (  WL) External Name: i2

*-------------------------------------------------------------------------*
*  Member Name: CPPIPAOB                       (P) 2001/01/03 15:10:51 *
*                                                  5647A01    V2 R10   *
*-------------------------------------------------------------------------*

   User Comment:
AGGRCOPY(NOOVERLAP) ANSIALIAS ARCH(2) ARGPARSE NOCOMPACT NOCOMPRESS
NOCONVLIT NOCSECT CVFT DLL(NOCALLBACKANY) EXECOPS NOEXPORTALL
FLOAT(HEX, FOLD, NOAFP) NOGOFF NOGONUMBER NOIGNERRNO NOINITAUTO
IPA(NOLINK, OBJONLY, COMPRESS, OPTIMIZE, NOGONUM) LANGLVL(EXTENDED)
NOLIBANSI NOLOCALE LONGNAME MAXMEM(2097152) NOOPTIMIZE PLIST(HOST)
REDIR NOROCONST ROSTRING ROUND(Z) NOSERVICE SPILL(128) START STRICT
NOSTRICT_INDUCTION TARGET(LE, OSV2R10) NOTEST(HOOK) TUNE(3) NOXPLINK
COMPILED_ON_MVS of OBJECT

    (   L) Function Name: testeh()
    (   L) Function Name: f1()
    (   L) Function Name: a()
    (   L) Function Name: A::areallyreallyreallyreallyreallyreallyreally
                          longnamefunction()
    (   L) Function Name: A::operator+=(int)
    (   L) Function Name: A::x()
    (  WL) External Name: i1
    (  WL) External Name: i2

*-------------------------------------------------------------------------*
*  Member Name: CPPXPLNK                       (P) 2001/00/00 13:51:25 *
*                                                  5647A01    V2 R10   *
*-------------------------------------------------------------------------*

   User Comment:
AGGRCOPY(NOOVERLAP) ANSIALIAS ARCH(2) ARGPARSE NOCOMPACT NOCOMPRESS
NOCONVLIT CSECT(CODE, CPPSTUFF#C) CSECT(STATIC, CPPSTUFF#S)
CSECT(TEST, CPPSTUFF#T) CVFT DLL(NOCALLBACKANY) EXECOPS NOEXPORTALL
FLOAT(HEX, FOLD, NOAFP) GOFF NOGONUMBER NOIGNERRNO NOINITAUTO NOIPA
LANGLVL(EXTENDED) NOLIBANSI NOLOCALE LONGNAME MAXMEM(2097152)
NOOPTIMIZE PLIST(HOST) REDIR NOROCONST ROSTRING ROUND(Z) NOSERVICE
SPILL(128) START STRICT NOSTRICT_INDUCTION TARGET(LE, OSV2R10)
NOTEST(HOOK) TUNE(3) XPLINK COMPILED_ON_MVS
```

```
    ( X L) Function Name: testeh()
    ( X L) Function Name: f1()
    ( X L) Function Name: a()
    ( X L) Function Name: A::areallyreallyreallyreallyreallyreallyreally
                          longnamefunction()
    ( X L) Function Name: A::operator+=(int)
    ( X L) Function Name: A::x()
    (  WL) External Name: i1
    (  WL) External Name: i2
    ( X L) Function Name: CPPSTUFF#C
    (  WL) External Name: CPPSTUFF#T
    (  WL) External Name: CPPSTUFF#S

*---------------------------------------------------------------------*
*  Member Name:    CXOBJ                       (P) 2001/01/03 15:10:51 *
*                                                  5647A01    V2 R10   *
*---------------------------------------------------------------------*

   User Comment:
AGGRCOPY(NOOVERLAP) NOALIAS ANSIALIAS ARCH(2) ARGPARSE NOCOMPACT
NOCOMPRESS NOCONVLIT NOCSECT NODLL(NOCALLBACKANY) EXECOPS NOEXPORTALL
FLOAT(HEX, FOLD, NOAFP) NOGOFF NOGONUMBER NOHWOPTS NOIGNERRNO
NOINITAUTO NOINLINE NOIPA LANGLVL(*EXTENDED) NOLIBANSI NOLOCALE
LONGNAME MAXMEM(2097152) OPTIMIZE(0) PLIST(HOST) REDIR RENT NOROCONST
NOROSTRING NOSERVICE SPILL(128) START STRICT NOSTRICT_INDUCTION
TARGET(LE, OSV2R10) NOTEST TUNE(3) NOUPCONV NOXPLINK COMPILED_ON_MVS

    (  WL) External Name: this_int_is_in_writable_static_and_its_name_wi
                          ll_warp_because_it_is_too_long
    (   L) Function Name: foo


=======================================================================
|                      Symbol Definition Map                          |
=======================================================================


*---------------------------------------------------------------------*
| 5  Symbol name: CSTUFF#C                                            |
*---------------------------------------------------------------------*

  6  From member:    CGOFF Type: Function (   L)

*---------------------------------------------------------------------*
|  Symbol name: this_int_is_in_writable_static_and_its_name_will_warp_ |
|               because_it_is_too_long                                |
*---------------------------------------------------------------------*

  From member:    CGOFF Type: External (  WL)
  From member:    CXOBJ Type: External (  WL)

*---------------------------------------------------------------------*
|  Symbol name: foo                                                   |
*---------------------------------------------------------------------*

  From member:    CGOFF Type: Function (   L)
  From member:    CXOBJ Type: Function (   L)
```

```
*-------------------------------------------------------------------------*
|  Symbol name: CSTUFF#T                                                  |
*-------------------------------------------------------------------------*

  From member:    CGOFF Type: External (  WL)

*-------------------------------------------------------------------------*
|  Symbol name: CSTUFF#S                                                  |
*-------------------------------------------------------------------------*

  From member:    CGOFF Type: External (  WL)

*-------------------------------------------------------------------------*
|  Symbol name: testeh()                                                  |
*-------------------------------------------------------------------------*

  From member:  CPPIPAO Type: Function (   L)
  From member: CPPIPAOB Type: Function (   L)
  From member: CPPXPLNK Type: Function ( X L)
  From member: CPPIPANO Type: Function (I  L)

*-------------------------------------------------------------------------*
|  Symbol name: f1()                                                      |
*-------------------------------------------------------------------------*

  From member:  CPPIPAO Type: Function (   L)
  From member: CPPIPAOB Type: Function (   L)
  From member: CPPXPLNK Type: Function ( X L)
  From member: CPPIPANO Type: Function (I  L)

*-------------------------------------------------------------------------*
|  Symbol name: a()                                                       |
*-------------------------------------------------------------------------*

  From member:  CPPIPAO Type: Function (   L)
  From member: CPPIPAOB Type: Function (   L)
  From member: CPPXPLNK Type: Function ( X L)
  From member: CPPIPANO Type: Function (I  L)

*-------------------------------------------------------------------------*
|  Symbol name: A::areallyreallyreallyreallyreallyreallyreallylongname    |
|               function()                                                |
*-------------------------------------------------------------------------*

  From member:  CPPIPAO Type: Function (   L)
  From member: CPPIPAOB Type: Function (   L)
  From member: CPPXPLNK Type: Function ( X L)
  From member: CPPIPANO Type: Function (I  L)

*-------------------------------------------------------------------------*
|  Symbol name: A::operator+=(int)                                        |
*-------------------------------------------------------------------------*

  From member:  CPPIPAO Type: Function (   L)
  From member: CPPIPAOB Type: Function (   L)
  From member: CPPXPLNK Type: Function ( X L)
  From member: CPPIPANO Type: Function (I  L)
```

```
*-------------------------------------------------------------------------*
|  Symbol name: A::x()                                                    |
*-------------------------------------------------------------------------*

  From member:  CPPIPAO Type: Function (   L)
  From member: CPPIPAOB Type: Function (   L)
  From member: CPPXPLNK Type: Function ( X L)
  From member: CPPIPANO Type: Function (I  L)

*-------------------------------------------------------------------------*
|  Symbol name: i1                                                        |
*-------------------------------------------------------------------------*

  From member:  CPPIPAO Type: External (  WL)
  From member: CPPIPAOB Type: External (  WL)
  From member: CPPXPLNK Type: External (  WL)
  From member: CPPIPANO Type: External (I  L)

*-------------------------------------------------------------------------*
|  Symbol name: i2                                                        |
*-------------------------------------------------------------------------*

  From member:  CPPIPAO Type: External (  WL)
  From member: CPPIPAOB Type: External (  WL)
  From member: CPPXPLNK Type: External (  WL)
  From member: CPPIPANO Type: External (I  L)

*-------------------------------------------------------------------------*
|  Symbol name: CPPSTUFF#C                                                |
*-------------------------------------------------------------------------*

  From member: CPPXPLNK Type: Function ( X L)

*-------------------------------------------------------------------------*
|  Symbol name: CPPSTUFF#T                                                |
*-------------------------------------------------------------------------*

  From member: CPPXPLNK Type: External (  WL)

*-------------------------------------------------------------------------*
|  Symbol name: CPPSTUFF#S                                                |
*-------------------------------------------------------------------------*

  From member: CPPXPLNK Type: External (  WL)

=========  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 contains information on each member of the library that is in XOBJ format.

```
=============================================================================
| 1                      Object Library Utility Map                         |
|                                                                           |
|C370LIB:5647A01 V2 R10 M0 IBM Language Environment 2001/01/03 15:10:51|
=============================================================================

 2  Library Name: FRANK.A.OBJLIB

*---------------------------------------------------------------------------*
* 3  Member Name:    CGOFF                      (P) 2001/00/00 13:51:23 *
*---------------------------------------------------------------------------*

*---------------------------------------------------------------------------*
*  Member Name: CPPIPANO                     (P) 2001/01/03 15:10:51 *
*                                               5647A01    V2 R10   *
*---------------------------------------------------------------------------*

   User Comment:
AGGRCOPY(NOOVERLAP) ANSIALIAS ARCH(2) ARGPARSE NOCOMPACT NOCOMPRESS
NOCONVLIT NOCSECT CVFT DLL(NOCALLBACKANY) EXECOPS NOEXPORTALL
FLOAT(HEX, FOLD, NOAFP) NOGOFF NOGONUMBER NOIGNERRNO NOINITAUTO
IPA(NOLINK, NOOBJECT, COMPRESS, OPTIMIZE, NOGONUM) LANGLVL(EXTENDED)
NOLIBANSI NOLOCALE LONGNAME MAXMEM(2097152) NOOPTIMIZE PLIST(HOST)
REDIR NOROCONST ROSTRING ROUND(Z) NOSERVICE SPILL(128) START STRICT
NOSTRICT_INDUCTION TARGET(LE, OSV2R10) NOTEST(HOOK) TUNE(3) NOXPLINK
COMPILED_ON_MVS

 4   (   L) Function Name: testeh()
    (   L) Function Name: f1()
    (   L) Function Name: a()
    (   L) Function Name: A::areallyreallyreallyreallyreallyreallyreally
                          longnamefunction()
    (   L) Function Name: A::operator+=(int)
    (   L) Function Name: A::x()
    (   L) External Name: i1
    (   L) External Name: i2

*---------------------------------------------------------------------------*
*  Member Name:  CPPIPAO                     (P) 2001/01/03 15:10:51 *
*                                               5647A01    V2 R10   *
*---------------------------------------------------------------------------*

   User Comment:
of IPA OBJECT AGGRCOPY(NOOVERLAP) ANSIALIAS ARCH(2) ARGPARSE NOCOMPACT
NOCOMPRESS NOCONVLIT NOCSECT CVFT DLL(NOCALLBACKANY) EXECOPS
NOEXPORTALL FLOAT(HEX, FOLD, NOAFP) NOGOFF NOGONUMBER NOIGNERRNO
NOINITAUTO IPA(NOLINK, OBJECT, COMPRESS, OPTIMIZE, NOGONUM)
LANGLVL(EXTENDED) NOLIBANSI NOLOCALE LONGNAME MAXMEM(2097152)
NOOPTIMIZE PLIST(HOST) REDIR NOROCONST ROSTRING ROUND(Z) NOSERVICE
SPILL(128) START STRICT NOSTRICT_INDUCTION TARGET(LE, OSV2R10)
NOTEST(HOOK) TUNE(3) NOXPLINK COMPILED_ON_MVS of OBJECT

    (  WL) External Name: i1
    (  WL) External Name: i2
    (   L) Function Name: testeh()
    (   L) Function Name: f1()
    (   L) Function Name: a()
    (   L) Function Name: A::operator+=(int)
    (   L) Function Name: A::areallyreallyreallyreallyreallyreallyreally
                          longnamefunction()
    (   L) Function Name: A::x()
```

```
*--------------------------------------------------------------------*
*  Member Name: CPPIPAOB                      (P) 2001/01/03 15:10:51 *
*                                                 5647A01    V2 R10   *
*--------------------------------------------------------------------*

   User Comment:
AGGRCOPY(NOOVERLAP) ANSIALIAS ARCH(2) ARGPARSE NOCOMPACT NOCOMPRESS
NOCONVLIT NOCSECT CVFT DLL(NOCALLBACKANY) EXECOPS NOEXPORTALL
FLOAT(HEX, FOLD, NOAFP) NOGOFF NOGONUMBER NOIGNERRNO NOINITAUTO
IPA(NOLINK, OBJONLY, COMPRESS, OPTIMIZE, NOGONUM) LANGLVL(EXTENDED)
NOLIBANSI NOLOCALE LONGNAME MAXMEM(2097152) NOOPTIMIZE PLIST(HOST)
REDIR NOROCONST NOSTRING ROUND(Z) NOSERVICE SPILL(128) START STRICT
NOSTRICT_INDUCTION TARGET(LE, OSV2R10) NOTEST(HOOK) TUNE(3) NOXPLINK
COMPILED_ON_MVS of OBJECT

    (  WL) External Name: i1
    (  WL) External Name: i2
    (   L) Function Name: testeh()
    (   L) Function Name: f1()
    (   L) Function Name: a()
    (   L) Function Name: A::areallyreallyreallyreallyreallyreallyreally
                          longnamefunction()
    (   L) Function Name: A::operator+=(int)
    (   L) Function Name: A::x()

*--------------------------------------------------------------------*
*  Member Name: CPPXPLNK                      (P) 2001/00/00 13:51:25 *
*--------------------------------------------------------------------*

*--------------------------------------------------------------------*
*  Member Name:    CXOBJ                      (P) 2001/01/03 15:10:51 *
*                                                 5647A01    V2 R10   *
*--------------------------------------------------------------------*

   User Comment:
AGGRCOPY(NOOVERLAP) NOALIAS ANSIALIAS ARCH(2) ARGPARSE NOCOMPACT
NOCOMPRESS NOCONVLIT NOCSECT NODLL(NOCALLBACKANY) EXECOPS NOEXPORTALL
FLOAT(HEX, FOLD, NOAFP) NOGOFF NOGONUMBER NOHWOPTS NOIGNERRNO
NOINITAUTO NOINLINE NOIPA LANGLVL(*EXTENDED) NOLIBANSI NOLOCALE
LONGNAME MAXMEM(2097152) OPTIMIZE(0) PLIST(HOST) REDIR RENT NOROCONST
NOROSTRING NOSERVICE SPILL(128) START STRICT NOSTRICT_INDUCTION
TARGET(LE, OSV2R10) NOTEST TUNE(3) NOUPCONV NOXPLINK COMPILED_ON_MVS

    (  WL) External Name: this_int_is_in_writable_static_and_its_name_wi
                          ll_warp_because_it_is_too_long
    (   L) Function Name: foo

================================================================================
| 5                      Symbol Definition Map                                 |
================================================================================


*--------------------------------------------------------------------*
```

```
*-------------------------------------------------------------------------*
| 6  Symbol name: testeh()                                                 |
*-------------------------------------------------------------------------*

  From member: CPPIPANO Type: Function (   L)
  From member:  CPPIPAO Type: Function (   L)
  From member: CPPIPAOB Type: Function (   L)

*-------------------------------------------------------------------------*
|  Symbol name: f1()                                                      |
*-------------------------------------------------------------------------*

  From member: CPPIPANO T-ype: Function (   L)
  From member:  CPPIPAO Type: Function (   L)
  From member: CPPIPAOB Type: Function (   L)

*-------------------------------------------------------------------------*
|  Symbol name: a()                                                       |
*-------------------------------------------------------------------------*

  From member: CPPIPANO Type: Function (   L)
  From member:  CPPIPAO Type: Function (   L)
  From member: CPPIPAOB Type: Function (   L)

*-------------------------------------------------------------------------*
|  Symbol name: A::areallyreallyreallyreallyreallyreallyreallylongname    |
|               function()                                                |
*-------------------------------------------------------------------------*

  From member: CPPIPANO Type: Function (   L)
  From member:  CPPIPAO Type: Function (   L)
  From member: CPPIPAOB Type: Function (   L)

*-------------------------------------------------------------------------*
|  Symbol name: A::operator+=(int)                                        |
*-------------------------------------------------------------------------*

  From member: CPPIPANO Type: Function (   L)
  From member:  CPPIPAO Type: Function (   L)
  From member: CPPIPAOB Type: Function (   L)

*-------------------------------------------------------------------------*
|  Symbol name: A::x()                                                    |
*-------------------------------------------------------------------------*

  From member: CPPIPANO Type: Function (   L)
  From member:  CPPIPAO Type: Function (   L)
  From member: CPPIPAOB Type: Function (   L)
```

```
*-------------------------------------------------------------------------*
|  Symbol name: i1                                                        |
*-------------------------------------------------------------------------*

  From member: CPPIPANO Type: External (   L)
  From member:  CPPIPAO Type: External (  WL)
  From member: CPPIPAOB Type: External (  WL)

*-------------------------------------------------------------------------*
|  Symbol name: i2                                                        |
*-------------------------------------------------------------------------*

  From member: CPPIPANO Type: External (   L)
  From member:  CPPIPAO Type: External (  WL)
  From member: CPPIPAOB Type: External (  WL)

*-------------------------------------------------------------------------*
|  Symbol name: this_int_is_in_writable_static_and_its_name_will_warp_    |
|               because_it_is_too_long                                    |
*-------------------------------------------------------------------------*

  From member:    CXOBJ Type: External (  WL)

*-------------------------------------------------------------------------*
|  Symbol name: foo                                                       |
*-------------------------------------------------------------------------*

  From member:    CXOBJ Type: Function (   L)

=========  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 product number of the processor that produced the object module follows the name of the object module member. If the `END` record in the object module does not have the processor information in the appropriate format, the `Processor ID` field does not appear.

The `Timestamp` field appears in *yyyy/mm/dd* format. A letter that is enclosed in parentheses indicates the meaning of the timestamp. That is, the Object Library Utility retains a timestamp for each member and selects the time according to the following hierarchy:

(P) indicates that the timestamp is extracted from the object module from the date form or the timestamp form of `#pragma comment`, whichever comes first.

(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.

3 User Comments

Displays the user form of comments that `#pragma comment` generated. These comments are extracted from the `END` record. You can add such

comments on multiple END records and have them displayed in the listing. See the *C/C++ Language Reference* for more information on the END record.

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 additional information on each symbol. That is

'I' indicates that the name is compiled IPA(NOOBJECT).

'L' indicates that the name is a long name. An 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 is compiled XPLINK.

Note: WL indicates that the symbol is both a long name and in writable static.

Chapter 14. DLL Rename Utility

This chapter describes the DLL Rename utility, which is part of z/OS Language Environment. You can use the DLL Rename utility to package and redistribute DLLs with your application.

As of OS/390 Version 1 Release 3, the C/C++ IBM Open Class Library component is licensed with the z/OS base and can be used without enablement of the C/C++ features. If your application uses C++ Class Library DLLs for execution on OS/390 Version 1 Release 3 or a later release, you are not required to rename the IBM-supplied DLLs that are shipped with your application.

If your application uses the C++ Class Library DLLs for execution on a system prior to OS/390 Version 1 Release 3, you MUST use the DLL Rename utility to rename the IBM-supplied DLLs, and ship the renamed DLLs with your application.

Note: The DLL Rename utility does not support `GOFF` (`XPLINK`) created DLLs.

With the DLL Rename utility, you can modify an executable application or a DLL to change the names of any DLLs that are loaded at execution time. The DLL Rename utility also provides a report which you can use to understand the DLL dependencies of your application.

Note: This utility does not change the names of variables or functions that are exported by the DLL or imported by your application.

You can use the DLL Rename utility under z/OS batch, TSO, and z/OS UNIX System Services. CICS and IMS do not support it.

Note: If you want to use the DLL Rename Utility, do not specify the Linkage editor option NE when you link-edit the DLL Rename Utility load module. This option removes the information the DLL Rename Utility requires to rename the DLL.

For information on building and using DLLs, see “Using DLLs” on page 494, and the *z/OS C/C++ Programming Guide*.

DLL Redistribution Scenario

Here is an example of a DLL redistribution situation. This example only applies if the application is intended for a release of OS/390 prior to OS/390 Version 1 Release 3.

Your C++ application is targetted to run on OS/390 Version 1 Release 2 C++, and references the `iostream` class library. You do not want your customers to license the C/C++ feature of OS/390 in order to access `IOSTREAM DLL` through the C++ Class Library DLLs. The following steps outline the process for renaming the IBM-supplied `IOSTREAM DLL` so that you can repackage it with your product. For details on the exact steps see “Using the DLL Rename Utility under z/OS Batch” on page 442 or “Using the DLL Rename Utility under TSO” on page 443.

1. Copy the DLL member `IOSTREAM` from the IBM-supplied library to your product library by using the `IEBCOPY` utility with the `COPYMOD` command. You should retain the original version of `IOSTREAM DLL`, especially if other applications use it.

2. Run the DLL Rename utility and rename references to IOSTREAM to a new name, PAHZIOST, so that your program will reference the new name. The name PAHZIOST is an example, you can use any valid PDS member name or a member in a PDSE built using OS/390 Version 2 Release 4 of the Binder.

```
//* Assumption is that PAH are the
characters used for your product
//QUERY  EXEC PGM=EDCDLLRN
//SYSIN  DD *
   'userid.product.load(PAHPGM1)'
   'userid.product.load(IOSTREAM)'   IOSTREAM=PAHZIOST
/*
```

   When the DLL Rename utility has completed, you will notice that member IOSTREAM has been renamed to PAHZIOST, and all references to IOSTREAM in your application are changed to PAHZIOST. You can verify this by running the DLL rename utility again without any rename cards.

   Note: If you want to be able to rebuild your application in the future, and you are required to rename IBM-supplied DLLs, you should copy the IOSTREAM definition side-deck into a private library, and change the name of the DLL on the IMPORT cards from IOSTREAM to PAHZIOST. You can then rebuild your application and bind it with the new definition side-deck. Otherwise, you will have to run the DLL Rename utility each time you rebuild.

3. You should also copy the members ICLBMSGT, CLB3MSGE, and CLB3MSGK from the PDS that contains the IBM-supplied class library DLLs. These members are used to display error messages in the event of failures in the class library code. You should rename these members to new names starting with PAHZ with an alias to the old name. For example:
   - ICLBMSGT - determines which error message member should be loaded based on the language level (LANGLVL) run-time option. Rename it to PAHZMSGT with an alias to ICLBMSGT
   - ICLBMSGE - English error messages, rename it to PAHZMSGE with an alias to ICLBMSGE
   - ICLBMSGK - Kanji error messages, rename it to PAHZMSGK with an alias to ICLBMSGK
4. Ship your product, renamed class library DLLs, and error messages load modules to your customers.

Inputs and Outputs

Input to the DLL Rename utility is a set of one or more programs or DLLs in either of the following:

- Any PDS
- A PDSE compiled with z/OS C/C++ compiler and bound with the binder.

Note: The DLL Rename utility does not support PDSE members built using the z/OS Language Environment Prelinker.

The modules can be applications that call DLLs or DLL modules themselves. Your applications and all the DLLs referenced (either by the application or by another DLL) can be all modified in a single step. Specify a DLL if it may call other DLLs. If the DLL does not reference any DLLs being renamed and is not itself being renamed, no modifications are performed.

Note: If the DLL being renamed is also specified as an input module, the DLL Rename utility attempts to rename the module itself. Copy the DLL first if it may be used by other applications using the old name.

Restriction

The input and output load-library modules must be PDS or PDSE members with the following attributes:

`RECFM=U, 256<=BLKSIZE<=32760`

If you run this utility under z/OS batch, input comes from the `SYSIN` data set. All the specified data gets passed to the DLL Rename utility including any sequence numbers resulting from using the ISPF editor. You must ensure that there are no sequence numbers in columns 73 to 80 in the data passed to the DLL Rename utility, otherwise it will fail.

If you run this utility interactively with TSO, the output goes to your terminal. If you use z/OS batch or TSO batch, output goes to the first of these data sets for which there is a definition:

- `DD:SYSPRINT`
- `DD:SYSTERM`
- `DD:SYSERR`

If you have not defined any of these data sets, output goes to `SYSOUT = *`.

When you specify rename statements, the old DLL name and the new DLL name must be different. The DLL names must be 8 characters or less in length. You must ensure that the name is a valid name for a load library.

The DLL Rename utility generates a report. For each program or DLL, it shows a list of DLLs that may be loaded or referenced. This information may help you understand the DLL dependencies of your application. The report contains the following:

- The fully qualified name of the input
- A list of DLLs that the module imports
- Any renaming of those DLLs that was performed

The following are examples of the DLL Rename utility reports:

```
DLLRNAME Report:                        1997/06/20  15:20:00

USERID.PROJECT.LOAD(PAHZAPP1)
  The following is a list of DLLs that are imported:

     IOSTREAM
```

Figure 51. Example of Output from DLLRNAME Utility - Query Only

Note: For any input module that does not reference a DLL, the report lists the module name, but does not list any information about it.

The DLL Rename utility also provides a report when it successfully renames any DLLs.

```
DLLRNAME Report:                        1997/06/20  15:45:00

USERID.PROJECT.LOAD(PAHZAPP1)
  The following is a list of DLLs that are imported:

     PAHZIOST which was renamed from IOSTREAM
```

Figure 52. Example of Output from DLLRNAME Utility

Using the DLL Rename Utility under z/OS Batch

The following is an example of JCL that you can use to rename a DLL under z/OS batch.:

```
//RENAME EXEC EDCDLLRN,GOPARM='LEopts / options'
//SYSIN DD *
     modname1 modname2
     modname3 modname4   oldname=newname
/*
```

where:

`EDCDLLRN` is an IBM-supplied procedure that runs the DLL rename utility. The procedure is shipped in the data set `CEE.SCEEPROC`.

`LEopts` refers to z/OS Language Environment run-time options. If you want to receive messages in Kanji, use the `NATLANG` option. For detailed information about z/OS Language Environment run-time options, see the *z/OS Language Environment Programming Reference*.

options you can enter valid options in uppercase or lowercase. The following are valid options:

- `NOREPORT` Does not generate output, unless you are performing a query.
- `FORCE` If the *newname* specified for a DLL is the same as an existing DLL member name, the renamed DLL erases and replaces the existing DLL.

modname is an existing program or DLL. You can enter more than one value for modname. The modules can be fully qualified or can assume a high-level qualifier of the current user prefix.

oldname is the member name of the existing DLL being referenced.

newname is the new DLL member name.

The following list shows the order for determining the default output data set name:

- `DD:SYSPRINT`, if defined
- `DD:SYSTERM`, if defined
- `DD:SYSERR`, if defined
- `SYSOUT=*` appended to the JOB log.

You can use an input file instead of specifying it in instream JCL. The input file must contain the module names for each application or DLL and the corresponding `oldname=newname` strings. The input file must also be assigned to `DD SYSIN`.

Notes:

1. If rename statements `oldname=newname` are not specified in the input, DLL Rename utility queries the input `modname`s and lists the DLLs that they load.
2. If you are renaming a DLL that is shared by many applications, you should copy the DLL (by using the COPYMOD command of IEBCOPY) to preserve the old DLL and create the new DLL.
3. For the `SYSIN DD *` statement, ensure that there are no sequence numbers in columns 73 through to 80 that is passed as input to DLL Rename utility, otherwise it will fail.

Example of Renaming a DLL under z/OS Batch

To rename the DLL described in “DLL Redistribution Scenario” on page 439,

1. Query your application to find all imported DLLs

```
//QUERY   EXEC PGM=EDCDLLRN
//SYSIN   DD   *
  'userid.product.LOAD(PAHPGM1)'
/*
```

2. Run the DLL Rename utility to rename the class library DLL

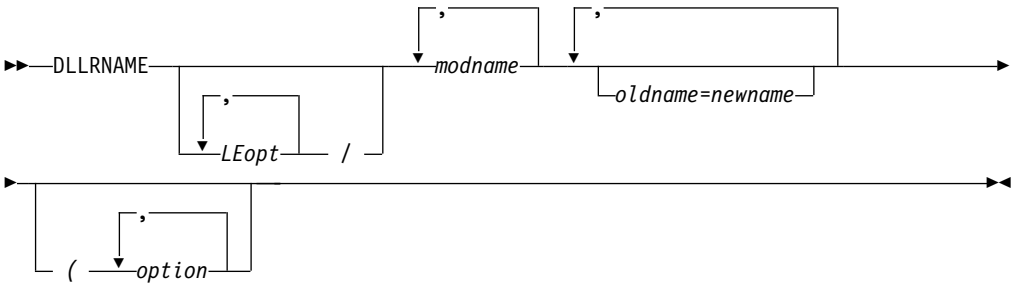
```
//RENAME  EXEC PGM=EDCDLLRN
//SYSIN   DD   *
 'userid.product.load(PAHPGM1)' IOSTREAM=PAHZIOST
 /*
```

3. Ship `PAHPGM1` to your customers with the `PAHZIOST` DLL.

Using the DLL Rename Utility under TSO

The following is the syntax diagram for specifying all parameters directly with the DLL Rename utility

Specifying DLLRNAME Parameters Directly



where:

`LEopt` refers to z/OS Language Environment run-time options. If you want to receive messages in Kanji, use the `NATLANG` option. For detailed information about z/OS Language Environment run-time options, see the *z/OS Language Environment Programming Reference*.

modname is an existing program or DLL. You can enter more than one value for modname. The modules can be fully qualified or can assume a high-level qualifier of the current user prefix.

oldname is the member name of the existing DLL being referenced.

newname is the new DLL member name.

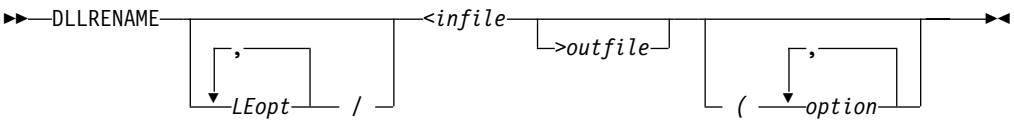
options you can enter valid options in uppercase or lowercase. The following are valid options:

`NOREPORT` Does not generate output, unless you are performing a query.

`FORCE` If the *newname* specified for a DLL is the same as an existing DLL member name, the renamed DLL erases and replaces the existing DLL.

Specifying DLLRNAME Parameters Using an Input File

The following is the syntax diagram for using an input file to provide parameters to the DLL Rename utility under TSO.



where:

`LEopt` refers to z/OS Language Environment run-time options. If you want to receive messages in Kanji, use the `NATLANG` option. For detailed information about z/OS Language Environment run-time options, see the *z/OS Language Environment Programming Reference*.

infile the name of the file that supplies input parameters to the DLL Rename utility. It contains the module name for each application or DLL and the corresponding `oldname=newname` strings.

If you do not specify an input file, the default is `SYSIN`.

outfile the file name for the output from the DLL Rename utility. The following list shows the order for determining the default outfile under TSO batch:

- `DD:SYSPRINT`, if defined
- `DD:SYSTERM`, if defined
- `DD:SYSERR`, if defined
- `SYSOUT=*` appended to the JOB log

Under TSO interactive, the default outfile destination is the terminal.

option You can enter valid options in uppercase or lowercase. These are the valid options:

`NOREPORT` Does not generate output, unless you are performing a query.

`FORCE` If the *newname* specified for a DLL is the same as an existing DLL member name, the renamed DLL erases and replaces the existing DLL.

Note: If there are no `oldname=newname` strings in the input, DLLRNAME queries the input `modnames` and lists the DLLs that they load.

The z/OS Language Environment run-time load library and the load library that contains DLLRNAME must be allocated to the `STEPLIB` DD name. This data set is called `CEE.SCEERUN`.

If you have not allocated the output load-library module, the data set is allocated with the attributes of the input load-library module.

Example of Renaming a DLL under TSO

To rename the DLL described in “DLL Redistribution Scenario” on page 439, run the Utility:

```
DLLRNAME 'userid.product.load(PAHPGM1)' IOSTREAM=PAHZIOST
```

Note: If you receive an error, run DLLRNAME as a query (without any *oldname=newname* parameters) to see if any DLLs were renamed.

Chapter 15. Filter Utility

This chapter describes how to use the CXXFILT utility to convert mangled names to demangled names.

When z/OS C++ compiles a program, it has the ability to encode function names. It also has the ability to encode other identifiers to include type and scoping information. This encoding process is called *mangling*. Mangled names ensure type-safe linking.

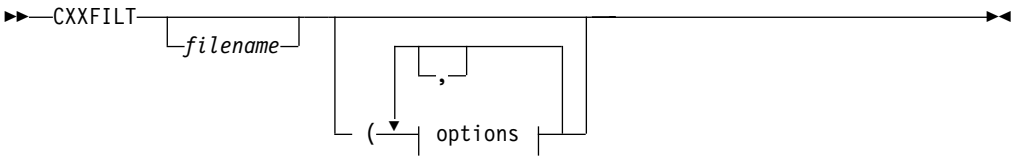
Use the CXXFILT utility to convert these mangled names to demangled names. The utility copies the characters from either a given file or from standard input, to standard output. It replaces all mangled names with their corresponding demangled names.

The CXXFILT utility demangles any of the following classes of mangled names when the appropriate options are specified.

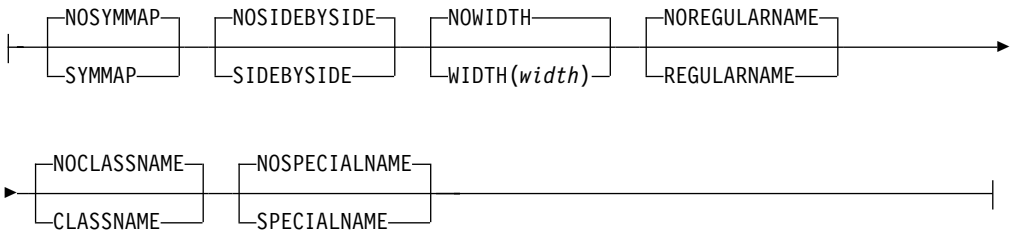
regular names
Names that appear within the context of a function name or a member variable. For example, the mangled name `__ls__7ostreamFPCc` is demangled as `ostream::operator<<(const char*)`.

class names
Includes stand-alone class names that do not appear within the context of a function name or a member variable. For example, the stand-alone class name `Q2_1X1Y` is demangled as `X::Y`.

special names
Special compiler-generated class objects. 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 specify no file name, CXXFILT reads from `stdin`.

The following section 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.

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 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

Demangles special names, such as compiler-generated symbol names, for example __vft1X.

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.

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 FILE1 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.

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 `/*`.

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 FILE1 contains the following two mangled names:

```
pr__3FOOFv
__vft1X
```

For example, enter the following command:

```
cxxfilt FILE1 (SPECIALNAME WIDTH(35) SIDEBYSIDE
```

The above command produces the following output:

```
FOO::pr()                          pr__3FOOFv
X::virtual-fn-table-ptr            __vft1X
```

CXXFILT terminates when it reads the end-of-file.

Chapter 16. DSECT Conversion Utility

This chapter 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 structure according to the options specified.

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 47. 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
UNNAMED | 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

SECT

DEFAULT: SECT(ALL)

The SECT option specifies the section names for which structures are to 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 the above 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 the fields above 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 the user 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.

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.

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.

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.

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.

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.

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 options BITF0XL, COMMENT, DEFSUB, EQUATE, HDRSKIP, INDENT, LOWERCASE, PPCOND, and UNNAMED, 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
```

For 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).

UNNAMED | NOUNNAMED

DEFAULT: NOUNNAMED

The UNNAMED 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 to ensure 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  0F,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

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 53. 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.

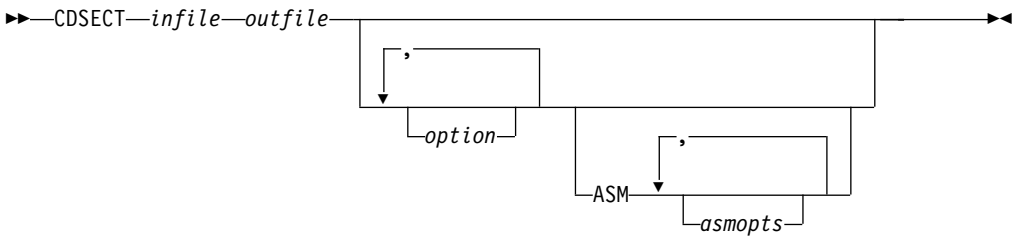
The parameters to the `EDCDSECT` procedure are:

Table 48. 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 *options* are any valid DSECT utility options. 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 54. 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 17. Coded Character Set and Locale Utilities

This chapter 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 prior to compiling:

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. The standard C library functions such as `iconv_open()`, `iconv()`, and `iconv_close()` are called from the iconv utility to perform coded character set translation. Any program that requires coded character set translation can call these functions. For more information on these functions, refer to the *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 C/C++, or to build code set conversions for existing code pages.

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 under these environments and 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 uses the `iconv_open()`, `iconv()`, and `iconv_close()` functions to perform the conversion requested. It 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. For more information on the `iconv()` function, refer to the *z/OS C/C++ Run-Time Library Reference*.

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

For example:

```
//ICONV    EXEC PROC=EDCICONV,
//         INFILE='FRED.INFILE',
//         OUTFILE='FRED.OUTFILE',
//         FROMC='IBM-037',
//         TOC='IBM-1047'
```

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, refer to the *z/OS 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:

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.

For example,

```
ICONV INPUT.FILE OUTPUT.FILE FROMCODE(IBM-037) TOCODE(IBM-1047)
```

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, refer to the *z/OS C/C++ Programming Guide*.

Under the z/OS Shell

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 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, behavior depends on the `iconv()` function of the system. See *z/OS C/C++ Run-Time Library Reference* for more information about the character that is used for converting incorrect characters.

See *z/OS C/C++ Programming Guide* for a list of code pages that the z/OS shell supports.

You can use `iconv` to convert singlebyte data or doublebyte 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 version 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, refer to the *z/OS 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.

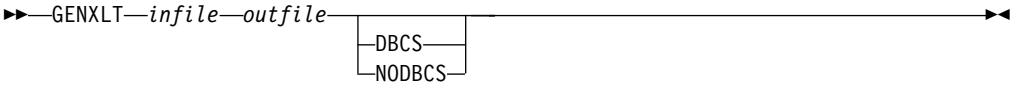
For example:

```
//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:



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

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 and XPLINK 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 HFS under the z/OS shell. The non-XPLINK version of the locale object should always be generated. If any XPLINK applications will use the locale then the XPLINK version should also be generated.

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. A user can create his own customized locale definition file.

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().

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 sections describe how you can invoke the localedef utility. For more information on locale source files, codeset definition files (CHARMAPs), and locale object names, refer to the *z/OS C/C++ Programming Guide*. For information on using the localedef utility under z/OS UNIX System Services, refer to the *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 C/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 XPLINK version should have CEH$ or CEH@ as the first four characters of the member name.

LOPT — The options for the localedef utility

For example:

```
//LOCALDEF    EXEC PROC=EDCLDEF,
//            INFILE='FRED.LOCALE.SOURCE(EDC$EUEY)',
//            OUTFILE='FRED.LOCALE.LOADLIB(EDC$EUEM)',
//            LOPT='CHARMAP(IBM-297)'
```

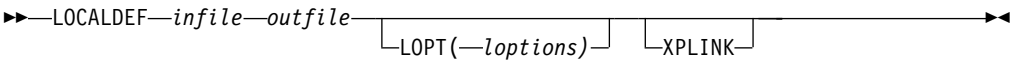
Under z/OS batch, you specify the options on the EXEC PARM and separate them by spaces or commas.

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 C/C++ compiler to compile the C source generated
3. Invokes the Linkage Editor to build the locale into a loadable module

The invocation syntax for the LOCALDEF REXX EXEC is as follows:



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.

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 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**] *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 localdef 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. 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: This option is only valid if the **–A** option is also specified, otherwise a severe error message is issued and the processing is 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 optimized locale object.

name Is the target locale. The HFS name for the non-XPLINK version of the locale can be arbitrarily assigned, but by convention the *name* is the same as the descriptive name of the locale. The HFS name for the XPLINK version of the locale is then formed by adding the suffix ″.xplink″ to the end of the non-XPLINK name. Locale descriptive names are described in the *z/OS C/C++ Programming Guide*. It is permitted to ignore these naming conventions, but you are then required to explicitly supply the full path name of the locale object on each setlocale() invocation. In any event, the non-XPLINK and XPLINK versions of the locale must have distinct names. The convention of .xplink at the end of the XPLINK locales satisfies this requirement. It is common for setlocale() to be given the descriptive locale name using environment variables. When the conventions are followed then the system can find both the non-XPLINK and XPLINK when needed and without having to change the environment variables to fully specify the HFS locale. See the *z/OS C/C++ Programming Guide* for more information about HFS resident locale object names.

z/OS ships two versions of the `localedef` utility:

- One is invocable under z/OS Batch and TSO, and is shipped with the z/OS C/C++ compiler.
- The other is invocable under z/OS UNIX System Services, and is shipped with z/OS UNIX System Services.

For more information, refer to *z/OS UNIX System Services Planning*.

The TSO REXX Exec `LOCALDEF`, included in the 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.

Part 5. z/OS UNIX Utilities

This part contains information about the z/OS UNIX System Services utilities.

Chapter 18. Archive and Make Utilities

This chapter 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 the *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 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 file.

Archive Libraries

The `ar` utility allows you to create and maintain a library of z/OS 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 for object file for data sets. For more information, see “Chapter 13. Object Library Utility” on page 425.

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. For 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 HFS working directory. The `-c` option tells `ar` to suppress the message that it normally sends when it creates an archive library file.

For control purposes, 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
```

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*, which you create, 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 normal HFS 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 C/C++ compiler, and the binder, to recompile and rebind an updated application program.

See the *z/OS UNIX System Services Programming Tools*, and the *z/OS UNIX System Services Command Reference* for a detailed discussion of the shell `make` utility and how to best take advantage of its function.

Makedepend Utility

The `makedepend` utility can also be used to create a makefile that can be used by make. 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 the *z/OS UNIX System Services Command Reference* for a detailed discussion of the `makedepend` utility.

Chapter 19. BPXBATCH Utility

This chapter provides a quick reference for the IBM-supplied BPXBATCH program. BPXBATCH makes it easy for you to run shell scripts and z/OS C/C++ executable files that reside in hierarchical file system (HFS) 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, a user who wants 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 the user's profile) can use BPXBATSL. BPXBATSL provides users with an alternate entry point into BPXBATCH, and forces a program to run using a local spawn instead of fork/exec as BPXBATCH does. This ultimately allows a program to run faster.

BPXBATSL is also useful when the user wants to perform a local spawn of their program, but also needs subsequent child processes to be fork/executed. Formerly, with BPXBATCH, this could not be done since BPXBATCH and the requested program shared the same environment variables. BPXBATSL is an alias of BPXBATCH.

For information on c89 commands, see "Appendix F. c89 — Compile, link-edit and assemble a z/OS C program and create an executable file" on page 577.

BPXBATCH Usage

The BPXBATCH program allows you to submit z/OS batch jobs that run shell commands, scripts, or z/OS C/C++ executable files in hierarchical file system (HFS) 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 HFS files for passing input, for shell command processing, and writing output and error messages. If you do allocate standard files, they must be HFS 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 to the program that is being invoked. This can be useful when not using the shell, such as when using the PGM parameter.

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. The total length of the parameter string must not exceed 100 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 C/C++ executable file that is located in an HFS 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 name or the HFS pathname for the shell script or z/OS 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 the user's *initial* working directory information to the program pathname.

Usage Notes

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 HFS pathnames, using only the POSIX portable character set. For information on the POSIX portable character set, see the *C/C++ Language Reference*.
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 other than 0, 1, or 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

- 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.

Part 6. Appendixes

Appendix A. Prelinking and Linking z/OS C/C++ Programs

Instead of using the prelinker and linkage editor, you can use the binder. See “Chapter 10. Binding z/OS C/C++ Programs” on page 365 for more information.

This chapter shows how to prelink and link your programs under z/OS with the z/OS Language Environment. The z/OS 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 z/OS 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 490. For information on creating object libraries in z/OS C++, refer to “Chapter 13. Object Library Utility” on page 425. 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 515.

Restrictions on Using the Prelinker

You cannot use the prelinker if you specified either the `XPLINK` or `GOFF` compiler option when you compiled.

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).

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

z/OS 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 longnames to resolve exported and imported symbols. For information on how to create a DLL or an application that uses DLLs, see the *z/OS 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 49. For details on the attributes of specific data sets see "Description of Data Sets Used" on page 555.

Table 49. 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 z/OS 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
SYSPRINT	Output	Destination of error messages generated by the prelinker
User-specified[1]	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 in-line 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 518.)

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 z/OS Language Environment Library (STEPLIB)
To prelink your program the system must be able to locate the data sets that contain the prelinker and z/OS 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` must be in the `JOBLIB` or `STEPLIB`. For information on the search order, see “Chapter 12. Running a C or C++ Application” on page 413.

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 page 310.

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.

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 z/OS 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 z/OS 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 z/OS 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 520.
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 the above 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 the *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 HFS 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 50 shows the five data set names and their characteristics.

Table 50. 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 in-line 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 518.)

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 the `DECK` or `OBJECT` compiler options)
- 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
- z/OS 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 C/C++, refer to “Description of Data Sets Used” on page 555.

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 that is described above.

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 DFSMS Program Management*.

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 51.

Table 51. 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
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 DFSMS Program Management* .

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 the *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 C/C++ programs under z/OS Language Environment, refer to the *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 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 z/OS Language Environment and must be specified as one of the libraries to be searched during autocall.

Link-Editing Multiple Object Modules

z/OS 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 as described above.

Building DLLs

Note: This section 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 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.

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 the above 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.

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 the above 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 15. Filter Utility” on page 447.

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.

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.

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.

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 */
```

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 */
}
```

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;
}
```

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 55. 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
//SYSOUT   DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
```

Figure 55. 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.

Prelinking and Linking an Application Under z/OS Batch and TSO

Figure 56 shows the basic prelinking and linking process for your C or C++ application.

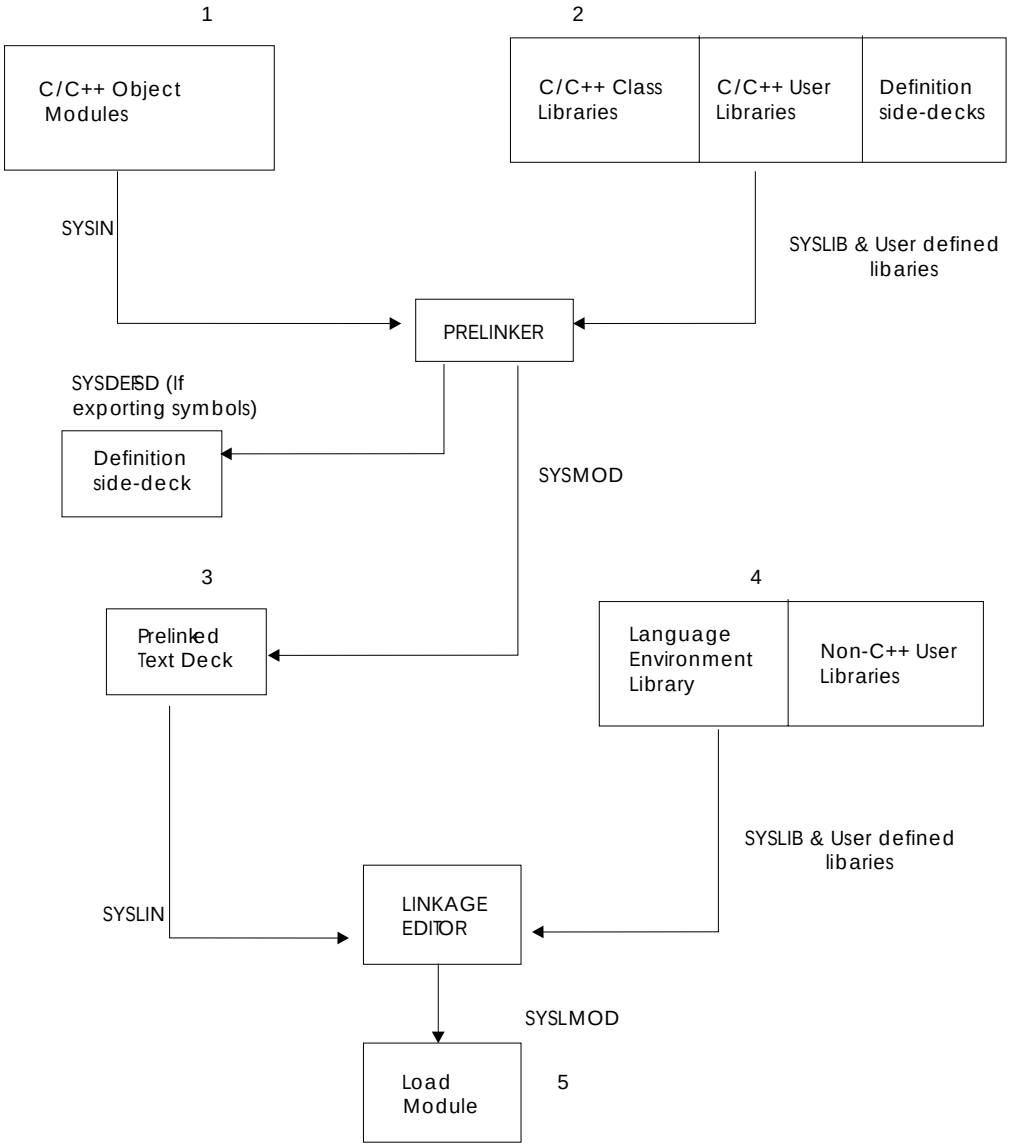


Figure 56. Basic Prelinker and Linkage Editor Processing

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 z/OS 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.

z/OS Language Environment Prelinker Map

When you use the MAP prelinker option, the z/OS Language Environment Prelinker produces a Prelinker Map. The listing contains several individual sections that are only generated if they are applicable.

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 58 on page 500, the definition side-deck USERID.DLL.IMPORTS(EXPONLY) contains the record IMPORT DATA 'EXPONLY' year.

The map that is shown in Figure 59 on page 500 was created by compiling the program that is shown in Figure 57. Figure 59 on page 500 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 57. z/OS 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 58. Example of JCL Used to Generate the Example Prelinker Map for a C++ program.

```
================================================================================
|                          Prelinker Map    1                                  |
|                                                                              |
| CPLINK:5647A01 V2 R10 M0 IBM Language Environment 2000/05/17 15:45:56|
 ==============================================================================

 Command Options. . . . . : NONCAL   NOMEMORY ER       DUP      MAP
                          : NOOMVS   NOUPCASE


 ==============================================================================
|                    Object Resolution Warnings    2                         |
 ==============================================================================

 WARNING EDC4015: Unresolved references are detected:
 CEESTART CEESG003 @@TRGLOR
```

Figure 59. Prelinker Map (Part 1 of 3)

```
================================================================================
|                              File Map    3                                   |
================================================================================

*ORIGIN  FILE ID  FILE NAME

   P      00001   DD:SYSIN
   A      00002   CEE210.SCEECPP(EDCHSG03)
   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 59. Prelinker Map (Part 2 of 3)

```
================================================================================
|                          Export Symbol Map    7                              |
================================================================================

*TYPE    FILE ID  NAME

   C      00001   get_year()
   C      00001   next_year()
   D      00001   this_int_is_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   NEXT@YEA   next_year()
   D      00001   @ST00003   Name_Collision_In_First8()
   D      00001   @ST00002   Name_Collision_In_First_Eight()
   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 59. 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 HFS file name and directory

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.

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 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 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 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 C/C++ Object Library Utility, see “Chapter 13. Object Library Utility” on page 425.

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 59 on page 500.

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 520 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 OS/390 V2R10 version of the static library is in CBC.SCLBCPP. The z/OS V1R2 version of the static library is in CBC.SCLBCPP2.

Note: Your application cannot use multiple copies of an IBM Open Class library. 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 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 “Appendix D. Cataloged Procedures and REXX EXECs” on page 551 for more information on REXX EXECs and their uses.

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 60. 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 “Appendix D. Cataloged Procedures and REXX EXECs” on page 551.

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 525, for linkage editor options see “Linkage Editor Options” on page 527.

Example of JCL to Prelink and Link

Figure 61 on page 507 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
//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 61. 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 “Appendix D. Cataloged Procedures and REXX EXECs” on page 551 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 62 on page 508.

```
   ⋮
//SYSLIN DD    DSNAME=&&GOFILE,DISP=(SHR,DELETE)
//       DD    *
   INCLUDE ddname(member)
   LIBRARY ADDLIB(CPGM10)
/*
```

Figure 62. Linkage Editor Control Statements

A 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 all the other input is processed.

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.

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 described above must be specified by linkage editor control statements in the primary input.

Under TSO

The z/OS 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 C Utilities” on page 561.

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.

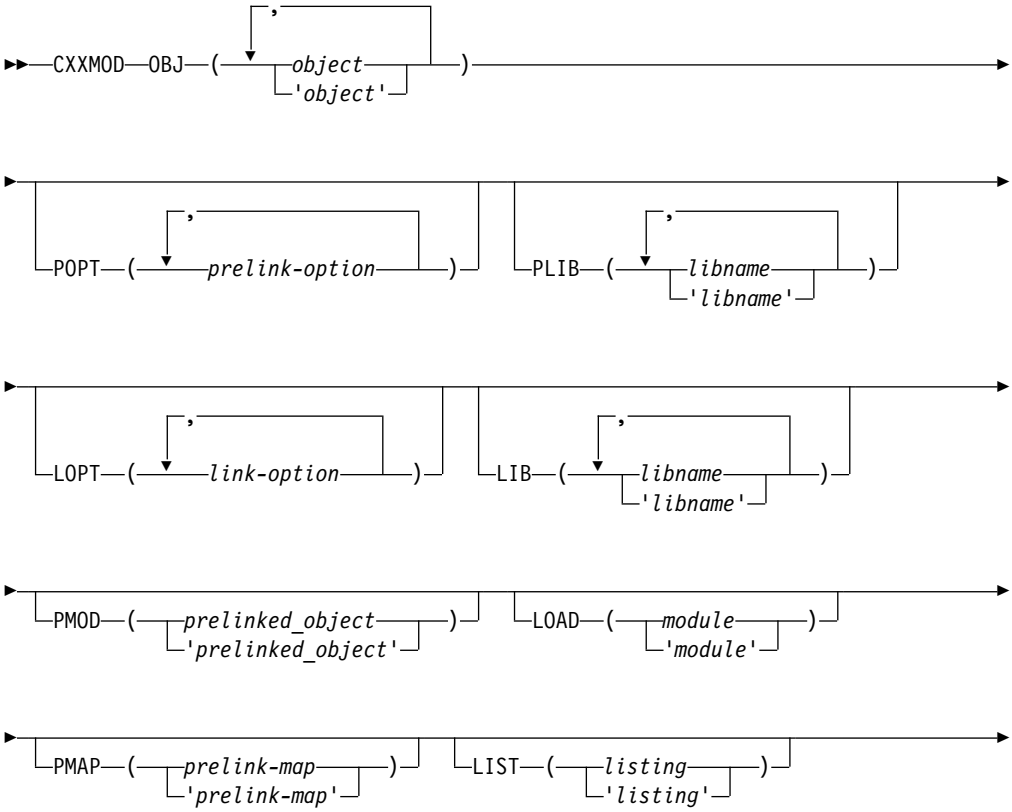
Any linkage editor processing beyond the basic processing described above must be specified by linkage editor control statements in the primary input. The IBM-supplied catalog procedures and REXX EXECs use the DLL versions of the IBM-supplied class libraries by default.

To link-edit your z/OS 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 C and z/OS 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 C Utilities” on page 561.

Prelinking and Linking under TSO

This section describes how to prelink and link your z/OS C++ or z/OS C program by invoking the CXXMOD REXX EXEC. This REXX EXEC creates an executable module.

The syntax for the CXXMOD REXX EXEC is:



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 an HFS 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.

For HFS file names: Neither commas nor special characters need to be escaped. But you must place file names containing special characters or commas between single quotes. If a single quote is part of the file name, the quote must be specified twice. HFS filenames 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 an HFS 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 an HFS 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 an HFS 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 an HFS 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 an HFS 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

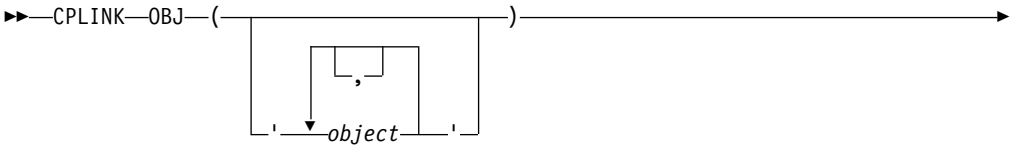
In the following example, the user prefix is `RYAN` and the input object module members `MAIN` and `FN` are in the PDS called `'RYAN.ACCOUNT.OBJ'`. A prelink map is to be generated and placed in `'RYAN.ACCOUNT.MAP(SALES)'`. The load module will be placed in a PDS member called `'GROUP.ACCOUNT.LOAD(SALES)'`. The linkage editor listing will be written to `'RYAN.ACCOUNT.LIST(SALES)'`.

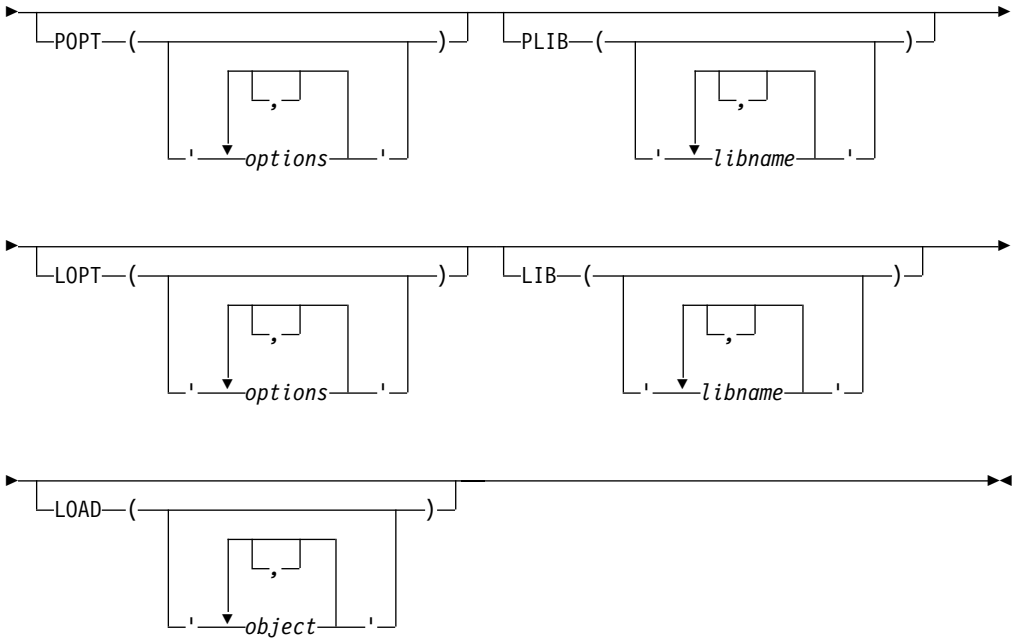
```
CXXMOD OBJ(ACCOUNT.OBJ(MAIN), ACCOUNT.OBJ(FN))
       POPT(MAP) LOPT(XREF, MAP)
       LOAD('GROUP.ACCOUNT.LOAD(SALES)') MAP(ACCOUNT.MAP(SALES))
       LIST(ACCOUNT.LIST(SALES))
```

In this instance, both the z/OS 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:





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 DFSMS Program Management*.

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) '
//*-------------------------------------------------------------
```

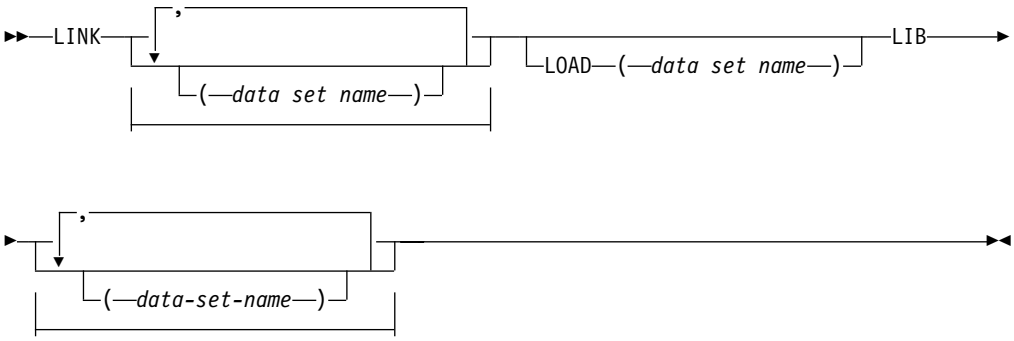
Figure 63. Example of Prelinking under z/OS Batch

```
CPLINK OBJ('''PAUL.C.OBJ(INCPYRL)''')
      POPT('NOMAP,NONCAL')
      PLIB('''RAINBOW.LIB.SUB''')
      LOAD('TBD.LOAD(MOD)')
```

Figure 64. 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 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.

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 492.

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 525.

Using your JCL

The example JCL in Figure 65 links to an archive library and to z/OS data sets. Include files may be PDS members, sequential files, or HFS files. Libraries may be partitioned data sets, or archive libraries.

```
//jobcard information...
//*-------------------------------------------------------------------
//*----- prelink -----------------------------------------------------
//RAWPLINK  EXEC PGM=EDCPRLK,
//          PARM='OMVS,MEMORY,MAP,NONCAL'
//STEPLIB   DD DISP=SHR,DSN=CEE.SCEERUN
//SYSMSGS   DD DISP=SHR,DSN=CEE.SCEEMSGP(EDCPMSGE)
//SYSLIB    DD DUMMY
//* object file
//DDOBJ1    DD PATH='/u/myuserid/callfoogoohoo.o'
//* PDS member
//DDOBJ2    DD DISP=SHR,DSN=MYUSERID.QAPARTNR.OBJ(MEM1)
//* archive library
//DDLIB3    DD PATH='/u/myuserid/mylibrary.a'
//* PDS Library
//DDLIB4    DD DISP=SHR,DSN=MYUSERID.QAPARTNR.OBJ
//SYSIN DD DATA,DLM=@@
 INCLUDE DDOBJ1
 INCLUDE DDOBJ2
 LIBRARY DDLIB3
 LIBRARY DDLIB4
@@
//SYSMOD    DD DISP=SHR,DSN=MYUSERID.TEMP.OBJ(MEM1)
//SYSOUT    DD SYSOUT=*
//SYSPRINT  DD SYSOUT=*
//SYSDEFSD  DD DUMMY
```

Figure 65. Using OMVS to prelink and link

The JCL in Figure 65 produces the following prelinker map:

```
================================================================================
|                            Prelinker Map                                     |
|                                                                              |
| CPLINK:5645001 V1 R7 M00 IBM Language Environment 1997/01/20 16:28:55|
================================================================================

Command Options. . . . . : NONCAL   MEMORY   ER       DUP      MAP
                         : OMVS     NOUPCASE


================================================================================
|                      Object Resolution Warnings                              |
================================================================================

WARNING EDC4015: Unresolved references are detected:
CEEBETBL CEEROOTA goo      CEESG003 EDCINPL


================================================================================
|                               File Map                                       |
================================================================================

*ORIGIN  FILE ID  FILE NAME

   PI     00001   /u/myusrd/callfoogoohoo.o
   PI     00002   MYUSRID.QAPARTNR.OBJ(MEM1)
   A      00003   /u/myusrd/mylibrary.a(foo.o)
   A      00004   MYUSRID.QAPARTNR.OBJ(MEMHOO)

*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

   P      00001   CEESTART   CEESTART
   P      00001   CEEMAIN    CEEMAIN
   D      00001   MAIN       main
   D      00003   FOO        foo
   D              GOO        goo
   D      00004   HOO        hoo
   P              CEESG003   CEESG003
   P              EDCINPL    EDCINPL
   D      00002   FUNC@IN@   func_in_MEM1

*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 66. 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 {_STEPS} environment for each of these utilities to use the Prelinker for link-edit output files that are PDSEs or HFS files.

Once you set the {_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 unset the {_STEPS} environment variable.

For a complete description of c89, c++, and cc, see “Appendix F. c89 — Compile, link-edit and assemble a z/OS C program and create an executable file” on page 577. For a description of the {_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 525.

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 nonblank 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 nonblank 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. For example, if you want the name

```
SymbolNameWithAQuote'InTheMiddle
```

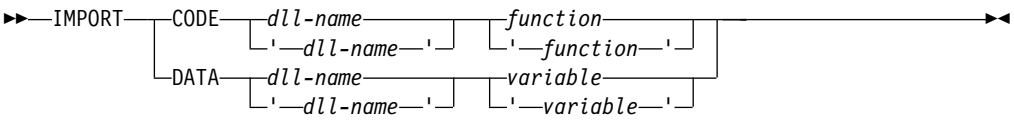
specify it as follows:

```
'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 quote in the name, all subsequent statements will continue in column two.

IMPORT Control Statement

The `IMPORT` control statement has the following syntax:



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 longname. The *dll-name* comes from the value specified on the `DLLNAME` prelinker option. For more information, see “Prelinker Options” on page 525.

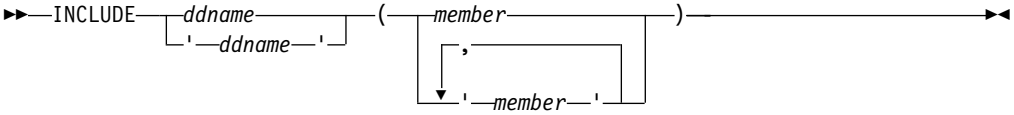
variable
An exported variable name. It is a mixed case longname. To indicate a continuation across statements, either use a non-blank character in column 72 of the card and begin the next line in column 16, or enclose the name in single quotation marks, end the first line in column 71, and put a blank character in column 1 of the next line.

function
An exported function name. It is a mixed case longname. 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:



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.

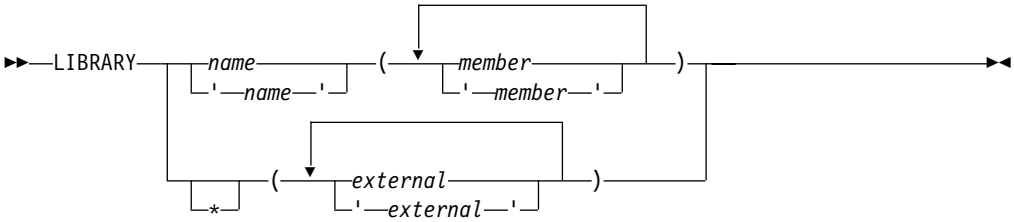
Note: The INCLUDE control statement is removed and not placed in the prelinker output object module; the system linkage editor does not see the INCLUDE control statement.

For more information on the linkage editor, refer to *z/OS DFSMS Program Management* .

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.

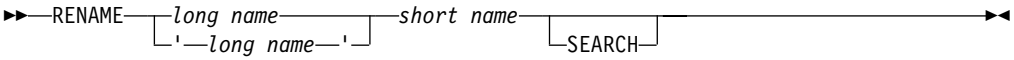
external
an external reference that may be unresolved after primary input processing. An Automatic Library call will not resolve this external reference. Because both short names and long names can be specified, case distinction is significant. If you use an long name, you can use the same kinds of continuations 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.

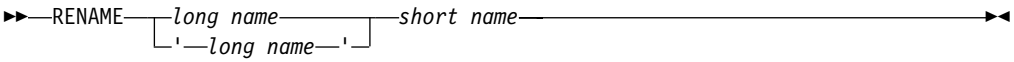
RENAME Control Statement

The RENAME control statement has the following syntax:

NOOMVS



OMVS



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 an 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 488 for a description of this rule.

- Explicitly bind an long name to an 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 488.
  - 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 section discusses how to use the prelinker to make your program reentrant. For detailed information on reentrancy, see the *z/OS 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 the writable static objects that are listed above 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 your C program contains writable static, you can use the prelinker to make your program reentrant. If your C program does not contain writable static, you do not need to use the prelinker to ensure reentrancy. The prelinker is called automatically for C++ programs.

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 UNIX System Services C/C++ application programs be reentrant. If you are using the `c89` utility, it automatically invokes the z/OS 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.

The prelinker can handle object modules from languages other than C or C++. However, only C or assembler code using the macros `EDCDXD` and `EDCLA` may refer to writable static objects.

Generating a Reentrant Load Module in C

To use the prelinker to generate a reentrant load module in C, you must follow these steps:

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 ensure that 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 z/OS 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 z/OS 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 the HFS. To do so you must install the program in the HFS, 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.

Generating a Reentrant Load Module in C++

To generate a reentrant load module in C++, you must follow these steps:

1. Compile your source code.
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 the *z/OS 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

See *z/OS C/C++ Programming Guide* for more information on external variables.

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 at run-time Language Environment 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 the *z/OS 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. The *C/C++ Language Reference* documents these functions. If you use the `XPLINK` compiler option, 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 C/C++ Run-Time Library Reference*.

Appendix B. Prelinker and Linkage Editor Options

This chapter contains the prelink options and link options for your programs under z/OS Language Environment. For more information on using the z/OS Language Environment Prelinker, see “Appendix A. Prelinking and Linking z/OS C/C++ Programs” on page 485.

Prelinker Options

The following section describes the prelink options available in z/OS C/C++ by using z/OS Language Environment.

DLLNAME(dll-name)

DLLNAME specifies the DLL name that appears on generated IMPORT control statements, described in “IMPORT Control Statement” on page 518. 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.

ER | NOER

DEFAULT: ER

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.

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 “z/OS Language Environment Prelinker Map” on page 499 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 HFS 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 processs 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 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 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 6. Binder Options and Control Statements” on page 287 or the *z/OS DFSMS Program Management* manuals.

Appendix C. Diagnosing Problems and the PMR/APAR Process

This appendix tells you how to diagnose failures in the z/OS C/C++ compiler. If you discover that the problem is a valid compiler problem, refer to “PMR/APAR Process” on page 536.

Problem Checklist

The following list contains suggestions to help you rule out some common sources of problems.

1. 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.
2. 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.
3. 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 C/C++ compiler, its utility components, or the z/OS C/C++ IPA Link step. Information on the messages is found in the *z/OS 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 C/C++-specific messages from the run-time environment. Information on Language Environment messages is found in the *z/OS Language Environment Run-Time Messages*.
   - CEE - for language-independent messages from the common execution environment (CEE) library component of z/OS Language Environment. Information on Language Environment messages is found in the *z/OS Language Environment Run-Time Messages*.
   - IBM, PLI, IGZ - for language-specific messages from z/OS Language Environment. Information on Language Environment messages is found in the *z/OS Language Environment Run-Time Messages*.
   - EQA - for Debug Tool messages.
   - CLB for messages that relate to OS/390 V2R10 class libraries and CLE for messages that relate to z/OS V1R2 class libraries. See the *z/OS C/C++ Messages* for more information.
   - BPX - messages that relate to z/OS UNIX System Services.

   You can cross reference the prefix to the message manual 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.
4. 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.

5. 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.
6. 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).
7. 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.
8. 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.
9. If a failing application is communicating with other IBM products, make sure that it uses the correct interface procedure as documented in the *z/OS C/C++ Programming Guide*. In many cases, you can localize the failing condition by taking out the function calls or making them no-ops.
10. 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 C/C++ compiler, the following may cause problems:
    - The source code does not support the applicable following standards:
      - *International Organization for Standardization (ISO) C Standard* (X3.159-1989)
      - ISO C++ 1998 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 IEE long double data types may have different sizes on a different plaform.
    - The source code is system dependent
11. If your application was prelinked, make sure that the prelinking was successful as indicated in “Appendix A. Prelinking and Linking z/OS C/C++ Programs” on page 485.

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 z/OS Language Environment, 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 DFSMS Program Management*.

After you identify the failure, you can write a small test case that re-creates 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 contains the error. Compile with the ″PPONLY″ option and send the expanded file as your source code. This is to ensure 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.

If your program uses ″#include″ directives, put all the relevant code from the ″#include″ file directly in the main file. Use the ″PPONLY″ option and then use the expanded file as your source for submission to your IBM representative.

Remove unrelated code if it is not necessary for syntactic or semantic validity.

Remove unreferenced variables. You can find them by using the ″XREF″, the ″CHECKOUT″(C only), or the ″INFO″ (C++ only) option.

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).

Remove all code and declarations from the body of any other functions.

If your program uses structure variables, try replacing them with scalar variables.

Complexity of Optimization

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.

1. Begin with a non-IPA compile and link using progressively higher levels of optimization:
   - `OPT(0)`
   - `OPT(2)` 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. Next, use `IPA(OBJONLY)` and `OPT(2)`.
   - This adds the IPA compile-time optimizations.
   - This 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. Finally, use the full IPA Compile and IPA(Level(2)) Link path
   - IPA Level 2 performs additional link-time optimizations

The Error Occurs 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 C++ does not allow missing prototypes.
2. Compile your program with either the `CHECKOUT` (C-only) or the `INFO` (C++ only) option. 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 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 non-standard code pages. See the *z/OS C/C++ Programming Guide* for more details.
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.

The Error Occurs at IPA Link Time

1. Ensure that the region that is used for the IPA Link step is sufficient. In a number of instances where `OPT(2)` has been used with IPA Link, more than 256 MB was required.
2. Ensure that the object module which defines main() contains 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.
8. 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.
9. 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.
10. 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.
11. 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 of OS/390 C/C++ or z/OS C/C++. 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.

The Error Occurs at Bind Time

For information on bind time errors, see “Error recovery” on page 407.

The Error Occurs at Prelink Time

1. Do not prelink the object modules separately.
2. Use the prelinker option `MAP` to obtain a full map of input data sets and symbols.
3. Use the prelinker options `DUP` and `ER` to obtain a full list of duplicate and unresolved symbols.
4. 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.
5. 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 494, or the *z/OS C/C++ Programming Guide*.
6. 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.
7. Only naturally reentrant code can be linked with the output of the prelinker. For more information, see the *z/OS C/C++ Programming Guide*.

The Error Occurs at Link Time

1. If you have a link-time error while working with the C/C++ component of z/OS Language Environment, you can find diagnostics and debugging information in *z/OS DFSMS Program Management* .
2. If you have a link time error while working with common execution environment (CEE) library component of z/OS Language Environment, you can find diagnostics and debugging information for link-time errors in *z/OS Language Environment Customization* and *z/OS Language Environment Debugging Guide*.

The Error Occurs at Run Time

1. If the problem occurs during execution, 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.
`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 (PP) only. 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`	To get information about the contents of variables at the point of the error, and to enable the use of the 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. *z/OS Language Environment Customization* and *z/OS Language Environment Debugging Guide* describe diagnostics and debugging information for run-time errors when executing with z/OS Language Environment.
5. 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 z/OS Language Environment. 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 C/C++ Compiler and Run-Time Migration Guide* documents these solutions.
6. 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.

Installation Problems

You can avoid or solve most installation problems if you follow these steps:

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 on page 7 on page 530.

If you still cannot solve the problem, develop a keyword string and contact your IBM Support Center.

You may need to reinstall the z/OS 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.

PMR/APAR Process

If you have already attempted to diagnose your problem using the process described in the previous sections of this appendix then this section describes your next step in the diagnosis of failures in the z/OS C/C++ compiler. (The z/OS C/C++ compiler may be referred to as the z/OS C/C++ program in the rest of this section.) The z/OS C/C++ uses z/OS Language Environment as a run-time environment, but to diagnose product failures that you encounter in that run-time environment, you should consult *z/OS Language Environment Customization*.

This file assumes that you have already determined that the suspected failure is not a user error; that is, it was not caused by incorrect use of the z/OS C or z/OS C++ compilers or by an error in the logic of the application program. If a user error, is the cause of the problem, and the Problem Checklist found at the beginning of this appendix does not address your case, consult the following books for more information:

- *Debug Tool User's Guide and Reference*
- *z/OS Language Environment Debugging Guide*

This process should help you to determine if a correction for a product failure similar to yours has been previously documented. If the problem has not been previously reported, "Preparing an Authorized Program Analysis Report (APAR)" on page 548 explains how to open a Problem Management Record (PMR) to report the problem to IBM, and if the problem is with an IBM product, how to prepare an Authorized Program Analysis Report (APAR).

If you are a customer who has an electronic link to one of the IBM databases, such as ServiceLink, you should have some knowledge of the databases before using them for searches. ServiceLink is the part of IBMLink that lets you access IBM service information online. Instead of calling the IBM support centre, you can use ServiceLink to search for service and support information, view product installation information and maintenance information, electronically report problems and receive answers, and monitor the status of APARs and Program Temporary Fixes (PTFs). Basically, running a search in ServiceLink consists of accessing Service Information Search (SIS), specifying what you want to search for and where, and then selecting the items to display, print or order from the lists of items found during your search. SIS provides access to a wide variety of service and support information about IBM products.

Isolating Reportable Problems

Failures in the z/OS C/C++ compiler can be described through the use of keywords. A keyword is a descriptive word or abbreviation assigned to describe one aspect of a product failure. A set of keywords, called a keyword string, describes the failure in detail. The procedures in this section will help you construct a keyword string that describes what you currently know about the compiler failure. This section is designed to help you specifically with failures in the z/OS C/C++ compiler, but it also contains generic procedures to follow when reporting any problem to IBM. For more information about failures that occur in the z/OS Language Environment run-time product, or for user run-time errors, see *z/OS Language Environment Debugging Guide*.

After you construct the keyword string, you can use it as a search argument against an IBM software support database, such as the Service Information Search (SIS). The database contains keyword and text information describing all current problems reported through Authorized Program Analysis Reports (APARs) and associated Program Temporary Fixes (PTFs). IBM Support Center personnel have access to the software support database and are responsible for storing and retrieving the information. Using the keyword string, they will search the database to retrieve records that describe similar known problems.

If you have IBMLink or some other connection to the IBM databases, you can do your own search for previously recorded product failures before calling the IBM Support Center.

If the keyword string matches an entry in the software support database, the search may yield a fuller description of the problem and possibly identify a correction or circumvention. Such a search may yield several matches to previously reported problems. Review each error description carefully to determine if the problem description in the database matches the failure.

If a match is not found, use the keyword string you have constructed to describe the failure when you contact the IBM support center for assistance and when you submit an APAR. Keywords are intended to ensure that identical program errors will be described with identical keyword strings. Spelling the keywords exactly as they are presented in this file is especially important for a successful match.

After you have stepped through each of the items in the Problem Checklist, to see if they apply to your problem, you can then develop a keyword string to use when you search the SIS software support database. It describes options which, if specified when you search the database, will supply you with all available diagnostic information. You will need this information to discuss the problem with your IBM support representative if your search fails to locate a fix for your problem.

If a problem occurs while you are using the z/OS C/C++ compiler, its cause may not be obvious; it might be an error in your application or in the z/OS C/C++ compiler itself. After you identify the failure, you may want to write a small test case that re-creates the problem. It will help you to:

- Pinpoint the problem
- Determine whether the error is in the application program or in the z/OS C/C++ compile
- Choose keywords that best describe the error if it is in the z/OS C/C++ compiler

If the problem appears to be the result of an error in the application program, change your source code accordingly and then compile, prelink (if required), and link the new code. If the problem appears to be the result of an error in the z/OS C/C++ compiler, develop a set of keywords using the procedure in “Keyword Usage”. Depending on your particular situation, then you can either search the SIS or contact your Service Representative.

Use the following procedures to diagnose the problem. As you go through the procedures, always note the sequence of events that leads to the error.

Keyword Usage

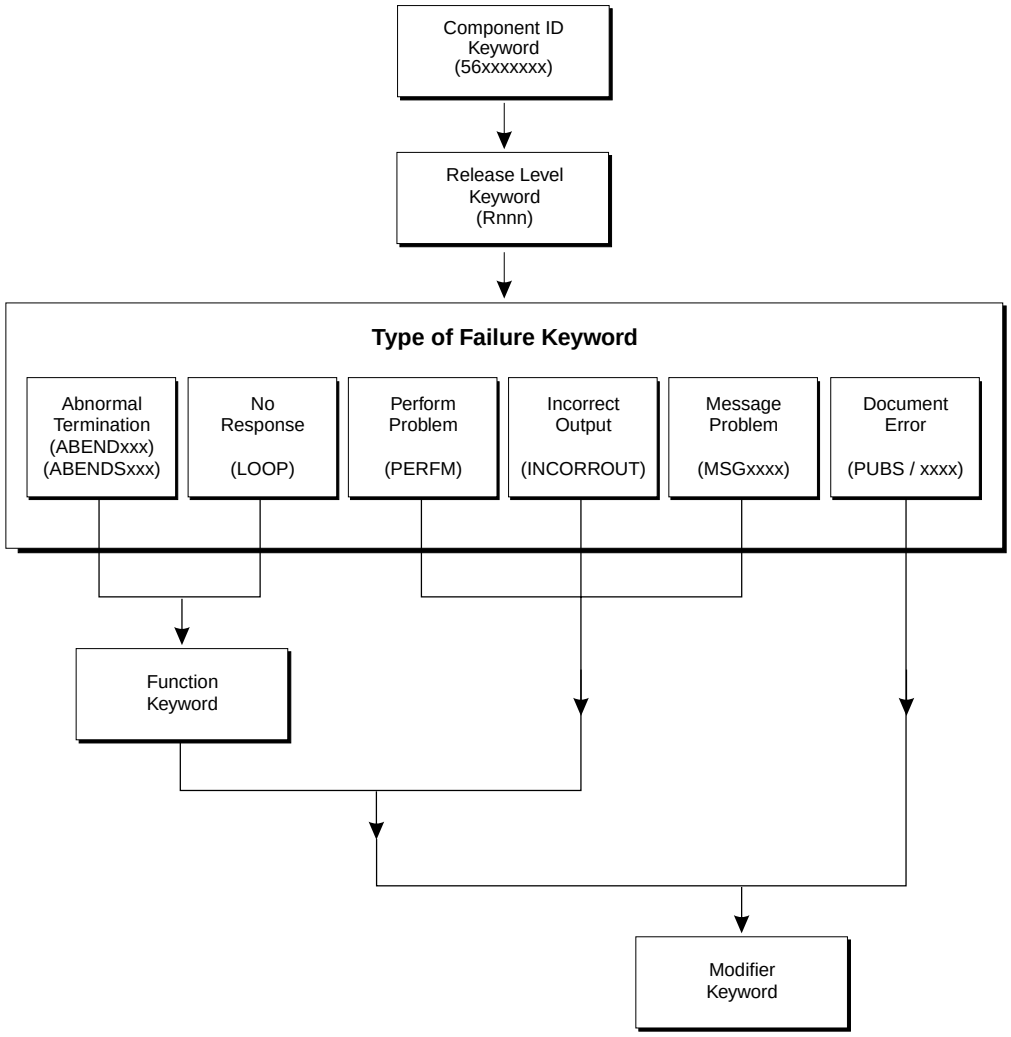
The first keyword of a keyword string is called the component identification, or ID. The component identification for the z/OS C/C++ compiler should be the component identifier, 56551210A. A search of the software support database with this single keyword would locate all problems reported for the compiler. A list of component identifiers that you may need appears in “Component Identification Keyword” on page 540.

Each additional keyword added to the keyword string narrows the scope of the search and helps to eliminate unnecessary examination of problem descriptions that have similar, but not matching, characteristics. In some cases, a correction for a product failure might be located with less than a full string of keywords. If it is unclear how to select a particular keyword to describe your problem, omit that keyword to avoid incorrectly identifying the problem.

A full set of keywords for the z/OS C/C++ compiler contains:

- The component identification
- The release level
- The type of failure
- The name of the module that failed, if applicable and available
- One or more modifier keywords, depending on the type of failure, if applicable

Follow the steps in the keyword procedures until you are directed to use the keyword string in a search argument. The following figure illustrates the process of creating a keyword string.



Using the Problem Identification Worksheet

You can use the ″Problem Identification Worksheet″ to help you construct and record a keyword string. As you identify the keywords associated with your software problem, record them in the spaces provided.

```
PROBLEM IDENTIFICATION WORKSHEET

  COMPONENT IDENTIFICATION:     _________________________

  RELEASE LEVEL:                ____

  TYPE OF FAILURE:              _________________________

  MODULE:                       __________

  SYSTEM TYPE:                  _________________________

  MODIFIERS:                    _________________________

                                _________________________

                                _________________________

  NOTE:  Some keywords may not be applicable to all problems.
```

Component Identification Keyword

This procedure shows you what to specify in the component identification keyword. It is always the first keyword placed in the search argument string. It comes from the z/OS C/C++ compiler program number and limits the search to the area within a software support database such as SIS, which contains items for the z/OS C/C++ compiler.

Component identifiers you may need are:

- 56551210A z/OS C/C++ compiler
- 56551210B IBM z/OS C/C++ class library
- 568819421 Debug Tool
- 5655A4501 z/OS C/C++ Performance Analyzer
- 568819801 Common Execution Environment (CEE) library component of z/OS Language Environment
- 568819805 z/OS C/C++ Language component of z/OS Language Environment
- 568819807 ANSI C/C++ component of z/OS Language Environment

The component identification keyword should be used with at least a type-of-failure keyword to search the software support database. If it is used without additional keywords, a full listing of all items affecting the z/OS C/C++ compiler will be produced.

1. Use 56551210A as the component identification keyword for the z/OS C/C++ compiler. (The component identification for the z/OS C/C++ language component of z/OS Language Environment is 568819805. Diagnostics for z/OS Language Environment are explained in *z/OS Language Environment Customization* and *z/OS Language Environment Debugging Guide*.)
2. If service tapes have been applied to the licensed program, note the tape level of the last service tape applied. See your system programmer for the current service level of your z/OS C/C++ compiler and z/OS Language Environment. Although the service tape level is not used as a keyword search argument, it will be useful when reviewing APARs selected during the keyword search.
3. Continue the diagnostic procedure with “Release Level Keyword” on page 541.

Release Level Keyword

The release level keyword identifies the specific release level of the z/OS C/C++ compiler, z/OS Language Environment, and the operating system you were using when the failure occurred.

1. If you do not know the release level code for your product level, you can find it in the appropriate program directory. The release level keyword is the release level code for your system prefixed with an R. If the release code for your product is 705, the release level keyword is R705. Some release level keywords you may use for component ID (COMPID) 56551210A:

   R705 z/OS V1R2 C compiler and utilities

   R715 z/OS V1R2 C++ compiler and utilities

   R725 z/OS V1R2 Kanji compiler messages

   R703 OS/390 V2R10 C compiler and utilities

   R713 OS/390 V2R10 C++ compiler and utilities

   R723 OS/390 V2R10 Kanji compiler messages

   R705 Standard class libraries

   R715 Collection class and application support class libraries

   R725 Kanji class library messages

   R735 HFS class library

   Some release level keywords you may use for COMPID 56551210B are:

   R705 Standard class libraries

   R715 Collection class and application support class libraries

   R725 Kanji class library messages

   R735 HFS class library
2. Continue the diagnostic procedure with “Type-of-Failure Keyword”.

Type-of-Failure Keyword

A specific failure may occur in the z/OS C/C++ licensed program. For product failures that occur in the z/OS Language Environment run-time library, consult *z/OS Language Environment Customization* or *z/OS Language Environment Debugging Guide*.

Read the following table and select the type of failure that best describes your problem. Refer to the associated keyword procedure for instructions on how to complete the keywords for that type of failure. If more than one keyword describes your problem, use the one that appears first in the table.

Type of Failure	Symptom	Procedure
Abnormal Termination	The compiler or program terminated abnormally with a completion code that indicates that an abend has occurred.	See “Abnormal Termination” on page 542
Message Problems	The compiler or program issued a message that is inappropriate or not valid.	See “Message Problems” on page 543

Type of Failure	Symptom	Procedure
No Response from the Compiler	An unexpected compiler or program suspension occurred; no response has been received in interactive mode.	See “No Response Problems” on page 543
Documentation Problems	Information in one of the z/OS C/C++ publications is incorrect or missing.	See “Documentation Problems” on page 543
Output Problems	The output from the compiler or program is missing or not valid.	See “Output Problems” on page 544
Performance Problems	The performance of a z/OS C/C++ compilation or program is degraded.	See “Performance Problems” on page 545

Abnormal Termination

If the z/OS C/C++ compiler terminates abnormally you will get one or more of the following:

- A message from z/OS Language Environment
- A traceback
- A system abend, such as an 0C4

If you get a z/OS Language Environment message but no traceback after an abnormal compiler termination, use z/OS Language Environment Debugging Guide and Run-Time Messages to diagnose the problem.

If you get a message from the operating system indicating a system abend but no traceback, follow the procedure described in the following information as the ″ABENDxxx Procedure.″

If you get a traceback, use the procedure described in “Function Keyword” on page 545. Record any other messages you get. You will need the other messages in order to do your keyword search or to report the problem.

Note: Do not use the abnormal termination procedures if termination was forced because too much time was spent in a wait state or an endless loop. Under MVS, this is usually accompanied by a system abend code x22. For example, an abend code of 322 indicates time exceeded and an abend code of 722 indicates lines exceeded. In this situation, refer to the procedure for “No Response Problems” on page 543.

Use the ABENDxxx procedure if the z/OS C/C++ compiler terminates abnormally with a system abend code. The following instructions help identify the ″ABENDxxx″ keyword needed for your keyword string.

1. Replace the ″xxx″ of ″ABENDxxx″ with the system abend code. For example, if the failure occurred during z/OS C compilation with a system abend ″0C4″, you would specify ″ABEND0C4″ as your keyword. Your keyword string at this point would appear as follows:

   56551210A R705 ABEND0C4

The compiler phase is shown in the listing. Use the name of the current compiler phase as a modifier keyword. For example, if the compiler phase was ″CCN3P″, your keyword string would appear as follows:

56551210A R705 ABEND0C4 CCN3P

2. Continue with “Modifier Keywords” on page 547.

Message Problems

The z/OS C/C++ compiler issues messages prefixed with ″EDC″ or ″CCN″. Messages with other prefixes are issued by the operating systems, subsystems, access methods or the common execution environment (CEE) library component of z/OS Language Environment run-time environment. These are not z/OS C/C++ product problems. See the message listings in the appropriate component manuals.

Use the MSGx keyword in your keyword string for any one of the following conditions:

- A message is issued when it should not have been issued
- A message contains data that is not valid or data is missing.

To construct the MSGx keyword:

1. Replace the x of MSGx with the message identifier in the format _ MSGaaannnn. For example, if the compiler message received is ″CCN0049 30″, the MSGx keyword would be MSGCCN0049. The severity code, ″30″ in this case, is not included in the MSGx keyword. Your set of keywords, so far, would look like this: 56551210A R705 MSGCCN0049
2. Proceed with “Modifier Keywords” on page 547.

No Response Problems

Use the LOOP keyword procedure for any of the following conditions:

- The z/OS C/C++ compiler seems to be doing nothing or appears to be in a loop.
- A terminal response is not received in interactive mode after the compiler is invoked.
- The compiler does not reach completion in batch mode.
- The system abend code is 322.

If the problem looks like a ″WAIT″ state and the compiler is not waiting for input from the terminal or console, it is probably a system problem. In this case, follow your local procedures for resolution, otherwise try the following suggestions:

- If you are running in batch mode and the error is a system abend with abend code 322, indicating not enough time, increase the time allotment and re-compile. If the problem is still unresolved, your set of keywords would now look like this: 56551210A R705 LOOP
- Continue with “Function Keyword” on page 545.

Documentation Problems

Follow this PUBS keyword procedure when you notice a problem caused by incorrect or missing information in one of the z/OS C/C++ hardcopy or softcopy documents.

1. Review the updates to the documentation which are in the CCN.SCBCDOC(BOOKS) data set and on the web at `http://www.ibm.com/software/ad/c390/czos/czosdocs.html`

2. If the existing information is wrong, locate the page or pages in the document or the online panel where the problem occurs, and prepare a description of the error and the problem it caused. If it is missing, locate the place where you think it should appear. This information will be required for APAR preparation if no similar problem is found in the software support database.
3. Decide whether this documentation problem is severe enough to cause lost time for other users. If the problem is not severe, fill out the Readers' Comments Form (RCF) attached to the back of the publication in question. If the RCF is missing, send a note to the address shown on the edition notice for this book. Include the problem description you have developed, along with your name and return address, so that IBM can respond to your comments. If the problem is severe enough to cause lost time for other users, continue creating your keyword string to determine whether IBM has a record of the problem. If this is a new problem, a severity 3 or 4 documentation (PUBS) APAR will be created.
4. In the case of hardcopy or softcopy books, use the order number on the cover of the document along with the PUBS keyword as your type-of-failure keyword, but omit the hyphens. For example, if the number on the cover is SC09-4815-01 (the *C/C++ Language Reference*) then use SC09481501. Place a forward slash '/' between PUBS and the document number. Your set of keywords would now consist of: 56551210A R705 PUBS/SC09481501
5. To determine if this documentation problem has already been reported, see "Using the Keyword String as a Search Argument" on page 547. If, after searching the IBM software support database, you do not find a matching description, return here to continue.
6. Before discontinuing your database search, you may want to search again, using the wildcard character '*' to replace a single character in the search string, as in the following format: 56551210A R705 PUBS/SC094815** In case several levels of the document exist, you can use two asterisks appended to the document number to search for all problems reported for the document rather than only those for a specific release of the document.
7. Go to "Using the Keyword String as a Search Argument" on page 547.

Output Problems

Use this procedure when the output appears to be incorrect or missing, but the z/OS C/C++ compiler otherwise terminated normally. If the data or records were repeated endlessly, follow the steps under "No Response Problems" on page 543 instead of using the ″INCORROUT″ procedure to create your keyword string.

1. Use ″INCORROUT″ as your type-of-failure keyword.
2. Select a modifier keyword from the following table to describe the type of error in the output. For more information on the use of modifier keywords, see the section "Modifier Keywords" on page 547.

Modifier Keyword	Type of Incorrect Output
MISSING	Some expected output was missing
DUPLICATE	Some data or records were duplicated, but were not repeated endlessly
INVALID	The output that appeared was not as expected; that is, the output was bad or incorrect

3. Select another modifier keyword from the following table to describe the portion of the output where the error occurred.

Modifier Keyword	Portion of Output in Error
AGGREGATE (C only)	Structure maps
ATTRIBUTE (C++ only)	List of identifiers
SOURCE	Source listing
OBJECT	Machine-language object program
XREF	Cross-reference listing
OFFSET	Storage offset listing
INLRPT	Inline report
PPONLY	Preprocessor output (this goes to SYSUT10 DD)
LIST	Assembler language expansion of source listing
MESSAGE	Diagnostic messages
TERM	Progress and diagnostic messages on the SYSTERM data set
OFFSET	Offset listing
EXPMAC	Macro expansions

For example, if you think that the compiler has given an incorrect cross-reference listing, your keyword string so far would contain the following: 56551210A R705 INCORROUT INVALID XREF

4. Continue the diagnostic procedure with “Modifier Keywords” on page 547.

Performance Problems

Most performance problems can be related to system tuning and should be handled by system engineers and system programmers. Use this keyword procedure when the performance problem could not be corrected by system tuning and performance is significantly below explicitly stated expectations.

1. Use PERFM as your type-of-failure keyword. For example, your keyword string for performance problems of the z/OS C compiler might look like this: 56551210A R705 PERFM
2. Continue the diagnostic procedure with “Modifier Keywords” on page 547.

Function Keyword

For product failures that occur at z/OS Language Environment run-time, see *z/OS Language Environment Customization* and for user run-time problems, see *z/OS Language Environment Debugging Guide*.

Should the compiler terminate abnormally, the following procedure will help you determine if there are function or language element modifier keywords that you should use in your keyword string. Use this procedure to locate the point of compiler failure and the language element you were using when the failure occurred.

The z/OS C/C++ compiler is itself a z/OS C/C++ program, and runs under z/OS Language Environment. If an error occurs in the compiler, it is handled by z/OS Language Environment which produces a traceback with information about the compiler error. The output is directed to ″DD CEEDUMP″.

How to Find the Function Name in the Traceback

```
Status
Call
Exception
Call
Call
Call
52BA121C
CEE3DMP: Condition processing resulted in the Unhandled condition.

Information for enclave main

  Information for thread 8000000000000000

  Traceback:
    DSA Addr  Program Unit  PU Addr   PU Offset  Entry         E Addr    E
Offset   Statement  Status
    00D63018  CEEHDSP       00CD15B8  +00001FB0  CEEHDSP       00CD15B8
+00001FB0             Call
    00D1CA40               0250DCC8  +0000008C  Tokenizer     0250DCC8
+0000008C             Exception
    00D1C950               0239D8B8  +00000474  CCNP          0239D8B8
+00000474             Call
    00D1C8A0               029108A0  +0009940C  @@FECCNBC130  029108A0
+0009940C             Call
    00D1C7B8               02D73470  +000000E4  InvokeFetch   02D73470
+000000E4             Call
    00D1C1E0               02D784E0  +000002BA  main          02D784E0
+000002BA             Call
    00D1C0C8               02ABCE46  +000000B0  @@MNINV       02ABCE46
+000000B0             Call
    00D1C018  CEEBBEXT     00CBF4B0  +00000138  CEEBBEXT      00CBF4B0
+00000138             Call

  Condition Information for Active Routines
    Condition Information for  (DSA address 00D1CA40)
      CIB Address: 00D633C8
      Current Condition:
        CEE3204S The system detected a Protection exception.
      Location:
        Program Unit:  Entry: Tokenizer Statement:  Offset: +0000008C
      Machine State:
        ILC..... 0004    Interruption Code..... 0004
        PSW..... 03EC0400 8250DD58
        GPR0..... 00D1CC08  GPR1..... 00D1CB08  GPR2..... 00000000
        GPR3..... 00000000
```

⋮

1. Look in the text that follows the word ″Traceback:″ in the sample traceback above.
2. Find the row that contains the word ″Exception″ in the ″Status″ column.
3. Use the function name found in the ″Entry″ column in that row. (Enter it in SIS in capital letters).

If the compiler failed and you received a traceback like in the previous figure, you would use ″TOKENIZER″ as your function keyword. At this point your keyword string would look like this: 56551210A R705 ″TOKENIZER″

If the failure is peculiar to a compiler statement or library function, use the statement or function name as a modifier keyword. For example, if the source of failure was the z/OS C/C++ library function ″ctime()″, your keyword string would look like this: 56551210A R705 CTIME

See “Modifier Keywords” for information about using modifier keywords in your keyword search.

Modifier Keywords

You can use one or more modifier keywords in the same keyword string to define the compiler problem. They help to make the search argument more specific. Capitalize all of the letters of the modifier in the keyword string. The various types of modifier keywords include:

- z/OS C/C++ compiler
  - Language Elements
  - Listing Control Statements
  - Compiler options (Select from your compiler listing those compiler options that you consider significant to the type of failure. The option name itself is the keyword.)
  - Message IDs
- z/OS Language Environment Library
  - Library functions
  - Link-edit options
  - Run-time options

If the failure appears to be associated with any particular compiler option or options, such as SHOWINC, use those options as modifier keywords, as in this example: 56551210A R705 SHOWINC

Continue the diagnostic procedure with “Using the Keyword String as a Search Argument”.

Using the Keyword String as a Search Argument

Searches using the SIS in ServiceLink will be most successful if you follow these rules:

- Spell keywords the way they are spelled in this file. Any variation in spelling may result in an unsuccessful search.
- Include all the appropriate keywords in any discussion with IBM support personnel or in an APAR.

The following explains how to use the keyword string as a search argument against a software support database,such as SIS.

1. Search the software support database, using the full set of keywords you have developed. Here is an example: 56551210A R705 ABEND0C4 CCNEP FUNCTION
2. If the search produces a list of APARs, continue with step 3; otherwise, go to step 6.
3. When your search is complete, eliminate from the list of possible PTFs those that have already been applied to your system.
4. Compare each of the remaining closed APAR descriptions and PTF descriptions with the current failure symptoms.
5. If a match is found, apply the program temporary fix (PTF) to your system and exit from this procedure.
6. If the search did not produce a list of APARs, or you did not find an APAR description matching the current failure, broaden the search, using the following techniques:

a. Omit the release level keyword (for example, R705) from the search argument, thereby broadening the search to include similar failures on other release levels.
b. Drop one keyword from the right end of the search argument string.

   By dropping a keyword from the right, you broaden your search while maintaining the relevancy of your search argument string. Perform the search against the software support database, using your shortened search argument string. Repeat this step as necessary, dropping keywords from the right of the string until only the COMPID is left.

7. If a match is not found after you have used the preceding methods, go to "Preparing an Authorized Program Analysis Report (APAR)".

Preparing an Authorized Program Analysis Report (APAR)

A Problem Management Record (PMR) can be opened after you have done the following:

1. Eliminated user errors as a possible cause of the problem.
2. Followed the diagnostic procedures.
3. You or your local IBM Support Center has been unsuccessful with the keyword search.

If you have IBMLink or some other connection to IBM databases, you can open a PMR yourself. Otherwise, the IBM Support Center may open the PMR after consulting with you on the phone. You may be asked for any of the documentation in the list below. The PMR is used to document your problem and to record the work that the Support Center does on the problem. If IBM concludes that the problem described in the PMR is a problem with the z/OS C/C++ product, they will work with you to open an Authorized Program Analysis Report (APAR) so the problem can be fixed. After the APAR is opened and the fix is produced, the description of the problem and the fix will be in the software support database in SIS, accessible through ServiceLink.

Before you initiate an APAR:

1. Contact the IBM Support Center for assistance. You will have been in contact with the IBM Support Center while they were working on the PMR. Be prepared to supply the following information:
   - Customer number and security code
   - PMR number
   - Operating system
   - Operating system release level
   - Current z/OS C/C++ compiler maintenance level (PTF list and list of APAR fixes applied), which can be determined by specifying the PHASEID compiler option
   - The various keyword strings used to search the software support database
   - Processor number (model and serial)
2. From the following list, you may be asked to include the applicable z/OS C/C++ environmental information with your APAR:
   - Job control statements
   - Compiler listings, including:
     – Source listing
     – Object listing

  - Storage map
  - Traceback
  - Cross-reference listing

  Use LIST, MAP, SOURCE, XREF and any other options pertinent to the problem.
- A machine-readable copy of the program causing the problem, including all ″#include″ files required by the program
- A dump on tape if available
- The compiler on tape if requested by an IBM representative
- A hard copy of the job control language for unloading the submitted machine-readable tape
- Any other data that may help in re-creating the problem

In addition, a description of the application, the data set organization, and the operating instructions or console log may be helpful in reproducing the error. Any listings you supply must be from the z/OS C/C++ compilation version that failed.

The following table describes how to produce documentation required for submission with the APAR. Additional requirements are explained after the table. Many of these materials may already have been produced in their required format during the formulation of the keyword string. (See “Isolating Reportable Problems” on page 537.)

Item	Materials Required	How to Obtain Materials
1	Machine-readable source program	The source program must be supplied in machine-readable form, generated by an IBM-supplied system utility program. The source program should be reduced to the smallest, least complex form that still produces the error.
2	Compiler listings: Source listing Cross-reference listing Assembler—language expansion	SOURCE option XREF option LIST option
3	Compiler termination dump	SYSMDUMP DD statement (as directed by IBM support personnel)
4	Partition/region size/virtual storage size	JCL or system programmer
5	List of applied PTFs	System programmer
6	Operating instructions or console log	Application programmer
7	Job control statements with MSGLEVEL(1,1), TSO ALLOCATE statements	The JCL that was used to invoke and run the compiler should be supplied in machine-readable form.

Preparation of Material

When submitting material for an APAR to IBM, be sure that the media containing source programs, job stream data, or interactive environment information are carefully packed and clearly identified. Each magnetic tape submitted must have the following information attached and visible:

1. The PMR number assigned by IBM.

2. A list of data sets on the tape (such as source program, JCL or interactive environment information).
3. A description of how the tape was made, including the following information:
   a. A full listing of JCL or interactive environment information used to produce the machine-readable source. Include the block size, LRECL, and format of each file. If the file was unloaded from a partitioned data set, include the block size, LRECL and number of directory blocks in the original data set.
   b. Labeling information used for the volume and its data sets.
   c. The recording mode and density.
   d. The name of the utility program that created each data set.
   e. The record format and block size used for each data set.

Maintenance

IBM Software Manufacturing Solutions (ISMS) provides corrective and preventive service for product defects, as well as support for resolving program problems. This is done through Central Service, including the IBM Support Center. For details of how these facilities work and a list of all the products supported, refer to *Field Engineering Programming System General Information*, G229-2228.

IBM distributes, when necessary, service updates in response to problems found in IBM licensed programs. When problems are identified and solutions established, a PTF is created and made available.

Corrective Service — Program Temporary Fixes (PTFS)

A program temporary fix (PTF) is a correction for a known problem or problems in a particular product. When a problem is identified and its solution established, a PTF is made available on tape. You can order this tape from IBM. You can also order a cumulative tape containing all the PTFs for your site.

A PTF arrives as a System Modification Program (SMP/E) data set accompanied by all the materials necessary for its installation and use. If you receive the PTF on tape, you must copy the PTF from the tape into a data set.

You should create a backup copy of the current compiler before the PTF tape is applied.

Preventive Service — Extended Service Offering Tapes (ESOS)

Extended service offering tapes (ESOS) can be ordered from IBM Software Manufacturing Solutions (ISMS). These tapes contain the PTFs to problems in IBM licensed programs under your operating system. Note that each ESO tape contains only PTFs for one month. Therefore, to bring z/OS C/C++ up to the latest level, you must install all prior ESO tapes sequentially, starting with the oldest.

Installing Corrective or Preventive Service

For specific procedures, always refer to the installation materials provided with the service. For product-specific information, refer to the appropriate program directory for service maintenance. The *SMP/E User's Guide*, SC28-1302 will provide detailed procedures for corrective and preventative service.

Appendix D. Cataloged Procedures and REXX EXECs

This appendix describes the REXX EXECs (TSO) and cataloged procedures that the z/OS C/C++ compiler provides in conjunction with z/OS Language Environment, to call the various z/OS 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 z/OS Language Environment provides, 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 10. Binding z/OS C/C++ Programs” on page 365. For a description of `CXXMOD` see “Prelinking and Linking under TSO” on page 509. For a list of the old syntax REXX EXECs, see “Other z/OS C Utilities” on page 561.

	Name	**Task Description**
REXX EXECs for z/OS C and z/OS C++	C370LIB	Maintain an object library under TSO
	CXXBIND	Generate an executable module under TSO
	CXXMOD	Generate an executable module under TSO
	DLLRNAME	Run the DLLRNAME utility
Cataloged Procedures for z/OS C and z/OS C++	EDCDLLRN	Rename DLLs with the DLLRNAME utility
	EDCLIB	Maintain an object library
REXX EXECs for z/OS C	CC	Compile (new syntax - recommended approach)
	CDSECT	Run DSECT utility
	CPLINK	Interactively prelink and link a C program
	GENXLT	Generate a translate table
	ICONV	Run the character conversion utility
	LOCALEDEF	Produce a locale object

	Name	Task Description
Cataloged Procedures for z/OS C	CEEWG	Run
	CEEWL	Link
	CEEWLG	Link and run
	CEEXL	Bind an XPLINK z/OS C program
	CEEXLR	Bind and run an XPLINK z/OS C program
	CEEXR	Run an XPLINK z/OS C program
	EDCC	Compile
	EDCCB	Compile and Bind
	EDCCBG	Compile, bind, and run
	EDCCL	Compile and link-edit
	EDCCLG	Compile, link-edit, and run
	EDCCLIB	Compile and maintain an object library
	EDCI	Run IPA Link step
	EDCPL	Prelink and link-edit
	EDCCPLG	Compile, prelink, link-edit, and run.
	EDCDSECT	Run the DSECT Conversion Utility
	EDCGNXLT	Generate a translate table
	EDCICONV	Run the character conversion utility
	EDCLDEF	Produce a locale object
	EDCXCB	Compile and bind an XPLINK z/OS C program
	EDCXCBG	Compile, bind, and run an XPLINK z/OS C program
	EDCXI	Run IPA Link step for XPLINK

	Name	Task Description
Cataloged procedures for z/OS C	EDCXLDEF	Create z/OS C source from a locale, compile, and bind the XPLINK program to produce an XPLINK locale object
REXX EXECs for z/OS C++	CXX	Compile under TSO
Cataloged procedures for z/OS C++	CBCC	Compile
	CBCCB	Compile and bind
	CBCCBG	Compile, bind and run
	CBCB	Bind
	CBCBG	Bind and run
	CBCCL	Compile, prelink and link
	CBCCLG	Compile, prelink, link and run
	CBCG	Run
	CBCI	Run IPA Link step
	CBCL	Prelink and link
	CBCLG	Prelink, link and run
	CBCXB	Bind an XPLINK z/OS C++ Program
	CBCXBG	Bind and run an XPLINK z/OS C++ Program
	CBCXCB	Compile and bind an XPLINK z/OS C++ program
	CBCXCBG	Compile, bind, and run an XPLINK z/OS C++ program
	CBCXG	Run an XPLINK z/OS C++ program
	CBCXI	Run IPA Link for XPLINK

Tailoring PROCs, REXX EXECs, and EXECs

Your system programmer must modify the PROCs, and REXX EXECs before they are used. For example, the prefix symbolic parameters `LIBPRFX` and `LNGPRFX` should be changed from the defaults supplied by IBM to the high-level qualifier that you chose to install the z/OS C/C++ compiler and z/OS Language Environment.

The following data sets contain the PROCs and REXX EXECs that are to be modified:

- `CBC.SCCNPRC`
- `CBC.SCCNUTL`
- `CEE.SCEEPROC`
- `CEE.SCEECLST`

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 C/C++ source file name, PDS name of source files, or directory name of source files. For IPA link procedures (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 procedures that do not have an execution step (EDCC, EDCCL, and EDCPL). For the cataloged procedures containing a link-edit step, specify the name of the file where the load module is to be stored. For cataloged procedures without a link-edit step, 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.
BPARM	Bind utility options: If two contradictory options are specified, the last is accepted and the first ignored.
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 runtime (Go step) options and parameters: If two contradictory Language Environment runtime options are specified, the last is accepted and the first ignored.
CRUN	Compile step execution runtime parameters for the z/OS C/C++ compiler.
IRUN	IPA Link step runtime parameters: for the z/OS 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 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 section of the *z/OS C/C++ Programming Guide* for more information about the attributes that are used when opening different types of files.

Table 52. Cross Reference of Data Set Used and Job Step

DD Statement	COMPILE	IPA Link	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	X
SYSUTx	X	X			X (SYSUT1)		
IPACNTL		X					

Note: [1] Optional data sets, if the compiler and runtime library are installed in the 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.

Note: You should check the defaults at your site for `SYSOUT=*`

Table 53. 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 C/C++ system header files for a source program. SYSLIB must be a PDS (`DSORG=PO`) *and* `RECFM=VS, V, VB, VBS, F, FB LRECL≤32760`. For more information on searching system header files, see “SEARCH | NOSEARCH” on page 187.

Table 53. 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	SYSLIN	Data set for object module. One of the following: • RECFM=F or FS • RECFM=FB or FBS. It can be a PDS.
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 C/C++ compiler and run-time library modules. STEPLIB must be a PDS (DSORG=PO) with RECFM=U, BLKSIZE≤32760.
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 and SYSUT4	Work data sets. LRECL=80 and RECFM=F or FB or FBS.
COMPILE	SYSUT5, SYSUT6, SYSUT7, SYSUT8, and SYSUT14	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 C, or 882 in z/OS 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.
COMPILE	TEMPINC (C++ only)	Template instantiation file. Must be a PDS. 72≤LRECL≤32760, RECFM=VS, V, VB, VBS, F or FB (default is V).

Table 53. 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	USERLIB	User header files. Must be a PDS. LRECL≤32760, and RECFM=VS, V, VB, VBS, F or FB. For more information on searching user header files, see "SEARCH | NOSEARCH" on page 187.
IPA Link	SYSIN	Data set containing object module for the IPA Link step. LRECL=80 and RECFM=F or FB.
IPA Link	IPACNTL	IPA Link control file directives. RECFM=VS, V, VB, VBS, F, FB, FBS, or FS, LRECL≤32760. It can be a PDS.
IPA Link	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 must be cataloged.
IPA Link	SYSLIN	Data set for object module. One of the following: • RECFM=F or FS • RECFM=FB or FBS It can be a PDS.
IPA Link	SYSOUT	Data set for displaying compiler error messages. LRECL=137, RECFM=VBA, BLKSIZE=882. (Defaults for SYSOUT=*).
IPA Link	STEPLIB	Data set for z/OS C/C++ compiler/runtime library modules. STEPLIB must be a PDS (DSORG=PO) with RECFM=U, BLKSIZE≤32760.
IPA Link	SYSCPRT	Output data set for IPA Link step listings. LRECL=137, RECFM=VBA, BLKSIZE=882 (default for SYSOUT=*).
IPA Link	SYSUT1 and SYSUT4	Work data sets. LRECL=80 and RECFM=F or FB or FBS.

Table 53. 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=*)
IPA Link	SYSUT5, SYSUT6, SYSUT7, SYSUT8, and SYSUT14	Work data sets. LRECL=3200, RECFM=FB, and BLKSIZE=3200*n* (where *n* is an integer value).
IPA Link	SYSUT9	Work data set. LRECL=137, RECFM=VB, and BLKSIZE=137*n* (where *n* is an integer value).
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 containing object module for the binder. LRECL=80 and RECFM=F, FB or FBS.
BIND	SYSLIB	Data set for binder automatic call library.
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 (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. LRECL=80 and RECFM=F, FB or FBS.
PLKED	SYSLIB	Data set for prelinkage automatic call library. SYSLIB must be cataloged *and* LRECL=80 and RECFM=F or FB or FBS. DSORG=PO

Table 53. 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	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.
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 C/C++ autocall library. SYSLIB must be a PDS (DSORG=PO) *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	SYSUT1	Work data set. The data set attributes will be supplied by the linkage editor.
GO	STEPLIB	Runtime libraries. STEPLIB must be a PDS (DSORG=PO) *and* have the attributes RECFM=U and BLKSIZE≤32760.

Table 53. 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	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 the data set must have the attributes RECFM=VBA, LRECL=125, and BLKSIZE=882.
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. It must be partitioned and not concatenated and 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 for target error and warning messages. 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 67. Example Compilation for z/OS 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 68. Example Compilation for z/OS C++ Using CBCC

Other z/OS C Utilities

Starting with C/C++ for MVS/ESA 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/ESA V3R2 release of the compiler. This section 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 69. Utilities for z/OS C

For a description of `CXXMOD` see “Prelinking and Linking under TSO” on page 509.

Using the Old Syntax for CC

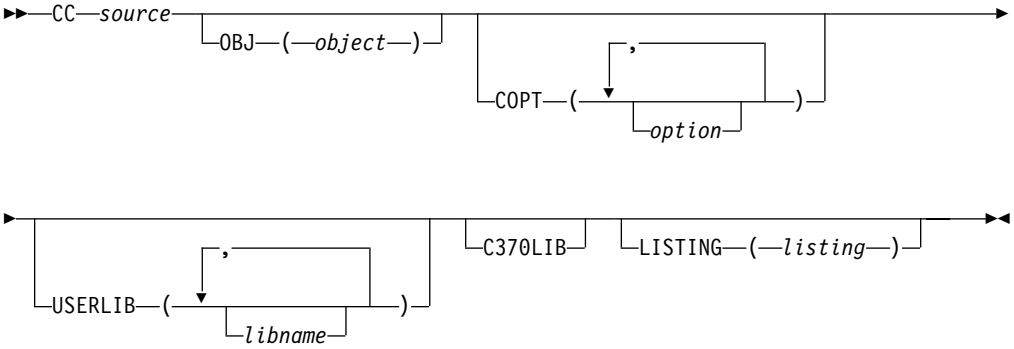
The CC command can now be invoked using a new syntax. At installation time, your system programmer can customize the `CC EXEC` to accept:

- Only the old syntax (the one supported by compilers prior to 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 Hierarchical File System (HFS) 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 HFS files.

For information on the new syntax, see "Using the CC and CXX REXX EXECs" on page 311. 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.

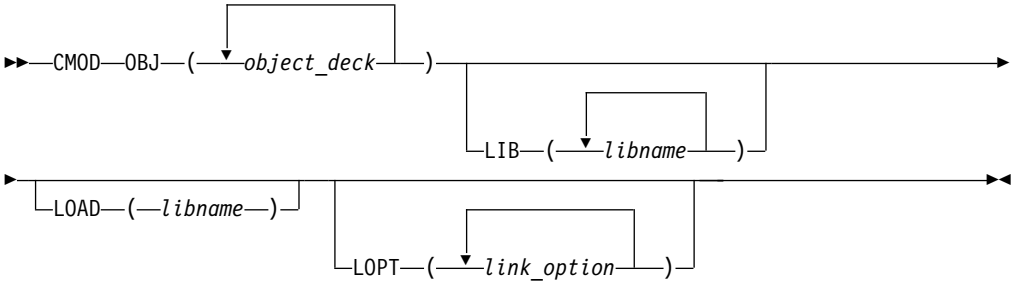
The following rules apply when you use the old syntax for the `CC REXX EXEC`:
- When you are specifying a data set name, if the name is not enclosed in single quotation mark ('), your user prefix will be added to the beginning of the data set name. If the data set name is enclosed in quotation marks, 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.

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:



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.

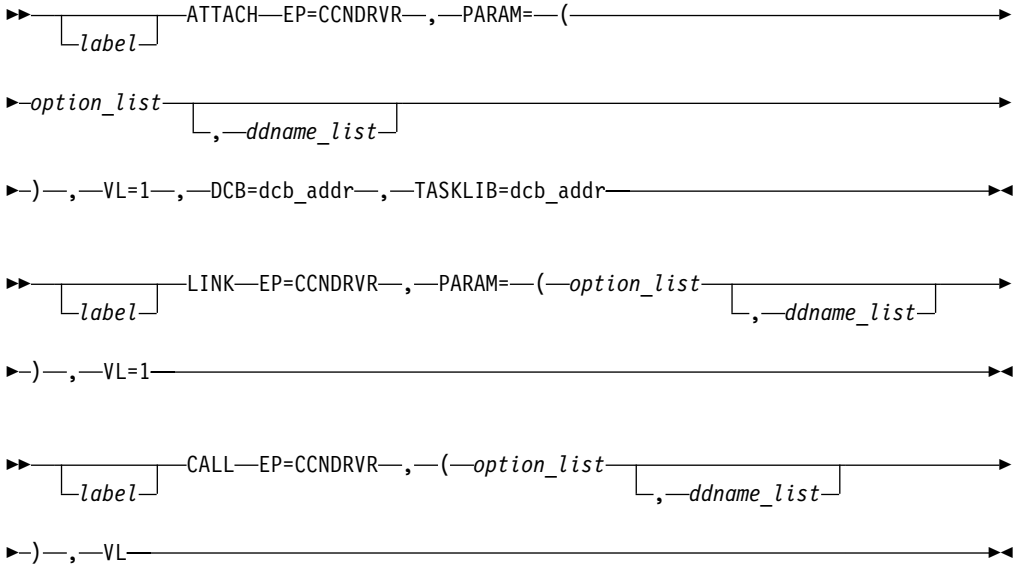
LOPT Specifies the options that you want to pass to the linkage editor. All options are passed to the TSO LINK command.

A non-zero return code indicates that an error has occurred. For diagnostic information, refer to “Appendix C. Diagnosing Problems and the PMR/APAR Process” on page 529. CMOD can also return the return code from LINK. See the appropriate book in your TSO library for more information on LINK.

Appendix E. Calling the Compiler from Assembler

To invoke the compiler dynamically under z/OS, you can use macro instructions such as `ATTACH`, `LINK`, or `CALL` in an assembler language program. For complete information on these macro instructions, refer to the list of manuals in *z/OS Information Roadmap*.

The following is the syntax of each macro instruction:
where:



`EP` — Specifies the symbolic name of the z/OS 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 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

You can omit an alternative ddname from the list by entering binary zeros in its 8-byte entry, or if it is at the end of the list, by shortening the list. If you omit the ddname, the compiler assumes the standard ddname.

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 128.

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.

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 70. 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 70. Using the Assembler ATTACH Macro (Part 2 of 2)

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
//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 71. JCL for the Assembler ATTACH Macro

Note that the sharing of resources between attached programs is not supported.

```
*********************************************************************
*                                                                   *
*  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 72. 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 72. Using the Assembler LINK Macro (Part 2 of 2)

```
//*-------------------------------------------------------------------
//* 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=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=...
//SYSUT4    DD DSN=...
//SYSUT5    DD DSN=...
//SYSUT6    DD DSN=...
//SYSUT7    DD DSN=...
//SYSUT8    DD DSN=...
//SYSUT9    DD DSN=...
//SYSUT10   DD SYSOUT=*
//SYSUT14   DD DSN=...
//SYSUT15   DD DSN=...
//SYSEVENT  DD DSN=...
//NEWRLIB   DD DSN=MYID.MYHDR.FILES,DISP=SHR
//*-------------------------------------------------------------------
```

Figure 73. JCL for the Assembler LINK 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 be in the LPA or       *
*  be 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 74. 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 74. Using the Assembler CALL Macro (Part 2 of 2)

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=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=...
//SYSUT15   DD SYSOUT=*
//NEWRLIB   DD DSN=MYID.MYHDR.FILES,DISP=SHR
//*--------------------------------------------------------------------
```

Figure 75. JCL for the Assembler CALL Macro

Appendix F. c89 — Compile, link-edit and assemble a z/OS C program and create an executable file

Format

c89 | **cc** | **c++** | **cxx** [**–+CcEFfgOpqrsVv012**]
[**–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.l*]... [*file.a*]... [**–l** *libname*]...

Note: The **I** option signifies an uppercase **i**, not a lowercase **L**.

Description

c89 and **cc** compile, assemble, and link-edit z/OS C programs; **c++** does the same for z/OS 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*.
- **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 V1R2, the C++ compiler supports the ISO 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 documention which describes the program you use for that step and the book which describes any messages issued by that program, and prefixes to those messages:

Table 54. c89, cc, and c++ Programs and Reference Documentation

Step Name	Book Describing Options and How to Call Program	Book Containing Messages Issued by Program	Prefix of Messages Issued by Program
ASSEMBLE	*HLASM Programmer's Guide*	*HLASM Programmer's Guide*	ASMA
COMPILE, IPACOMP, TEMPINC, IPATEMP, IPALINK	*z/OS C/C++ User's Guide*	*z/OS C/C++ User's Guide* for OS/390 V2R10 and z/OS V1R1.*z/OS C/C++ Messages* for z/OS V1R2.	CBC for OS/390 V2R10 and z/OS V1R1; CCN for z/OS V1R2

Table 54. c89, cc, and c++ Programs and Reference Documentation (continued)

Step Name	Book Describing Options and How to Call Program	Book Containing Messages Issued by Program	Prefix of Messages Issued by Program
PRELINK	*z/OS Language Environment Programming Guide* and *z/OS C/C++ User's Guide*	*z/OS Language Environment Debugging Guide*	EDC
LINKEDIT (Program Management Binder)	*z/OS DFSMS Program Management*	*z/OS MVS System Messages, Vol 8 (IEF-IGD)*	IEW2

Execution of any Language Environment program (including **c89** and the z/OS 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 the *z/OS Introduction and Release Guide* for further details. Also see {_CLIB_PREFIX} and {_CLIB_PREFIX} in "Environment Variables" on page 589 for information about the names of the z/OS C/C++ Optional Feature data sets that must be made available to **c89/cc/c++**.

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 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 **{_STEPS}** under "Environment Variables" on page 589 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 z/OS C++ source files in the **tempinc** subdirectory of the working directory (possibly creating and updating additional z/OS C++ source files during the automatic template generation step). After compiling all the z/OS 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 *z/OS C/C++ User's Guide* and *z/OS C/C++ Programming Guide*. Note that the **c++** command only supports using the **tempinc** subdirectory of the working directory for automatic template generation.

IPA is further described under the **–W** option on page 585.

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 **{_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 608. Also see Usage Note 13 on page 610.

–E Specifies that output of the compiler preprocessor phase be copied to **stdout**. Compilation into object and link-edit are not done.

–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 612.)

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 DFSMS Program Management*.

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 "**.

–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++**.

–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**, the debugger.

When specified for the compilation phase, the object file contains 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 guarantee that all the executables comprising the process is 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.s390.ibm.com/products/oe/index.html`

For more information on the **_BPX_PTRACE_ATTACH** environment variable, see *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

–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 quotes (") 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 608 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 (//). For 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 C/C++ compiler conversion rules (see Usage Note 4 on page 608). 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 hierarchical file system (HFS) 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 609 for a description of the usual places.

–0, –O (–1), –2
Specifies the level of compiler optimization (including inlining) to be used. The level **–1** (number one) is equivalent to **–O** (letter capital O). The level **–2** gives the highest level of optimization. The default is **–0** (level 0), no optimization and no inlining, when not using IPA (Interprocedural Analysis).

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 under the **–W** option on page 585.

If you compile your program to take advantage of **dbx** source-level debugging and specify **–g** (see the **–g** option on page 580), 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 the chapter on reentrancy in z/OS C/C++ in *z/OS C/C++ Programming Guide* or the description of the ROSTRING option in the *z/OS C/C++ User's Guide*.

–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 587 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**. For related information, see Usage Note 3 on page 608.

–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 then 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 608. Also see Usage Note 13 on page 610.

–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 612.) 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** or **–2** 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.

–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. 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 610.

–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. For example, to specify that an IPA compile should save source line number information, without writing a listing file, specify:

```
c89 -W I,list file.c
```

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. For 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 **{_STEPS}** under "Environment Variables" on page 589 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. For 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) the link-editing phase option DLL must be specified. For example:

  ```
  c89 –o outdll –W l,dll file.o
  ```

Most z/OS 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 C/C++ **#pragma export** compiler directive, or by using the z/OS 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 under Operands on page 589 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. For example, to compile and link a program into a DLL, you could specify:

```
c89 –o outdll –W c,dll,expo –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. For example:

```
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 HFS, 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.

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 C/C++ compiler.

When phase code **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. So for 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 i,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.

XPLINK (Extra Performance Linkage)

z/OS XPLINK provides improved performance for many C/C++ programs. The C/C++ **XPLINK** compiler option instructs the 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 the z/OS C/C++ **XPLINK** compiler option, refer to *z/OS C/C++ User's Guide*. 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 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. To indicate that different libraries should be concatenated, specify the **XPLINK** on **c89**. 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 {_PLIB_PREFIX} environment variable).

You cannot use **–W** to override the compiler options that correspond to **c89/cc/c++** options, with the following exceptions:

- Listing options (corresponding to **–V**)
- Inlining options (corresponding to **–O** and **–2**)
- 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

   COMPAT
   COMPAT=CURRENT

   DYNAM
   DYNAM=DLL

   LET LET=8

   LIST LIST=NOIMPORT

Note: References throughout this document to the link editor are generic references. **c89/cc/c++** specifically uses the Program Management binder for this function.

4. The z/OS C/C++ compiler is described in *z/OS C/C++ User's Guide*. Related information about the z/OS C/C++ runtime library, including information about DLL and IPA support, is described in *z/OS C/C++ Programming Guide*. Related information about the z/OS C and z/OS C++ languages, including information about compiler directives, is described in *C/C++ Language Reference*.
5. Since some compiler options are z/OS C–only and some compiler options are z/OS C++–only, 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. The prelinker is described in *z/OS C/C++ User's Guide*.
7. The *z/OS C/C++ User's Guide* also describes C/C++ compiler options. Any messages produced by it (CCN messages) are documented in *z/OS C/C++ Messages*.
8. You may see runtime 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 DFSMS Program Management*. 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 589 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 **{_EXTRA_ARGS}**. Also see Usage Note 3 on page 608 for related information.

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 **{_OSUFFIX_HOSTRULE}** and **{_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 under the **–W** option on page 585. 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 **{_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 under Options on page 582, 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 **{_IXXSUFFIX}** and **{_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*. If a partitioned data set is specified, more than one member name may be specified by separating each with a comma (,), for example:

```
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 **{_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 under Options on page 582, 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 under Options on page 582, 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*. If a partitioned data set is specified, more than one member name may be specified by separating each with a comma (,), for example:

```
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 609.

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 the z/OS C/C++ Object Library Utility, *z/OS C/C++ Programming Guide*. For more information about how load libraries are searched, see Usage Note 7 on page 609.

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*.

At the beginning of each environment variable description below, the name of the variable is shown in a *symbolic* notation. At the end of the description, the *actual* variable names used by the utilities are listed. The symbolic name is the same as the actual variable names, but omits the prefix and is enclosed in curly braces ({_variable_name}) to indicate that it is a symbolic name. Throughout the remainder of this command description, only the symbolic names are shown, but you must use the actual name when setting these variables. This means 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++** environment variables, the name shown must be prefixed with **_CXX** (for example, **_CXX_ACCEPTABLE_RC**).

Note: **c89/cc/c++** can accept parameters only in the syntax indicated here. A null values indicate that **c89/cc/c++** is to omit the corresponding parameters during dynamic allocation. Numbers in parentheses following the environment variable name correspond to usage notes, which begin on Page 607, and indicate specific usage information for the environment variable.

{_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.

Actual Variable Names: `_C89_ACCEPTABLE_RC`, `_CC_ACCEPTABLE_RC`, `_CXX_ACCEPTABLE_RC`

{_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"
```

Actual Variable Names: `_C89_ASUFFIX`, `_CC_ASUFFIX`, `_CXX_ASUFFIX`

{_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"
```

Actual Variable Names: `_C89_ASUFFIX_HOST`, `_CC_ASUFFIX_HOST`, `_CXX_ASUFFIX_HOST`

{_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"
```

Actual Variable Names: `_C89_CCMODE`, `_CC_CCMODE`, `_CXX_CCMODE`

{_CLASSLIB_PREFIX} (14,17)
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 {_CLASSLIB_PREFIX}.SCLBH.+ contain the z/OS C++ Class Library include (header) files.
- The data set {_CLASSLIB_PREFIX}.SCLBSID contains the z/OS C++ Class Library definition side-decks.

The following data sets are also used:

The data sets {_CLASSLIB_PREFIX}.SCLBDLL and {_CLASSLIBIB_PREFIX}.SCLBDLL2 contain the z/OS 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 varible:
`_CXX_CLIB_PREFIX`

Actual Variable Name: `_CXX_CLASSLIB_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 to be 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 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.`

Actual Variable Names: `_CXX_CLASSVERSION`

{_CLIB_PREFIX} (14,17)
The prefix for the following named data sets used during the compilation phase.

The following data sets are also used:

The data sets {_CLIB_PREFIX}.SCBCCMP and {_CLIB_PREFIX}.SCCNCMP contain the compiler programs called by **c89/cc/c++**.

The preceding 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 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:

```
"CBC"
```

Actual Variable Names: _C89_CLIB_PREFIX, _CC_CLIB_PREFIX, _CXX_CLIB_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"
```

Actual Variable Names: _C89_CMEMORY, _CC_CMEMORY, _CXX_CMEMORY

{_CMSGS} (14)

The Language Environment national language name used by the compiler program. A null value will cause the default Language Environment **NATLANG** runtime name to be used, and a non-null value must be a valid Language Environment **NATLANG** runtime option name (Language Environment runtime options are described in *z/OS Language Environment Programming Guide* . The default value is:

```
"" (null)
```

Actual Variable Names: _C89_CMSGS, _CC_CMSGS_RC, _CXX_CMSGS

{_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 {_CVERSION} set to a release prior to z/OS v1r2, the default value will be:

```
"CBCDRVR"
```

Actual Variable Names: _C89_CNAME, _CC_CNAME, _CXX_CNAME

{_CSUFFIX} (15)

The suffix by which **c89/cc/c++** recognizes a C source file. The default value is:

```
"c"
```

Actual Variable Names: _C89_CSUFFIX, _CC_CSUFFIX, _CXX_CSUFFIX

{_CSUFFIX_HOST} (15)
The suffix by which **c89/cc/c++** recognizes a C source data set. The default value is:

```
"C"
```

Actual Variable Names: _C89_CSUFFIX_HOST, _CC_CSUFFIX_HOST, _CXX_CSUFFIX_HOST

{_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 include (header) files provided by Language Environment are installed in the default location (in accordance with **{_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:

```
"{_PLIB_PREFIX}.SCEEH.H"
"{_PLIB_PREFIX}.SCEEH.SYS.H"
"{_PLIB_PREFIX}.SCEEH.ARPA.H"
"{_PLIB_PREFIX}.SCEEH.NET.H"
"{_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.

Actual Variable Names: _C89_CSYSLIB, _CC_CSYSLIB, _CXX_CSYSLIB

{_CVERSION}
The version of the C/C++ compiler to be invoked by **c89/cc/c++**. The setting of this variable allows **c89/cc/c++** to control which 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 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.
```

In order for **c89/cc/c++** to use the OS/390 Version 2 Release 10 C/C++ compiler and C++ Class Library, this variable should be set to the value:

```
0x220A0000
```

Actual Variable Names: _C89_CVERSION, _CC_CVERSION, _CXX_CVERSION

{_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++** command.

Actual Variable Name: _CXX_CXXSUFFIX

{_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++** command.

Actual Variable Names:_CXX_CXXSUFFIX_HOST

{_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

Actual Variable Names: _C89_DAMPLEVEL, _CC_DAMPLEVEL, _CXX_DAMPLEVEL

{_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"
```

Actual Variable Names: _C89_DAMPNAME, _CC_DAMPNAME, _CXX_DAMPNAME

{_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)"
```

Actual Variable Names: _C89_DCBF2008, _CC_DCBF2008, _CXX_DCBF2008

{_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)"
```

Actual Variable Names: _C89_DCBU, _CC_DCBU, _CXX_DCBU

{_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)"
```

Actual Variable Names: _C89_DCB121M, _CC_DCB121M, _CXX_DCB121M

{_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)"
```

Actual Variable Names: _C89_DCB133M, _CC_DCB133M, _CXX_DCB133M

{_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)"
```

Actual Variable Names: _C89_DCB137, _CC_DCB137, _CXX_DCB137

{_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)"
```

Actual Variable Names: _C89_DCB137A, _CC_DCB137A, _CXX_DCB137A

{_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)"
```

Actual Variable Names: _C89_DCB3200, _CC_DCB3200, _CXX_DCB3200

{_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)"
```

Actual Variable Names: _C89_DCB80, _CC_DCB80, _CXX_DCB80

{_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"
```

Actual Variable Names: _C89_ELINES, _CC_ELINES, _CXX_ELINES

{_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 page 579 for information about 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"
```

Actual Variable Names: _C89_EXTRA_ARGS, _CC_EXTRA_ARGS, _CXX_EXTRA_ARGS

{_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"
```

Actual Variable Names: _C89_ILCTL, _CC_ILCTL, _CXX_ILCTL

{_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 **{_CMSGS}** is. So if **{_CMSGS}** is set or defaults to "" (null), the default value is:

```
"" (null)
```

Actual Variable Names: _C89_ILMSGS, _CC_ILMSGS, _CXX_ILMSGS

{_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 **{_CNAME}** is. So if **{_CNAME}** is set or defaults to "CCNDRVR" the default value is:

```
"CCNDRVR"
```

Actual Variable Names: _C89_ILNAME, _CC_ILNAME, _CXX_ILNAME

{_ILSUFFIX} (15)
The suffix **c89/cc** uses when creating an IPA linker output file. The default value is:

```
"I"
```

Actual Variable Names: _C89_ILSUFFIX, _CC_ILSUFFIX, _CXX_ILSUFFIX

{_ILSUFFIX_HOST} (15)
The suffix **c89/cc** uses when creating an IPA linker output data set. The default value is:

```
"IPA"
```

Actual Variable Names: _C89_ILSUFFIX_HOST, _CC_ILSUFFIX_HOST, _CXX_ILSUFFIX_HOST

{_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 {_PSYSLIB} is set or defaults to, followed by whatever {_LSYSLIB} is set or defaults to.

Actual Variable Names: _C89_ILSYSLIB, _CC_ILSYSLIB, _CXX_ILSYSLIB

{_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 {_PSYSIX} is set or defaults to.

Actual Variable Names: _C89_ILSYSIX, _CC_ILSYSIX, _CXX_ILSYSIX

{_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 579). The default value is whatever {_LXSYSLIB} is set or defaults to.

Actual Variable Names: _C89_ILXSYSLIB, _CC_ILXSYSLIB, _CXX_ILXSYSLIB

{_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 579). The default value is whatever {_LXSYSIX}} is set or defaults to.

Actual Variable Names: _C89_ILXSYSIX, _CC_ILXSYSIX, _CXX_ILXSYSIX

{_INCDIRS} (22)
The directories used by **c89/cc/c++** as a default place to search for include files during compilation (before searching **{_INCLIBS}** and **{_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/ioclib/include
```

Actual Variable Names: _C89_INCDIRS, _CC_INCDIRS, _CXX_INCDIRS

{_INCLIBS} (22)
The directories used by **c89/cc/c++** as a default place to search for include files during compilation (after searching **{_INCDIRS}** and before searching **{_CSYSLIB}**). The default value depends on whether or not **c++** is being used. If **c++** is not being used the default value is:

```
"//'{_PLIB_PREFIX}.SCEEH.+'"
```

If **c++** is being used, the default value is:

```
"//'{_PLIB_PREFIX}.SCEEH.+' //'{_CLIB_PREFIX}.SCLBH.+'"
```

Actual Variable Names: _C89_INCLIBS, _CC_INCLIBS, _CXX_INCLIBS

{_ISUFFIX} (15)
The suffix by which **c89/cc/c++** recognizes a preprocessed C source file. The default value is:

```
"i"
```

Actual Variable Names: _C89_ISUFFIX, _CC_ISUFFIX, _CXX_ISUFFIX

{_ISUFFIX_HOST} (15)
The suffix by which **c89/cc/c++** recognizes a preprocessed (expanded) C source data set. The default value is:

```
"CEX"
```

Actual Variable Names: _C89_ISUFFIX_HOST, _CC_ISUFFIX_HOST, _CXX_ISUFFIX_HOST

{_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++** command.

Actual Variable Names: _CXX_IXXSUFFIX

{_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++** command.

Actual Variable Names: _CXX_IXXSUFFIX_HOST

{_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"
```

Actual Variable Names: _C89_LIBDIRS, _CC_LIBDIRS, _CXX_LIBDIRS

{_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 **{_PSYSLIB}** libraries always precede the **{_LSYSLIB}** libraries when resolving symbols in the link-editing phase. The default value is the concatenation:

```
"{_PLIB_PREFIX}.SCEELKEX"
"{_PLIB_PREFIX}.SCEELKED"
"{_SLIB_PREFIX}.CSSLIB"
```

Actual Variable Names: _C89_LSYSLIB, _CC_LSYSLIB, _CXX_LSYSLIB

{_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 579). The default value is the concatenation:

```
"{_PLIB_PREFIX}.SCEEBIND"
"{_SLIB_PREFIX}.CSSLIB"
```

Actual Variable Names: _C89_LXSYSLIB, _CC_LXSYSLIB, _CXX_LXSYSLIB

{_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 579). 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. If **c++** is not being used, the default value is the list:

```
"{_PLIB_PREFIX}.SCEELLIB(CELHS003,CELHS001)"
```

If **c++** is being used with {_PVERSION} and {_CLASSVERSION} defaulted to the current z/OS release, the default value is the list concatenation:

```
"{_PLIB_PREFIX}.SCEELIB(CELHS003,CELHS001,CELHSCPP,C128)"
"{_CLIB_PREFIX}.SCLBSID(IOC,IOSTREAM,COMPLEX,COLL)"
```

If **c++** is being used with {_PVERSION} and {_CLASSVERSION} set to a release prior to z/OS Version 1 Release 2, the default value is the list concatenation:

```
"{_PLIB_PREFIX}.SCEELIB(CELHS003,CELHS001,CELHSCPP)"
"{_CLIB_PREFIX}.SCLBSID(ASCCOLL,COMPLEX,IOSTREAM)"
```

Actual Variable Names: _C89_LXSYSIX, _CC_LXSYSIX, _CXX_LXSYSIX

{_MEMORY}
A suggestion as to the use of C/C++ Runtime 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"
```

Actual Variable Names: _C89_MEMORY, _CC_MEMORY, _CXX_MEMORY

{_NEW_DATACLAS} (18)
The DATACLAS parameter used by **c89/cc/c++** for any new datasets it creates. The default value is:

```
"" (null)
```

Actual Variable Names: _C89_NEW_DATACLAS, _CC_NEW_DATACLAS, _CXX_NEW_DATACLAS

{_NEW_DSNTYPE} (18, 20)
The DSNTYPE parameter used by **c89/cc/c++** for any new data sets it creates. The default value is:

```
"" (null)
```

Actual Variable Names: _C89_NEW_DSNTYPE, _CC_NEW_DSNTYPE, _CXX_NEW_DSNTYPE

{_NEW_MGMTCLAS} (18)
The MGMTCLAS parameter used by **c89/cc/c++** for any new datasets it creates. The default value is:

```
"" (null)
```

Actual Variable Names: _C89_NEW_MGMTCLAS, _CC_NEW_MGMTCLAS, _CXX_NEW_MGMTCLAS

{_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))"
```

Actual Variable Names: _C89_NEW_SPACE, _CC_NEW_SPACE, _CXX_NEW_SPACE

{_NEW_STORCLAS} (18)
The STORCLAS parameter used by **c89/cc/c++**for any new data sets it creates. The default value is:

```
"" (null)
```

Actual Variable Names: _C89_NEW_STORCLAS, _CC_NEW_STORCLAS, _CXX_NEW_STORACLAS

{_NEW_UNIT} (18)
The UNIT parameter used by **c89/cc/c++** for any new data sets it creates. The default value is:

```
"" (null)
```

Actual Variable Names: _C89_NEW_UNIT, _CC_NEW_UNIT, _CXX_NEW_UNIT

{_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)
```

Actual Variable Names: _C89_OPERANDS, _CC_OPERANDS, _CXX_OPERANDS

{_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)
```

Actual Variable Names: _C89_OPTIONS, _CC_OPTIONS, _CXX_OPTIONS

{_OSUFFIX} (15)
The suffix by which **c89/cc/c++** recognizes an object file. The default value is:

```
"o"
```

Actual Variable Names: _C89_OSUFFIX, _CC_OSUFFIX, _CXX_OSUFFIX

{_OSUFFIX_HOST} (15)
The suffix by which **c89/cc/c++** recognizes an object data set. The default value is:

```
"OBJ"
```

Actual Variable Names: _C89_OSUFFIX_HOST, _CC_OSUFFIX_HOST, _CXX_OSUFFIX_HOST

{_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 **{_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 **{_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.

Actual Variable Names: _C89_OSUFFIX_HOSTQUAL, _CC_OSUFFIX_HOSTQUAL, _CXX_OSUFFIX_HOSTQUAL

{_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".

Actual Variable Names: _C89_OSUFFIX_HOSTRULE, _CC_OSUFFIX_HOSTRULE, _CXX_OSUFFIX_HOSTRULE

{_PLIB_PREFIX} (14,17)

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 {_PLIB_PREFIX}.SCEEH.+ contain the include (header) files for use with the runtime library functions (where + can be any of H, SYS.H, ARPA.H, NET.H, and NETINET.H).
- The data set {_PLIB_PREFIX}.SCEEMAC contains COPY and MACRO files to be used during assembly.

- The data sets {_PLIB_PREFIX}.SCEEOBJ and {_PLIB_PREFIX}.SCEECPP contain runtime library bindings which exploit constructed reentrancy, used during the link-editing phase of non-XPLINK programs.
- The data set {_PLIB_PREFIX}.SCEELKEX contains C runtime library bindings which exploit L-names used during the link-editing phase of non-XPLINK programs. For more information about L-names, see the usage note 23 on page 612.
- The data set {_PLIB_PREFIX}.SCEELKED contains all other Language Environment runtime library bindings, used during the link-editing phase of non-XPLINK programs.
- The data set {_PLIB_PREFIX}.SCEEBIND contains all static Language Environment runtime library bindings, used during the link-editing phase of XPLINK programs.
- The data set {_PLIB_PREFIX}.SCEEBND2 contains all static Language Environment runtime library bindings, used during the link-editing phase of XPLINK programs.
- The data set {_PLIB_PREFIX}.SCEELIB contains the definition side-decks for the runtime library bindings (CELHS003 and CELHSCPP), and the Language Environment Callable Services (member CELHS001), used during the link-editing phase of XPLINK programs.

The following data sets are also used:

- The data sets {_PLIB_PREFIX}.SCEERUN and {_PLIB_PREFIX}.SCEERUN2 contains the runtime library programs.

The above 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"
```

Actual Variable Names: _C89_PLIB_PREFIX, _CC_PLIB_PREFIX, _CXX_PLIB_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"
```

_C89_PMEMORY, _CC_PMEMORY, _CXX_PMEMORY

{_PMSGS} (14)

The name of the message data set used by the prelinker program. It must be a member of the cataloged data set {_PLIB_PREFIX}.SCEEMSGP. The default value is:

```
 "EDCPMSGE"
```

Actual Variable Names: _C89_PMSGS, _CC_PMSGS, _CXX_PMSGS

{_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 {_PLIB_PREFIX}.SCEERUN data set. The default value is:

```
"EDCPRLK"
```

Actual Variable Names: _C89_PNAME, _CC_PNAME, _CXX_PNAME

{_PSUFFIX} (15)
The suffix **c89/cc/c++** uses when creating a prelinker (composite object) output file. The default value is:

```
"p"
```

Actual Variable Names: _C89_PSUFFIX, _CC_PSUFFIX, _CXX_PSUFFIX

{_PSUFFIX_HOST} (15)
The suffix **c89/cc/c++** uses when creating a prelinker (composite object) output data set. The default value is:

```
"CPOBJ"
```

Actual Variable Names: _C89_PSUFFIX_HOST, _CC_PSUFFIX_HOST, _CXX_PSUFFIX_HOST

{_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 {_PVERSION} and {_CLASSVERSION} set or defaulted to the current z/OS release, the default value is the list concatenation:

```
"{_PLIB_PREFIX}.SCEELIB(C128)"
"{_CLIB_PREFIX}.SCLBSID(IOC,IOSTREAM,COMPLEX,COLL)"
```

If **c++** is being used with {_PVERSION} and {_CLASSVERSION} set to a release prior to z/OS Version 1 Release 2, the default value is the list:

```
"{_CLIB_PREFIX}.SCLBSID(ASCCOLL,COMPLEX,IOSTREAM)"
```

Actual Variable Names: _C89_PSYSIX, _CC_PSYSIX, _CXX_PSYSIX

{_PSYSLIB} (16)
The system library data set list to be used to resolve symbols during the non-XPLINK link-editing phase. The **{_PSYSLIB}** libraries always precede the **{_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:

```
"{_PLIB_PREFIX}.SCEEOBJ"
```

If **c++** is being used, the default value is the list:

```
"{_PLIB_PREFIX}.SCEEOBJ"
"{_PLIB_PREFIX}.SCEECPP"
```

Actual Variable Names: _C89_PSYSLIB, _CC_PSYSLIB, _CXX_PSYSLIB

{_PVERSION} (27)
The version of the Language Environment 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++** processng 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 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
```

Actual Variable Names: _C89_PVERSION, _CC_PVERSION, _CXX_PVERSION

{_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 {_SLIB_PREFIX}.CSSLIB contains the z/OS UNIX assembler callable services bindings. The data sets {_SLIB_PREFIX}.MACLIB and {_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"
```

Actual Variable Names: _C89_SLIB_PREFIX, _CC_SLIB_PREFIX, _CXX_SLIB_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"
```

Actual Variable Names: _C89_SNAME, _CC_SNAME, _CXX_SNAME

{_SSUFFIX} (15)
The suffix by which **c89/cc/c++** recognizes an assembler source file. The default value is:

```
"s"
```

Actual Variable Names: _C89_SSUFFIX, _CC_SSUFFIX, _CXX_SSUFFIX

{_SSUFFIX_HOST} (15)
The suffix by which **c89/cc/c++** recognizes an assembler source data set. The default value is:

```
"ASM"
```

Actual Variable Names: _C89_SSUFFIX_HOST, _CC_SSUFFIX_HOST, _CXX_SSUFFIX_HOST

{_SSYSLIB} (16)
The system library data set concatenation to be used to find COPY and MACRO files during assembly. The default concatenation is:

```
"{_PLIB_PREFIX}.SCEEMAC"
"{_SLIB_PREFIX}.MACLIB"
"{_SLIB_PREFIX}.MODGEN"
```

Actual Variable Names: _C89_SSYSLIB, _CC_SSYSLIB, _CXX_SSYSLIB

{_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 under the **–W** option on page 585.

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

For 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 an HFS 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.

Actual Variable Names: `_C89_STEPS, _CC_STEPS, _CXX_STEPS`

{_SUSRLIB} (16)

The user library data set concatenation to be used to find COPY and MACRO files during assembly (before searching **{_SSYSLIB}**). The default value is:

`"" (null)`

Actual Variable Names: `_C89_SUSRLIB, _CC_SUSRLIB, _CXX_SUSRLIB`

{_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 **{_STEPS}**. Only the prelinker and IPA linker output can be captured using this variable.

For example, to capture the prelinker output, set this variable to:

```
"0xfffffffD" or the equivalent, -3
```

The default value is :

```
"0xffffffff" or the equivalent, -1
```

Actual Variable Names: _C89_TMPS, _CC_TMPS, _CXX_TMPS

{_WORK_DATACLAS} (18)
The DATACLAS parameter used by **c89/cc/c++** for unnamed temporary (work) data sets. The default value is:

```
"" (null)
```

Actual Variable Names: _C89_WORK_DATACLAS, _CC_WORK_DATACLAS, _CXX_WORK_DATACLAS

{_WORK_DSNTYPE} (18, 20)
The DSNTYPE parameter used by **c89/cc/c++** for unnamed temporary (work) data sets. The default value is:

```
"" (null)
```

Actual Variable Names: _C89_WORK_DSNTYPE, _CC_WORK_DSNTYPE, _CXX_WORK_DSNTYPE

{_WORK_MGMTCLAS} (18)
The MGMTCLAS parameter used by **c89/cc/c++** for unnamed temporary (work) data sets. The default value is:

```
"" (null)
```

Actual Variable Names: _C89_WORK_MGMTCLAS, _CC_WORK_MGMTCLAS, _CXX_WORK_MGMTCLAS

{_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))"
```

Actual Variable Names: _C89_WORK_SPACE, _CC_WORK_SPACE, _CXX_WORK_SPACE

{_WORK_STORCLAS} (18)
The STORCLAS parameter used by **c89/cc/c++** for unnamed temporary (work) data sets. The default value is:

```
"" (null)
```

Actual Variable Names: _C89_WORK_STORCLAS, _CC_WORK_STORCLAS, _CXX_WORK_STORCLAS

{_WORK_UNIT} (18)
The UNIT parameter used by **c89/cc/c++** for unnamed temporary (work) data sets. The default value is:

"SYSDA"

Actual Variable Names: _C89_WORK_UNIT, _CC_WORK_UNIT, _CXX_WORK_UNIT

{_XSUFFIX} (15)
The suffix by which **c89/cc/c++** recognizes a definition side-deck file of exported symbols. The default value is:
"x"

Actual Variable Names: _C89_XSUFFIX, _CC_XSUFFIX, _CXX_XSUFFIX

{_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"

Actual Variable Names: _C89_XSUFFIX_HOST, _CC_XSUFFIX_HOST, _CXX_XSUFFIX_HOST

Files

libc.a C/C++ Runtime Library function library (see Usage Note 7 on page 609).

libm.a C/C++ Runtime Library math function library (see Usage Note 7 on page 609).

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 608).

/lib The usual place to search for runtime library bindings (see Usage Note 7 on page 609).

/usr/lib
The usual place to search for runtime library bindings (see Usage Note 7 on page 609).

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 {_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 quotes.

   To escape these special characters, either enclose the option-argument or operand in double quotes, or precede each character with a backslash.

3. Some **c89/cc/c++** behavior applies only to files (and not to data sets).
   - The **–o** option does not allow a source file (*file.C*, *file.c*, *file.i*, *file.s*) to be specified.
   - 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 Language Environment. 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 **{_CSYSLIB}** for information.

   When include files are MVS PDS members, z/OS 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 dataset 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 Language Environment are installed into the hierarchical file system in the default location (in accordance with the **{_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 **{_CSYSLIB}**, **{_INCDIRS}**, and **{_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 UPCONV compiler options are set according to whether or not **cc** was invoked. The LANGLVL compiler option affects z/OS 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 not defined or undefined except by this compiler option. Together the

LANGLVL and UPCONV options affect the language rules used by the compiler. The LANGLVL option for C++ has individual suboptions which control the C++ language features.

For more information about these compiler options, see “Chapter 5. Compiler Options” on page 61. For more information about z/OS C/C++ predefined macros, see *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:

- Other than **cc**

  –W "0,langlvl(ansi),noupconv"
- **cc**

  –W "0,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 **{_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 runtime library bindings are not contained in them.

   Instead, MVS data sets installed with the Language Environment runtime library are used as the usual place to resolve runtime 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 **{_PLIB_PREFIX}** description, as well as descriptions of the following environment variables:

   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 579.

   {_ILSYSLIB}
   {_ILSYSIX}
   {_LSYSLIB}
   {_PSYSIX}
   {_PSYSLIB}

   This list of environment variables affects the link-editing phase of **c89**, but only for XPLINK link-editing. See XPLINK (Extra Performance Linkage) in “Options” on page 579.

   {_ILXSYSLIB}
   {_ILXSYSIX}
   {_LXSYSLIB}
   {_LXSYSIX}
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 C/C++ compiler and the Language Environment runtime library).
11. **c89/cc/c++** operation depends on the correct setting of their installation and configuration environment variables (see “Environment Variables” on page 589

page 589). 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/cc/c++** 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**, **–V**, **–c**, **–g** and **–s** (also ignores any *file.s* files).

    –g Overrides **–0**, **–O**, **–1**, **–2**, and **–s**.

    –s Overrides **–g** (the last one specified is honored).

    –0, –O, –1, –2, –V, –c
    : All are honored if not overridden. **–0**, **–O**, **–1**, and **–2**, override each other (the last one specified is honored).

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.
{_CMSGS}
{_CNAME}
{_DAMPNAME}
{_ILNAME}
{_ILMSGS}
{_PMSGS}
{_PNAME}
{_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:
{_ASUFFIX}
{_ASUFFIX_HOST}
{_CSUFFIX}
{_CSUFFIX_HOST}
{_CXXSUFFIX}
{_CXXSUFFIX_HOST}
{_ISUFFIX}
{_ISUFFIX_HOST}
{_ILSUFFIX}
{_ILSUFFIX_HOST}
{_IXXSUFFIX}
{_IXXSUFFIX_HOST}
{_OSUFFIX}
{_OSUFFIX_HOST}
{_PSUFFIX}
{_PSUFFIX_HOST}
{_SSUFFIX}
{_SSUFFIX_HOST}
{_XSUFFIX}
{_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:
{_CSYSLIB}
{_ILSYSLIB}
{_ILSYSIX}
{_ILXSYSLIB}
{_ILXSYSIX}
{_LSYSLIB}
{_LXSYSLIB}
{_LXSYSIX}
{_PSYSIX}
{_PSYSLIB}
{_SSYSLIB}
{_SUSRLIB}
17. The following environment variables can be at most 44 characters in length:
{_CLASSLIB_PREFIX}
{_CLIB_PREFIX}
{_PLIB_PREFIX}
{_SLIB_PREFIX}
18. The following environment variables can be at most 63 characters in length:
{_NEW_DATACLAS}

{_NEW_DSNTYPE}
{_NEW_MGMTCLAS}
{_NEW_SPACE}
{_NEW_STORCLAS}
{_NEW_UNIT}
{_WORK_DATACLAS}
{_WORK_DSNTYPE}
{_WORK_MGMTCLAS}
{_WORK_SPACE}
{_WORK_STORCLAS}
{_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.
    {_NEW_SPACE}
    {_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):
    {_NEW_DSNTYPE}
    {_WORK_DSNTYPE}
21. The following environment variables can be at most 127 characters in length:
    {_DCBF2008}
    {_DCBU}
    {_DCB121M}
    {_DCB133M}
    {_DCB137}
    {_DCB137A}
    {_DCB3200}
    {_DCB80}

    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:
    {_INCDIRS}
    {_INCLIBS}
    {_LIBDIRS}
    {_OPTIONS}
    {_OPERANDS}
23. An S-name is a *short* external symbol name, such as produced by the z/OS C/C++ compiler when compiling z/OS C programs with the `NOLONGNAME` option. An L-name is a *long* external symbol name, such as produced by the z/OS C/C++ compiler when compiling z/OS C programs with the `LONGNAME` option.

24. The C/C++ Runtime 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, see *z/OS C/C++ Programming Guide*. When coding in assembler language, the code must not violate reentrancy. If it does, the resulting executable may not be reentrant.
26. When shell variable _**MAKE_BI** is set to YES, **sh** will use the built-in **c89**, **cc**, **c++**, and **make** commands instead of **/bin/make, /bin/c89, /bin/cc**, and **/bin/c++**. **make** will also call the built-in **c89**, **cc**, and **c++** commands, instead of **/bin/c89, /bin/cc**, and **/bin/c++**. For more information, see *z/OS UNIX System Services Planning*. describes UNIX built-in commands.
27. The {_CVERSION}, {_PVERSION} and {_CLASSVERSION} environment variables are set to a hex string in the format 0xPVVRRMMMM where P is product, VV is version, RR is release and MMMM is modification level .

    To use the z/OS V1R2 compiler, specify CVRESION=0x44020000. To use the OS/390 V2R10 compiler, specify CVRESION=0x220A0000. To use the z/OS V1R2 class library, specify CLASSVRESION=0x44020000. And to use the OS/390 V2R10 class library, specify CVRESION=0x220A0000.

    Note: You can use the OS/390 V2R10 class library with either the z/OS V1R2 compiler or the OS/390 V2R10 compiler, but to do so, you must target back to OS/390 V2R10. Otherwise, you won't be able to get to the V3 headers.

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**, and **—2** options
- DLL support
- IPA optimization support
- The behavior of the **—o** option in combination with the **—c** option and a single source file.

Related Information

See the information on the following utilities in *z/OS UNIX System Services Command Reference*: **ar**, **dbx**, **file**, **lex**, **make**, **makedepend**, **nm**, **strings**, **strip**, **yacc**

Appendix G. Layout of the Events File

This appendix 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 110.

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 78 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 76. simple.c

```
1   add some;
2   errors in the header file;
```

Figure 77. 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 78. 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 78. Table 55 on page 616 describes the FILEID identifiers.

```
FILEID 0 1 0 10 ./simple.c
       A B C D  E
```

Table 55. Explanation of the FILEID 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	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 78 on page 615. Table 56 describes the FILEEND identifiers.

```
FILEEND 0 1 6
        A B C
```

Table 56. 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 78 on page 615. Table 57 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 57. 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 57. 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 is 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.

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, 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 OR CONDITIONS OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

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 the z/OS operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming Interface Information

This publication documents *intended* Programming Interfaces that allow the customer to write z/OS C/C++ programs.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States or other countries or both:

AFP	AIX	AT
BookManager	BookMaster	C/370
C/MVS	CICS	CICS/ESA
CT	DB2	DB2 Universal Database

DFSMS	DFSMS/MVS	DRDA
GDDM	Hiperspace	IBM
IBMLink	IMS	IMS/ESA
Language Environment	Library Reader	MVS
MVS/DFP	MVS/ESA	Open Class
OpenEdition	OS/2	OS/390
OS/400	QMF	RACF
Resource Link	SOM	S/370
S/390	SP	System Object Model
VisualAge	VM/ESA	VSE/ESA
z/OS	zSeries	400

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the U.S. and/or other countries.

UNIX is a registered trademark of The Open Group in the U.S. and/or other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the U.S. and/or other countries.

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

Standards

Extracts are reprinted from IEEE Std 1003.1—1990, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C language], copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from IEEE P1003.1a Draft 6 July 1991, Draft Revision to Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language], copyright 1992 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from IEEE Std 1003.2—1992, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shells and Utilities, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from IEEE Std P1003.4a/D6—1992, IEEE Draft Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 2: Threads Extension [C language], copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

For more information on IEEE, visit their web site at `http://www.ieee.org/`.

Extracts from *ISO/IEC 9899:1990* have been reproduced with the permission of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). The complete standard can be obtained from any ISO or IEC member or from the ISO or IEC Central Offices, Case postale 56, CH - 1211 Geneva 20, Switzerland. Copyright remains ISO and IEC. For more information on ISO, visit their web site at `http://www.iso.ch/`.

Glossary

This glossary defines technical terms and abbreviations that are used in z/OS C/C++ documentation. If you do not find the term you are looking for, refer to the index of the appropriate z/OS C/C++ manual or view *IBM Glossary of Computing Terms*, located at:

`http://www.ibm.com/ibm/terminology/goc/gocmain.htm`

This glossary includes terms and definitions from:

- *American National Standard Dictionary for Information Systems*, ANSI/ISO X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI/ISO). Copies may be purchased from the American National Standards Institute, 11 West 42nd Street, New York, New York 10036. Definitions are indicated by the symbol *ANSI/ISO* after the definition.
- *IBM Dictionary of Computing*, SC20-1699. These definitions are indicated by the registered trademark *IBM* after the definition.
- *X/Open CAE Specification, Commands and Utilities, Issue 4. July, 1992*. These definitions are indicated by the symbol *X/Open* after the definition.
- *ISO/IEC 9945-1:1990/IEEE POSIX 1003.1-1990*. These definitions are indicated by the symbol *ISO.1* after the definition.
- *The Information Technology Vocabulary*, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol *ISO-JTC1* after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol *ISO Draft* after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

A

abstract class. (1) A class with at least one pure virtual function that is used as a base class for other classes. The abstract class represents a concept; classes derived from it represent implementations of the concept. You cannot create a direct object of an abstract class, but you can create references and pointers to an abstract class and set them to refer to objects of classes derived from the abstract class. See also *base class*. (2) A class that allows polymorphism. There can be no objects of an abstract class; they are only used to derive new classes.

abstract code unit. See *ACU*.

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.

abstraction (data). 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.

access. An attribute that determines whether or not a class member is accessible in an expression or declaration.

access declaration. A declaration used to restore access to members of a base class.

access mode. (1) A technique that is used to obtain a particular logical record from, or to place a particular logical record into, a file assigned to a mass storage device. *ANSI/ISO*. (2) The manner in which files are referred to by a computer. Access can be sequential (records are referred to one after another in the order in which they appear on the file), access can be random (the individual records can be referred to in a nonsequential manner), or access can be dynamic (records can be accessed sequentially or randomly, depending on the form of the input/output request). *IBM*. (3) A particular form of access permitted to a file. *X/Open*.

access resolution. The process by which the accessibility of a particular class member is determined.

access specifier. One of the C++ keywords: public, private, and protected, used to define the access to a member.

ACU (abstract code unit). A measurement used by the z/OS 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.

addressing mode. See *AMODE*.

address space. (1) The range of addresses available to a computer program. *ANSI/ISO*. (2) The complete range of addresses that are available to a programmer. See also *virtual address space*. (3) The area of virtual storage available for a particular job. (4) The memory locations that can be referenced by a process. *X/Open*. *ISO.1*.

aggregate. (1) An array or a structure. (2) A compile-time option to show the layout of a structure or union in the listing. (3) In programming languages, a structured collection of data items that form a data type. *ISO-JTC1*. (4) 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 sent to a management services focal point in a network to identify a problem or an impending problem. *IBM*. (2) To cause the user's terminal to give some audible or visual indication that an error or some other event has occurred. When the standard output is directed to a terminal device, the method for alerting the terminal user is unspecified. When the standard output is not directed to a terminal device, the alert is accomplished by writing the alert character to standard output (unless the utility description indicates that the use of standard output produces undefined results in this case). *X/Open*.

alert character. A character that in the output stream should cause a terminal to alert its user via 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. *X/Open*.

This character is named <alert> in the portable character set.

alias. (1) An alternate label; for example, a label and one or more aliases may be used to refer to the same data element or point in a computer program. *ANSI/ISO*. (2) An alternate name for a member of a partitioned data set. *IBM*. (3) An alternate name used for a network. Synonymous with nickname. *IBM*.

alias name. (1) A word consisting solely of underscores, digits, and alphabetics from the portable file name character set, and any of the following characters: `! % , @`. Implementations may allow other characters within alias names as an extension. *X/Open*. (2) An alternate name. *IBM*. (3) A name that is defined in one network to represent a logical unit name in another interconnected network. The alias name does not have to be the same as the real name; if these names are not the same; translation is required. *IBM*.

alignment. The storing of data in relation to certain machine-dependent boundaries. *IBM*.

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 Standard Code for Information Interchange (ASCII). The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. *IBM*.

Note: IBM has defined an extension to ASCII code (characters 128–255).

American National Standards Institute (ANSI/ISO). An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States. *ANSI/ISO*.

AMODE (addressing mode). In z/OS, a program attribute that refers to the address length that a program is prepared to handle upon entry. In z/OS, addresses may be 24 or 31 bits in length. *IBM*.

angle brackets. The characters < (left angle bracket) and > (right angle bracket). When used in the phrase “enclosed in angle brackets”, the symbol < immediately precedes the object to be enclosed, and > immediately follows it. When describing these characters in the portable character set, the names <less-than-sign> and <greater-than-sign> are used. *X/Open*.

anonymous union. A union that is declared within a structure or class and does not have a name. It must not be followed by a declarator.

ANSI/ISO. See *American National Standards Institute*.

API (application program interface). A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program. *IBM*.

application. (1) The use to which an information processing system is put; for example, a payroll application, an airline reservation application, a network application. *IBM*. (2) A collection of software components used to perform specific types of user-oriented work on a computer. *IBM*.

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. A program written for or by a user that applies to the user's work, such as a program that does inventory control or payroll. *IBM*.

archive libraries. The archive library file, when created for application program object files, has a special symbol table for members that are object files.

argument. (1) A parameter passed between a calling program and a called program. *IBM*. (2) In a function call, an expression that represents a value that the calling function passes to the function specified in the call. (3) In the shell, a parameter passed to a utility as the equivalent of a single string in the *argv* array created by one of the *exec* functions. An argument is one of the options, option-arguments, or operands following the command name. *X/Open*.

argument declaration. See *parameter declaration*.

arithmetic object. (1) A bit field, or an integral, floating-point, or packed decimal (IBM extension) object. (2) A real object or objects having the type float, double, or long double.

array. In programming languages, an aggregate that consists of data objects with identical attributes, each of which may be uniquely referenced by subscripting. *ISO-JTC1*.

array element. A data item in an array. *IBM*.

ASCII. See *American National Standard Code for Information Interchange.*

Assembler H. An IBM licensed program. Translates symbolic assembler language into binary machine language.

assembler language. A source language that includes symbolic language statements in which there is a one-to-one correspondence with the instruction formats and data formats of the computer. *IBM*.

assembler user exit. In the z/OS Language Environment 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. *IBM*.

atexit list. A list of actions specified in the z/OS C/C++ `atexit()` function that occur at normal program termination.

auto storage class specifier. A specifier that enables the programmer to define a variable with automatic storage; its scope restricted to the current block.

automatic call library. Contains 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
- z/OS C/C++ run-time routines (SCEELKED)

automatic library call. The process in which control sections are processed by the binder or loader to resolve references to members of partitioned data sets. *IBM*.

automatic storage. Storage that is allocated on entry to a routine or block and is freed on the subsequent return. Sometimes referred to as *stack storage* or *dynamic storage*.

B

background process. (1) A process that does not require operator intervention but can be run by the computer while the workstation is used to do other work. *IBM*. (2) A mode of program execution in which the shell does not wait for program completion before prompting the user for another command. *IBM*. (3) A process that is a member of a background process group. *X/Open*. *ISO.1*.

background process group. Any process group, other than a foreground process group, that is a member of a session that has established a connection with a controlling terminal. *X/Open*. *ISO.1*.

backslash. The character \. This character is named <backslash> in the portable character set.

base class. A class from which other classes are derived. A base class may itself be derived from another base class. See also *abstract class*.

based on. The use of existing classes for implementing new classes.

binary expression. An expression containing two operands and one operator.

binary stream. (1) An ordered sequence of untranslated characters. (2) 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. *IBM*.

bind. (1) To combine one or more control sections or program modules into a single program module, resolving references between them. (2) To assign virtual storage addresses to external symbols.

binder. The DFSMS/MVS program that processes the output of language translators and compilers into an executable program (load module or program object). It replaces the linkage editor and batch loader in the MVS/ESA, OS/390, or z/OS operating system.

bit field. A member of a structure or union that contains a specified number of bits. *IBM*.

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. *ANSI/ISO*. (2) A character that represents an empty position in a graphic character string. *ISO Draft*. (3) 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. *X/Open*.

block. (1) 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. *ISO-JTC1*. (2) A string of data elements recorded or transmitted as a unit. The elements may be characters, words or physical records. *ISO Draft*. (3) The unit of data transmitted to and from a device. Each block contains one record, part of a record, or several records.

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 or C++ statement. *IBM*.

boundary alignment. The position in main storage of a fixed-length field, such as a halfword or doubleword, on a byte-level boundary for that unit of information. *IBM*.

braces. The characters { (left brace) and } (right brace), also known as *curly braces*. When used in the phrase “enclosed in (curly) braces” the symbol { immediately precedes the object to be enclosed, and } immediately follows it. When describing these characters in the portable character set, the names <left-brace> and <right-brace> are used. *X/Open*.

brackets. The characters [(left bracket) and] (right bracket), also known as *square brackets*. When used in the phrase *enclosed in (square) brackets* the symbol [immediately precedes the object to be enclosed, and] immediately follows it. When describing these characters in the portable character set, the names <left-bracket> and <right-bracket> are used. *X/Open*.

break statement. A C or C++ control statement that contains the keyword “`break`” and a semicolon. *IBM*. 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. (1) A function that the compiler will automatically inline instead of making the function call, unless the programmer specifies not to inline. (2) In programming languages, pertaining to a language object that is declared by the definition of the programming language; for example, the built-in function SIN in PL/I, the predefined data type INTEGER in FORTRAN. *ISO-JTC1*. Synonymous with *predefined*. *IBM*.

byte-oriented stream. See *orientation of a stream*.

C

C library. A system library that contains common C language subroutines for file access, string operators, character operations, memory allocation, and other functions. *IBM*.

C or C++ language statement. A C or C++ language statement contains zero or more expressions. A block statement begins with a { (left brace) symbol, ends with a } (right brace) symbol, and contains any number of statements.

All C or C++ language statements, except block statements, end with a ; (semicolon) symbol.

c89 utility. A utility used to compile and bind an application program from the z/OS shell.

C++ class library. A collection of C++ classes.

C++ library. A system library that contains common C++ language subroutines for file access, memory allocation, and other functions.

callable services. A set of services that can be invoked by z/OS Language Environment-conforming high level languages using the conventional z/OS Language Environment-defined call interface, and usable by all programs sharing the z/OS Language Environment conventions.

Use of these services helps to decrease an application's dependence on the specific form and content of the services delivered by any single operating system.

call chain. A trace of all active functions.

caller. A function that calls another function.

cancelability point. A specific point within the current thread that is enabled to solicit cancel requests. This is accomplished using the `pthread_testintr()` function.

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. The carriage-return is the character designated by '\r' 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 movement to the beginning of the line. *X/Open*.

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 integral expression and a colon. When the selector evaluates 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. The cast operator is used for explicit type conversions.

cataloged procedures. A set of control statements placed in a library and retrievable by name. *IBM*.

catch block. A block associated with a try block that receives control when an exception matching its argument is thrown.

char specifier. A char is a built-in data type. In the C++ language, char, signed char, and unsigned char are all distinct data types.

character. (1) A letter, digit, or other symbol that is used as part of the organization, control, or representation of data. A character is often in the form of a spatial arrangement of adjacent or connected strokes. *ANSI/ISO*. (2) A sequence of one or more bytes representing a single graphic symbol or control code. This term corresponds to the ISO C standard term *multibyte character* (multibyte character), where a single-byte character is a special case of the multibyte character. Unlike the usage in the ISO C standard, *character* here has no necessary relationship with storage space, and *byte* is used when storage space is discussed. *X/Open*. *ISO.1*.

character array. An array of type char. *X/Open*.

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. *X/Open*.

character constant. A string of any of the characters that can be represented, usually enclosed in quotes.

character set. (1) A finite set of different characters that is complete for a given purpose; for example, the character set in ISO Standard 646, 7-bit Coded Character Set for Information Processing Interchange. *ISO Draft*. (2) All the valid characters for a programming language or for a computer system. *IBM*. (3) A group of characters used for a specific reason; for example, the set of characters a printer can print. *IBM*. (4) See also *portable character set*.

character special file. (1) A special file that provides access to an input or output device. The character interface is used for devices that do not use block I/O. *IBM*. (2) A file that refers to a device. One specific type of character special file is a terminal device file. *X/Open*. *ISO.1*.

character string. A contiguous sequence of characters terminated by and including the first null byte. *X/Open*.

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*.

CICS (Customer Information Control System). Pertaining to an IBM licensed program that enables transactions entered at remote terminals to be processed concurrently by user-written application programs. It includes facilities for building, using, and maintaining databases. *IBM*.

CICS destination control table. See *DCT*.

CICS translator. A routine that accepts as input an application containing EXEC CICS commands and produces as output an equivalent application in which each CICS command has been translated into the language of the source.

class. (1) A C++ aggregate that may contain functions, types, and user-defined operators in addition to data. A class may be derived from another class, inheriting the properties of its parent class. A class may restrict access to its members. (2) 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. A collection of classes.

class member operator. An operator used to access class members through class objects or pointers to class objects. The class member operators are:

```
.  ->  .*  ->*
```

class name. A unique identifier that names a class type.

class scope. An indication that a name of a class can be used only in a member function of that class.

class tag. Synonym for *class name*.

class template. A blueprint describing how a set of related 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 class template definition 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.

client program. A program that uses a class. The program is said to be a *client* of the class.

CLIST. A programming language that typically executes a list of TSO commands.

CLLE (COBOL Load List Entry). Entry in the load list containing the name of the program and the load address.

COBCOM. Control block containing information about a COBOL partition.

COBOL (common business-oriented language). A high-level language, based on English, that is primarily used for business applications.

COBOL Load List Entry. See *CLLE*.

COBVEC. COBOL vector table containing the address of the library routines.

coded character set. (1) A set of graphic characters and their code point assignments. The set may contain fewer characters than the total number of possible characters: some code points may be unassigned. *IBM*. (2) A coded set whose elements are single characters; for example, all characters of an alphabet. *ISO Draft*. (3) Loosely, a code. *ANSI/ISO*.

code element set. (1) The result of applying a code to all elements of a coded set, for example, all the three-letter international representations of airport names. *ISO Draft*. (2) 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*. (3) Synonym for codeset.

code page. (1) An assignment of graphic characters and control function meanings to all code points; for example, assignment of characters and meanings to 256 code points for an 8-bit code, assignment of characters and meanings to 128 code points for a 7-bit code. (2) A particular assignment of hexadecimal identifiers to graphic characters.

code point. (1) A representation of a unique character. For example, in a single-byte character set each of 256 possible characters is represented by a one-byte code point. (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.

codeset. Synonym for code element set. *IBM*.

collating element. The smallest entity used to determine the logical ordering of character or wide-character 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. *X/Open*.

collating sequence. (1) A specified arrangement used in sequencing. *ISO-JTC1*. *ANSI/ISO*. (2) An ordering assigned to a set of items, such that any two sets in that assigned order can be collated. *ANSI/ISO*. (3) 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. This is the order used in ranges of characters and collating elements in regular expressions and pattern matching. In addition, the definition of the collating weights of characters and collating elements uses collating elements to represent their respective positions within the collation sequence.

collation. The logical ordering of character or wide-character strings according to defined precedence rules. These rules identify a collation sequence between the collating elements, and such additional rules that can be used to order strings consisting or multiple collating elements. *X/Open*.

collection. (1) An abstract class without any ordering, element properties, or key properties. (2) In a general sense, an implementation of an abstract data type for storing elements.

Collection Class Library. A set of classes that provide basic functions for collections, and can be used as base classes.

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 (not a function argument list) that contains two or more operands separated by commas. The compiler evaluates all operands in the order specified, discarding all but the last (rightmost). The value of the expression is the value of the rightmost operand. Typically this is done to produce side effects.

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 processor parameter list (CPPL). The format of a TSO parameter list. When a TSO terminal monitor application attaches a command processor, register 1 contains a pointer to the CPPL, containing addresses required by the command processor.

COMMAREA. A communication area made available to applications running under CICS.

Common Business-Oriented Language. See *COBOL*.

common expression elimination. Duplicated expressions are eliminated by using the result of the previous expression. This includes intermediate expressions within expressions.

compilation unit. (1) A portion of a computer program sufficiently complete to be compiled correctly. *IBM*. (2) A single compiled file and all its associated include files. (3) An independently compilable sequence of high-level language statements. Each high-level language product has different rules for what makes up a compilation unit.

complete class name. The complete qualification of a nested class name including all enclosing class names.

Complex Mathematics library. A C++ class library that provides the facilities to manipulate complex numbers and perform standard mathematical operations on them.

computational independence. No data modified by either a main task program or a parallel function is examined or modified by a parallel function that might be running simultaneously.

concrete class. (1) A class that is not abstract. (2) A class defining objects that can be created.

condition. (1) A relational expression that can be evaluated to a value of either true or false. *IBM*. (2) An exception that has been enabled, or recognized, by the z/OS Language Environment and thus is eligible to activate user and language condition handlers. Any alteration to the normal programmed flow of an application. 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.

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).

condition handler. A user-written condition handler or language-specific condition handler (such as a PL/I ON-unit or z/OS C/C++ `signal()` function call) invoked by the z/OS C/C++ *condition manager* to respond to conditions.

condition manager. Manages conditions in the common execution environment by invoking various user-written and language-specific *condition handlers*.

condition token. In the z/OS Language Environment, a data type consisting of 12 bytes (96 bits). 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.

const. (1) An attribute of a data object that declares the object cannot be changed. (2) A keyword that allows you to define a variable whose value does not change. (3) A keyword that allows you to define a parameter that is not changed by the function. (4) A keyword that allows you to define a member function that does not modify the state of the class for which it is defined.

constant. (1) In programming languages, a language object that takes only one specific value. *ISO-JTC1*. (2) A data item with a value that does not change. *IBM*.

constant expression. An expression having a value that can be determined during compilation and that does not change during the running of the program. *IBM*.

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. Contrast with *natural reentrancy*.

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. (1) A character whose occurrence in a particular context specifies a control function. *ISO Draft*. (2) Synonymous with non-printing character. *IBM*. (3) A character, other than a graphic character, that affects the recording, processing, transmission, or interpretation of text. *X/Open*.

control statement. (1) A statement that is used to alter the continuous sequential execution of statements; a control statement may be a conditional statement, such as `if`, or an imperative statement, such as `return`. (2) A statement that changes the path of execution.

controlling process. The session leader that establishes the connection to the controlling terminal. If the terminal ceases to be a controlling terminal for this session, the session leader ceases to be the controlling process. *X/Open*. *ISO.1*.

controlling terminal. A terminal that is associated with a session. Each session may have at most one controlling terminal associated with it, and a controlling terminal is associated with exactly one session. Certain input sequences from the controlling terminal cause signals to be sent to all processes in the process group associated with the controlling terminal. *X/Open*. *ISO.1*.

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. *ISO-JTC1*. (2) The process of changing from one method of data processing to another or from one data processing system to another. *IBM*. (3) The process of changing from one form of representation to another; for example to change from decimal representation to binary representation. *IBM*. (4) A change in the type of a value. For example, when you add values having different data types, the compiler converts both values to a common form before adding the values.

conversion descriptor. A per-process unique value used to identify an open codeset conversion. *X/Open*.

conversion function. A member function that specifies a conversion from its class type to another type.

coordinated universal time (UTC). Synonym for Greenwich Mean Time (GMT). See *GMT*.

copy constructor. A constructor that copies a class object of the same class type.

CSECT (control section). 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.

Cross System Product. See *CSP*.

CSP (Cross System Product). A set of licensed programs designed to permit the user to develop and run applications using independently defined maps (display and printer formats), data items (records, working storage, files, and single items), and processes (logic). The Cross System Product set consists of two parts: Cross System Product/Application Development (CSP/AD) and Cross System Product/Application Execution (CSP/AE). *IBM*.

current working directory. (1) A directory, associated with a process, that is used in path name resolution for path names that do not begin with a slash. *X/Open*. *ISO.1*. (2) In the OS/2 operating system, the first directory in which the operating system looks for programs and files and stores temporary files and output. *IBM*. (3) In the z/OS UNIX environment, a directory that is active and that can be displayed. Relative path name resolution begins in the current directory. *IBM*.

cursor. A reference to an element at a specific position in a data structure.

Customer Information Control System. See *CICS*.

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). (1) In the C and C++ languages, a definition that describes a data object, reserves storage for a data object, and can provide an initial value for a data object. A data definition appears outside a function or at the beginning of a block statement. *IBM*. (2) A program statement that describes the features of, specifies relationships of, or establishes context of, data. *ANSI/ISO*. (3) A statement that is stored in the environment and that externally identifies a file and the attributes with which it should be opened.

data definition name. See *ddname*.

data definition statement. See *DD statement*.

data member. The smallest possible piece of complete data. Elements are composed of data members.

data object. (1) A storage area used to hold a value. (2) Anything that exists in storage and on which operations can be performed, such as files, programs, classes, or arrays. (3) In a program, an element of data structure, such as a file, array, or operand, that is needed for the execution of a program and that is named or otherwise specified by the allowable character set of the language in which a program is coded. *IBM*.

data set. Under z/OS, a named collection of related data records that is stored and retrieved by an assigned name.

data stream. A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format. *IBM*.

data structure. The internal data representation of an implementation.

data type. The properties and internal representation that characterize data.

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 (double-byte character set). A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets.

Because each character requires 2 bytes, the typing, display, and printing of DBCS characters requires hardware and programs that support DBCS. *IBM*.

DCT (destination control table). A table that contains an entry for each extrapartition, intrapartition, and indirect destination. Extrapartition entries address data sets external to the CICS region. Intrapartition destination entries contain the information required to locate the queue in the intrapartition data set. Indirect destination entries contain the information required to locate the queue in the intrapartition data set.

ddname (data definition name). (1) The logical name of a file within an application. The ddname provides the means for the logical file to be connected to the physical file. (2) The part of the data definition before the equal sign. It is the name used in a call to `fopen` or `freopen` to refer to the data definition stored in the environment.

DD statement (data definition statement). (1) In z/OS, serves as the connection between the logical name of a file and the physical name of the file. (2) A job control statement that defines a file to the operating system, and is a request to the operating system for the allocation of input/output resources.

dead code elimination. A process that eliminates code that exists for calculations that are not necessary. Code may be designated as dead by other optimization techniques.

dead store elimination. A process that eliminates unnecessary storage use in code. A store is deemed unnecessary if the value stored is never referenced again in the code.

decimal constant. (1) A numerical data type used in standard arithmetic operations. (2) A number containing any of the digits 0 through 9. *IBM*.

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) In the C and C++ languages, a description that makes an external object or function available to a function or a block statement. *IBM*. (2) Establishes the names and characteristics of data objects and functions used in a program.

declarator. Designates a data object or function declared. Initializations can be performed in a declarator.

default argument. An argument that is declared with a default value in a function prototype or declaration. If a call to the function omits this argument, the default value is used. Arguments with default values must be the trailing arguments 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. *IBM*.

default constructor. A constructor that takes no arguments, or, if it takes 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.

default locale. (1) The C locale, which is always used when no selection of locale is performed. (2) A system default locale, named by locale-related environmental variables.

define directive. A preprocessor directive that directs the preprocessor to replace an identifier or macro invocation with special code.

definition. (1) A data description that reserves storage and may provide an initial value. (2) A declaration that allocates storage, and may initialize a data object or specify the body of a function.

degree. The number of children of a node.

delete. (1) A C++ keyword that identifies a free storage deallocation operator. (2) A C++ operator used to destroy objects created by `new`.

demangling. The conversion of mangled names back to their original source code names. During C++

compilation, identifiers such as function and static class member names are mangled (encoded) with type and scoping information to ensure type-safe linkage. These mangled names appear in the object file and the final executable file. Demangling (decoding) converts these names back to their original names to make program debugging easier. See also *mangling*.

deque. A queue that can have elements added and removed at both ends. A double-ended queue.

dequeue. An operation that removes the first element of a queue.

dereference. In the C and C++ languages, the application of the unary operator * to a pointer to access the object the pointer points to. Also known as *indirection*.

derivation. In the C++ language, to derive a class, called a derived class, from an existing class, called a base class.

derived class. A class that inherits from a base class. All members of the base class become members of the derived class. You can add additional data members and member functions to the derived class. A derived class object can be manipulated as if it is a base class object. The derived class can override virtual functions of the base class.

descriptor. 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. See *DCT*.

destructor. A special member function that has the same name as its class, preceded by a tilde (~), and that "cleans up" after an object of that class, for example, freeing storage that was allocated when the object was created. A destructor has no arguments and no return type.

detach state attribute. An attribute associated with a thread attribute object. This attribute has two possible values:

0 Undetached. An undetached thread keeps its resources after termination of the thread.

1 Detached. A detached thread has its resources freed by the system after termination.

device. A computer peripheral or an object that appears to the application as such. *X/Open*. *ISO.1*.

difference. For two sets A and B, the difference (A-B) is the set of all elements in A but not in B. For bags, there is an additional rule for duplicates: If bag P contains an element *m* times and bag Q contains the same element *n* times, then, if *m>n*, the difference contains that element *m-n* times. If *m≤n*, the difference contains that element zero times.

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.

directory. (1) In a hierarchical file system, a container for files or other directories. *IBM*. (2) The part of a partitioned data set that describes the members in the data set.

disabled signal. Synonym for *enabled signal*.

display. To direct the output to the user's terminal. If the output is not directed to the terminal, the results are undefined. *X/Open*.

DLL. See *dynamic link library*.

do statement. In 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). *IBM*.

dot. The file name consisting of a single dot character (.). *X/Open*. *ISO.1*.

double-byte character set. See *DBCS*.

double-precision. Pertaining to the use of two computer words to represent a number in accordance with the required precision. *ISO-JTC1*. *ANSI/ISO*.

double-quote. The character ", also known as *quotation mark*. *X/Open*.

This character is named <quotation-mark> in the portable character set.

doubleword. A contiguous sequence of bytes or characters that comprises two computer words and is capable of being addressed as a unit. *IBM*.

dynamic. Pertaining to an operation that occurs at the time it is needed rather than at a predetermined or fixed time. *IBM*.

dynamic allocation. Assignment of system resources to a program when the program is executed rather than when it is loaded into main storage. *IBM*.

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 run time. The code and data in a dynamic link library can be shared by several applications simultaneously. Compiling code with the `DLL` option does not mean that

the produced executable will be a DLL. To create a DLL, use `#pragma export` or the `EXPORTALL` compiler option.

DSA (dynamic storage area). An area of storage obtained during the running of an application that consists of a register save area and an area for automatic data, such as program variables. DSAs are generally allocated within Language Environment-managed stack segments. DSAs are added to the stack when a routine is entered and removed upon exit in a last in, first out (LIFO) manner. In Language Environment, a DSA is known as a stack frame.

dynamic storage. Synonym for *automatic storage*.

dynamic storage area . See DSA

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, as described in the *exec* family of functions and `setgid()`. *X/Open*. *ISO.1*.

effective user ID. (1) The user ID associated with the last authenticated user or the last `setuid()` program. It is equal to either the real or the saved user ID. (2) The current user ID, but not necessarily the user's login ID; for example, a user logged in under a login ID may change to another user's ID. The ID to which the user changes becomes the effective user ID until the user switches back to the original login ID. All discretionary access decisions are based on the effective user ID. *IBM*. (3) 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, as described in *exec* and `setuid()`. *X/Open*. *ISO.1*.

elaborated type specifier. A specifier typically used in an incomplete class declaration to qualify types that are otherwise hidden.

element. The component of an array, subrange, enumeration, or set.

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 nonunique 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 word else, followed by a statement. The else clause provides an action that is started when the if condition evaluates to a value of zero (false). *IBM*.

empty line. A line consisting of only a new-line character. *X/Open*.

empty string. (1) A string whose first byte is a null byte. Synonymous with null string. *X/Open*. (2) A character array whose first element is a null character. *ISO.1*.

enabled signal. The occurrence of an enabled signal results in the default system response or the execution of an established signal handler. If disabled, the occurrence of the signal is ignored.

encapsulation. Hiding the internal representation of data objects and implementation details of functions from the client program. This enables the end user to focus on the use of data objects and functions without having to know about their representation or implementation.

enclave. In z/OS Language Environment, an independent collection of routines, one of which is designated as the main routine. An enclave is roughly analogous to a program or run unit.

enqueue. (1) An operation that adds an element as the last element to a queue. (2) Request control of a serially reusable resource.

entry point. The address or label of the first instruction that is executed when a routine is entered for execution.

enumeration 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. *IBM*.

enumeration data type. (1) In the Fortran, C, and C++ language, a data type that represents a set of values that a user defines. *IBM*. (2) A type that represents integers and a set of enumeration constants. Each enumeration constant has an associated integer value.

enumeration tag. In the C and C++ language, the identifier that names an enumeration data type. *IBM*.

enumeration type. An enumeration type 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. In the C and C++ language, an enumeration constant and its associated value. *IBM*.

equivalence class. (1) A grouping of characters that are considered equal for the purpose 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. *IBM*. (2) A set of collating elements with the same primary collation weight.

Elements in an equivalence class are typically elements that naturally group together, such as all accented letters based on the same base letter.

The collation order of elements within an equivalence class is determined by the weights assigned on any subsequent levels after the primary weight. *X/Open*.

escape sequence. (1) A representation of a character. An escape sequence contains the \ symbol followed by one of the characters: `a`, `b`, `f`, `n`, `r`, `t`, `v`, `'`, `"`, `x`, `\`, or followed by one or more octal or hexadecimal digits. (2) A sequence of characters that represent, for example, non-printing characters, or the exact code point value to be used to represent variant and nonvariant characters regardless of code page. (3) In the C and C++ language, an escape character followed by one or more characters. The escape character indicates that a different code, or a different coded character set, is used to interpret the characters that follow. Any member of the character set used at run time can be represented using an escape sequence. (4) A character that is preceded by a backslash character and is interpreted to have a special meaning to the operating system. (5) A sequence sent to a terminal to perform actions such as moving the cursor, changing from normal to reverse video, and clearing the screen. Synonymous with multibyte control. *IBM*.

exception. (1) Any user, logic, or system error detected by a function that does not itself deal with the error but passes the error on to a handling routine (also called throwing the exception). (2) In programming languages, an abnormal situation that may arise during execution, that may cause a deviation from the normal execution sequence, and for which facilities exist in a programming language to define, raise, recognize, ignore, and handle it; for example, (ON-) condition in PL/I, exception in ADA. *ISO-JTC1*.

executable. A load module or program object which has yet to be loaded into memory for execution.

executable file. A regular file acceptable as a new process image file by the equivalent of the *exec* family of functions, and thus usable as one form of a utility. The standard utilities described as compilers can produce executable files, but other unspecified methods of producing executable files may also be provided. The internal format of an executable file is unspecified, but a conforming application cannot assume an executable file is a text file. *X/Open*.

exception handler. (1) Exception handlers are catch blocks in C++ applications. Catch blocks catch exceptions when they are thrown from a function enclosed in a try block. Try blocks, catch blocks, and throw expressions are the constructs used to implement formal exception handling in C++ applications. (2) A set of routines used to detect deadlock conditions or to process abnormal condition processing. An exception handler allows the normal running of processes to be interrupted and resumed. *IBM*.

executable file. A regular file acceptable as a new process image file by the equivalent of the *exec* family of functions, and thus usable as one form of a utility. The standard utilities described as compilers can produce executable files, but other unspecified methods of producing executable files may also be provided. The internal format of an executable file is unspecified, but a conforming application cannot assume an executable file is a text file. *X/Open*.

executable program. A program that has been link-edited and therefore can be run in a processor. *IBM*.

extended binary-coded data interchange code (EBCDIC). A coded character set of 256 8-bit characters. *IBM*.

extended-precision. Pertaining to the use of more than two computer words to represent a floating point number in accordance with the required precision. In z/OS four computer words are used for an extended-precision number.

extension. (1) An element or function not included in the standard language. (2) File name extension.

external data definition. A description of a variable appearing outside a function. It causes the system to allocate storage for that variable and makes that variable accessible to all functions that follow the definition and are located in the same file as the definition. *IBM*.

extern storage class specifier. A specifier that enables the programmer to declare objects and functions that several source files can use.

F

feature test macro (FTM). A macro (`#define`) used to determine whether a particular set of features will be included from a header. *X/Open*. *ISO.1*.

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 basis. Other characteristics of FIFOs are described in `open()`, `read()`, `write()`, and `lseek()`. *X/Open*. *ISO.1*.

file access permissions. The standard file access control mechanism uses the file permission bits. The

bits are set at the time of file creation by functions such as `open()`, `creat()`, `mkdir()`, and `mkfifo()` and can be changed by `chmod()`. The bits are read by `stat()` or `fstat()`. *X/Open*.

file descriptor. (1) A positive integer that the system uses instead of the file name to identify an open file. (2) A per-process unique, non-negative integer used to identify an open file for the purpose of file access. *ISO.1*.

The value of a file descriptor is from zero to {OPEN_MAX}—which is defined in <limits.h>. A process can have no more than {OPEN_MAX} file descriptors open simultaneously. File descriptors may also be used to implement directory streams. *X/Open*.

file mode. An object containing the *file mode bits* and file type of a file, as described in `<sys/stat.h>`. *X/Open*.

file mode bits. A file's file permission bits, set-user-ID-on-execution bit (S_ISUID) and set-group-ID-on-execution bit (S_ISGID). *X/Open*.

file permission bits. Information about a file that is used, along with other information, to determine if a process has read, write, or execute/search permission to a file. The bits are divided into three parts: owner, group, and other. Each part is used with the corresponding file class of process. These bits are contained in the file mode, as described in *<sys/stat.h>.* The detailed usage of the file permission bits is described in *file access permissions*. *X/Open*. *ISO.1*.

file scope. A name declared outside all blocks, classes, and function declarations has file scope and can be used after the point of declaration in a source file.

filter. A command whose operation consists of reading data from standard input or a list of input files and writing data to standard output. Typically, its function is to perform some transformation on the data stream. *X/Open*.

first element. The element visited first in an iteration over a collection. Each collection has its own definition for first element. For example, the first element of a sorted set is the element with the smallest value.

flat collection. A collection that has no hierarchical structure.

float constant. (1) A constant representing a nonintegral 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). *IBM*.

for statement. A looping statement that contains the word *for* followed by a for-initializing-statement, an optional condition, a semicolon, and an optional expression, all enclosed in parentheses.

foreground process. (1) A process that must run to completion before another command is issued. The foreground process is in the foreground process group, which is the group that receives the signals generated by a terminal. *IBM*. (2) A process that is a member of a foreground process group. *X/Open*. *ISO.1*.

foreground process group. (1) The group that receives the signals generated by a terminal. *IBM*. (2) A process group whose member processes have certain privileges, denied to processes in background process groups, when accessing their controlling terminal. Each session that has established a connection with a controlling terminal has exactly one process group of the session as the foreground process group of that controlling terminal. *X/Open*. *ISO.1*.

foreground process group ID. The process group ID of the foreground process group. *X/Open*. *ISO.1*.

form-feed character. A character in the output stream that indicates that printing should start on the next page of an output device. The formfeed is the character designated by '\f' in the C and C++ language. If the formfeed is not the first character of an output line, the result is unspecified. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next page. *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.

freestanding application. (1) An application that is created to run without the run-time environment or library with which it was developed. (2) An 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.

free store. Dynamically allocated memory. New and delete are used to allocate and deallocate free store.

friend class. A class in which all the 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. For example, the following source code makes all the functions and data in class `you` friends of class `me`:

```
class me {
   friend class you;
     // ...
};
```

friend function. A function that is granted access to the private and protected parts of a class. It is named in the declaration of the other class with the prefix `friend`.

function. A named group of statements that can be called and evaluated and can return a value to the calling statement. *IBM*.

function call. An expression that moves the path of execution from the current function to a specified function and evaluates to the return value provided by the called function. A function call contains the name of the function to which control moves and a parenthesized list of values. *IBM*.

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. *IBM*.

function definition. The complete description of a function. A function definition contains a sequence of specifiers (storage class, optional type, inline, virtual, optional friend), a function declarator, optional constructor-initializers, parameter declarations, optional const, and the block statement. Inline, virtual, friend, and const are not available with C.

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. Provides a blueprint describing how a set of related individual functions can be constructed.

G

Generalization. Refers to a class, function, or static data member which derives its definition from a template. An instantiation of a template function would be a generalization.

Generalized Object File Format (GOFF). It is the strategic object module format for S/390. It extends the capabilities of object modules to contain more information than current object modules. It removes the limitations of the previous object module format and supports future enhancements. It is required for XPLINK.

generic class. Synonym for *class templates*.

global. Pertaining to information available to more than one program or subroutine. *IBM*.

global scope. Synonym for *file scope*.

global variable. A symbol defined in one program module that is used in other independently compiled program modules.

GMT (Greenwich Mean Time). The solar time at the meridian of Greenwich, formerly used as the prime basis of standard time throughout the world. GMT has been superseded by coordinated universal time (UTC).

graphic character. (1) A visual representation of a character, other than a control character, that is normally produced by writing, printing, or displaying. *ISO Draft*. (2) A character that can be displayed or printed. *IBM*.

Graphical Data Display Manager (GDDM). Pertaining to an IBM licensed program that provides a group of routines that allows pictures to be defined and displayed procedurally through function routines that correspond to graphic primitives. *IBM*.

Greenwich Mean Time. See GMT.

group ID. (1) A non-negative integer that is used to identify a group of system users. Each system user is a member of at least one group. When the identity of a group is associated with a process, a group ID value is referred to as a real group ID, an effective group ID, one of the supplementary group IDs or a saved set-group-ID. *X/Open*. (2) A non-negative integer, which can be contained in an object of type *gid_t*, that is used to identify a group of system users. *ISO.1*.

H

halfword. A contiguous sequence of bytes or characters that constitutes half a computer word and can be addressed as a unit. *IBM*.

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 (that hash value). Using a hash table minimizes average search time.

header file. A text file that contains declarations used by a group of functions, programs, or users.

heap storage. An area of storage used for allocation of storage whose lifetime 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. Three examples for the hexadecimal constant with value 0 would be '\x00', '0x0', or '0X00'.

High Level Assembler. An IBM licensed program. Translates symbolic assembler language into binary machine language.

hiperspace memory file. An IBM file used under z/OS to deal with memory files as large as 2 gigabytes. *IBM*.

hooks. Instructions inserted into a program by a compiler at compile-time. Using hooks, you can set break-points to instruct the Debug Tool to gain control of the program at selected points during its execution.

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

I18N. Abbreviation for *internationalization*.

identifier. (1) One or more characters used to identify or name a data element and possibly to indicate certain properties of that data element. *ANSI/ISO*. (2) In programming languages, a token that names a data object such as a variable, an array, a record, a subprogram, or a function. *ANSI/ISO*. (3) A sequence of letters, digits, and underscores used to identify a data object or function. *IBM*.

if statement. A conditional statement that contains the keyword if, followed by an expression in parentheses (the condition), a statement (the action), and an optional else clause (the alternative action). *IBM*.

ILC (interlanguage call). A function call made by one language to a function coded in another language. Interlanguage calls are used to communicate between programs written in different languages.

ILC (interlanguage communication). 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.

implementation-defined behavior. Application behavior that is not defined by the standards. The implementing compiler and library defines this behavior when a program contains correct program constructs or uses correct data. Programs that rely on implementation-defined behavior may behave differently on different C or C++ implementations. Refer to the z/OS C/C++ books that are listed in “z/OS C/C++ and Related Publications” on page 4 for information about implementation-defined behavior in the z/OS C/C++ environment. Contrast with *unspecified behavior* and *undefined behavior*.

IMS (Information Management System). Pertaining to an IBM database/data communication (DB/DC) system that can manage complex databases and networks. *IBM*.

include directive. A preprocessor directive that causes the preprocessor to replace the statement with the contents of a specified file.

include file. See *header file*.

incomplete class declaration. A class declaration that does not define any members of a class. Until a class is fully declared, or defined, you can only use the class name where the size of the class is not required. Typically an incomplete class declaration is used as a forward declaration.

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. A void type can never be completed. Arrays of unknown size and structures or unions of unspecified content can be completed in further declarations.

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. Synonym for *reference class*.

induction variable. It is a controlling variable of a loop.

inheritance. A technique that allows the use of an existing class as the base for creating other classes.

initial heap. The z/OS C/C++ heap controlled by the HEAP run-time option and designated by a `heap_id` of 0. The initial heap contains dynamically allocated user data.

initializer. An expression used to initialize data objects. The C++ language, supports the following types of initializers:

- An expression followed by an assignment operator that is used to initialize fundamental data type objects or class objects that contain copy constructors.
- A parenthesized expression list that is used to initialize base classes and members that use constructors.

Both the C and C++ languages support an expression enclosed in braces ({ }), that used to initialize aggregates.

inlined 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 a system from an input unit. Synonymous with input job stream, job input stream. *IBM*.

instance. An object-oriented programming term synonymous with object. An instance is a particular instantiation of a data type. It is simply a region of storage that contains a value or group of values. For example, if a class `box` is previously defined, two instances of a class `box` could be instantiated with the declaration: `box box1, box2;`

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.

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. See *ISPF*.

interlanguage call. See *ILC (interlanguage call)*.

interlanguage communication. See *ILC (interlanguage communication)*.

internationalization. The capability of a computer program to adapt to the requirements of different native languages, local customs, and coded character sets. *X/Open*.

Synonymous with *I18N*.

interoperability. The capability to communicate, execute programs, or transfer data among various functional units in a way that requires the user to have little or no knowledge of the unique characteristics of those units.

Interprocedural Analysis. See *IPA*.

interprocess communication. (1) The exchange of information between processes or threads through semaphores, queues, and shared memory. (2) The process by which programs communicate data to each other to synchronize their activities. Semaphores, signals, and internal message queues are common methods of inter-process communication.

I/O Stream library. A class library that provides the facilities to deal with many varieties of input and output.

IPA (Interprocedural Analysis). A process for performing optimizations across compilation units.

ISPF (Interactive System Productivity Facility). An IBM licensed program that serves as a full-screen editor and dialogue manager. Used for writing application programs, it provides a means of generating standard screen panels and interactive dialogues between the application programmer and terminal user. (ISPF)

iteration. The process of repeatedly applying a function to a series of elements in a collection until some condition is satisfied.

J

JCL (job control language). A control language used to identify a job to an operating system and to describe the job's requirement. *IBM*.

K

keyword. (1) A predefined word reserved for the C and C++ languages, that may not be used as an identifier. (2) A symbol that identifies a parameter in JCL.

kind attribute. An attribute for a mutex attribute object. This attribute's value determines whether the mutex can be locked once or more than once for a thread and whether state changes to the mutex will be reported to the debug interface.

L

label. An identifier within or attached to a set of data elements. *ISO Draft*.

Language Environment. Abbreviated form of z/OS Language Environment. Pertaining to an IBM software product that provides a common run-time environment and run-time services to applications compiled by Language Environment-conforming compilers.

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.

late binding. Allowing the system to determine the specific class of the object and invoke the appropriate function implementations at run time. Late binding or dynamic binding hides the differences between a group of related classes from the application program.

leaves. Nodes without children. Synonymous with terminals.

lexically. Relating to the left-to-right order of units.

library. (1) A collection of functions, calls, subroutines, or other data. *IBM*. (2) A set of object modules that can be specified in a link command.

linkage editor. Synonym for linker. The linkage editor has been replaced by the *binder* for the MVS/ESA, OS/390, or z/OS operating systems. See *binder*.

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.

linker. A computer program for creating load modules from one or more object modules by resolving cross references among the modules and, if necessary, adjusting addresses. *IBM*.

link pack area (LPA). In z/OS, an area of storage containing re-enterable routines from system libraries. Their presence in main storage saves loading time.

literal. (1) In programming languages, a lexical unit that directly represents a value; for example, 14 represents the integer fourteen, “APRIL” represents the string of characters APRIL, 3.0005E2 represents the number 300.05. *ISO-JTC1*. (2) A symbol or a quantity in a source program that is itself data, rather than a reference to data. *IBM*. (3) A character string whose value is given by the characters themselves; for example, the numeric literal 7 has the value 7, and the character literal CHARACTERS has the value CHARACTERS. *IBM*.

loader. A routine, commonly a computer program, that reads data into main storage. *ANSI/ISO*.

load module. All or part of a computer program in a form suitable for loading into main storage for execution. A load module is usually the output of a linkage editor. *ISO Draft*.

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. *ISO-JTC1*. (2) Pertaining to that which is defined and used only in one subdivision of a computer program. *ANSI/ISO*.

local customs. The conventions of a geographical area or territory for such things as date, time, and currency formats. *X/Open*.

locale. The definition of the subset of a user's environment that depends on language and cultural conventions. *X/Open*.

localization. The process of establishing information within a computer system specific to the operation of particular native languages, local customs, and coded character sets. *X/Open*.

local scope. A name declared in a block has scope within the block, and can therefore only be used in that block.

Long name. An external name C++ name in an object module, or and external name in an object module created by the C compiler when the `LONGNAME` option is used. Long names are up to 1024 characters long and may contain both upper-case and lower-case characters.

lvalue. An expression that represents a data object that can be both examined and altered.

M

macro. An identifier followed by arguments (may be a parenthesized list of arguments) that the preprocessor replaces with the replacement code located in a preprocessor `#define` directive.

macro call. Synonym for *macro*.

macro instruction. Synonym for *macro*.

main function. An external function with the identifier `main` that is the first user function—aside from exit routines and C++ static object constructors—to get control when program execution begins. Each C and C++ program must have exactly one function named `main`.

makefile. A text file containing a list of your application's parts. The make utility uses makefiles to maintain application parts and dependencies.

make utility. Maintains all of the parts and dependencies for your application. The make utility uses a makefile to keep the parts of your program synchronized. If one part of your application changes, the make utility updates all other files that depend on the changed part. This utility is available under the z/OS shell and by default, uses the c89 utility to recompile and bind your application.

mangling. The encoding during compilation of identifiers such as function and variable names to include type and scope information. These mangled names ensure type-safe linkage. See also *demangling*.

manipulator. A value that can be inserted into streams or extracted from streams to affect or query the behavior of the stream.

member. A data object or function in a structure, union, or class. Members can also be classes, enumerations, bit fields, and type names.

member function. (1) An 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 objects of its class. Member functions are also called methods. (2) A function that performs operations on a class.

method. In the C++ language, a synonym for *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 move to a changed operating environment, usually to a new release or version of a system. *IBM*.

module. A program unit that usually performs a particular function or related functions, and that is distinct 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.

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 the C++ language through derivation, in which the derived class inherits members from more than one base class.

multitasking. A mode of operation that allows concurrent performance, or interleaved execution of two or more tasks. *ISO-JTC1*. *ANSI/ISO*.

mutex. A flag used by a semaphore to protect shared resources. The mutex is locked and unlocked by threads in a program. A mutex can only be locked by one thread at a time and can only be unlocked by the same thread that locked it. The current owner of a mutex is the thread that it is currently locked by. An unlocked mutex has no current owner.

mutex attribute object. Allows the user to manage the characteristics of mutexes in their application by defining a set of values to be used for the mutex during its creation. A mutex attribute object allows the user to create many mutexes with the same set of characteristics without redefining the same set of characteristics for each mutex created.

mutex object. Used to identify a mutex.

N

namespace. A category used to group similar types of identifiers.

named pipe. A FIFO file. Named pipes allow transfer of data between processes in a FIFO manner and synchronization of process execution. Allows processes to communicate even though they do not know what processes are on the other end of the pipe.

natural reentrancy. A program that contains no writable static and requires no additional processing to make it reentrant is considered naturally reentrant.

nested class. A 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. See also *child enclave* and *parent enclave*.

newline character. A character that in the output stream indicates that printing should start at the beginning of the next line. The newline character is designated by '\n' in the C and C++ language. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next line. *X/Open*.

nickname. Synonym for alias.

non-printing character. See *control character*.

null character (NUL). The ASCII or EBCDIC character '\0' with the hex value `00`, all bits turned off. It is used to represent the absence of a printed or displayed character. This character is named <NUL> in the portable character set.

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. *X/Open*.

null statement. A C or C++ statement that consists solely of a semicolon.

null string. (1) A string whose first byte is a null byte. Synonymous with *empty string*. *X/Open*. (2) A character array whose first element is a null character. *ISO.1*.

null value. A parameter position for which no value is specified. *IBM*.

null wide-character code. A wide-character code with all bits set to zero. *X/Open*.

number sign. The character #, also known as *pound sign* and *hash sign*. This character is named <number-sign> in the portable character set.

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. (See also *instance*.) (2) In object-oriented design or programming, an abstraction consisting of data and the operations associated with that data. See also *class*. *IBM*. (3) An instance of a class.

object code. Machine-executable instructions, usually generated by a compiler from source code written in a higher level language (such as the C++ language). For programs that must be linked, object code consists of relocatable machine code.

object module. (1) All or part of an object program sufficiently complete for linking. Assemblers and compilers usually produce object modules. *ISO Draft*. (2) A set of instructions in machine language produced by a compiler from a source program. *IBM*.

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 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. *X/Open*. *ISO.1*.

operand. An entity on which an operation is performed. *ISO-JTC1*. *ANSI/ISO*.

operating system (OS). Software that controls functions such as resource allocation, scheduling, input/output control, and data management.

operator function. An overloaded operator that is either a member of a class or that takes at least one argument that is a class type or a reference to a class type.

operator precedence. In programming languages, an order relation defining the sequence of the application of operators within an expression. *ISO-JTC1*.

orientation of a stream. After application of an input or output function to a stream, it becomes either byte-oriented or wide-oriented. A byte-oriented stream is a stream that had a byte input or output function applied to it when it had no orientation. A wide-oriented stream is a stream that had a wide character input or output function applied to it when it had no orientation. A stream has no orientation when it has been associated with an external file but has not had any operations performed on it.

overflow. (1) A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage. (2) That portion of an operation that exceeds the capacity of the intended unit of storage. *IBM*.

overlay. The technique of repeatedly using the same areas of internal storage during different stages of a program. *ANSI/ISO*. Unions are used to accomplish this in C and C++.

overloading. An object-oriented programming technique that allows you to redefine functions and most standard C++ operators when the functions and operators are used with class types.

P

parameter. (1) In the C and C++ languages, an object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier following the macro name in a function-like macro definition. *X/Open*. (2) Data passed between programs or procedures. *IBM*.

parameter declaration. A 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. See also *child enclave* and *nested enclave*.

parent process. (1) The program that originates the creation of other processes by means of `spawn` or `exec` function calls. See also *child process*. (2) A process that creates other processes.

parent process ID. (1) 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. *X/Open*. (2) An attribute of a new process after it is created by a currently active process. *ISO.1*.

partitioned concatenation. Specifying multiple PDSs or PDSEs under one ddname. The concatenated data sets act as one big PDS or PDSE and access can be made to any member with a unique name. An attempted access to a member whose name occurs more than once in the concatenated data sets, returns the first member with that name found in the entire concatenation.

partitioned data set (PDS). A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data. *IBM*.

partitioned data set extended (PDSE). Similar to *partitioned data set*, but with extended capabilities.

path name. (1) A string that is used to identify a file. A path name consists of, at most, {PATH_MAX} bytes, including the terminating null character. It has an optional beginning slash, followed by zero or more file names separated by slashes. If the path name refers to a directory, it may also have one or more trailing slashes. Multiple successive slashes are treated as one slash. A path name that begins with two successive slashes may be interpreted in an implementation-dependent manner, although more than two leading slashes are treated as a single slash. The interpretation of the path name is described in *path name resolution*. *ISO.1*. (2) A file name specifying all directories leading to the file.

path name resolution. Path name resolution is performed for a process to resolve 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. *X/Open*.

period. The character (**.**). The term *period* is contrasted against *dot*, which is used to describe a specific directory entry. This character is named <period> in the portable character set.

permissions. Codes that determine how a file can be used by any users who work on the system. See also *file access permissions*. *IBM*.

persistent environment. A program can explicitly establish a persistent environment, direct functions to it, and explicitly terminate it.

pointer. In the C and C++ languages, a variable that holds the address of a data object or a function. *IBM*.

pointer class. A class that implements pointers.

pointer to member. An operator used to access the address of non-static members of a class.

polymorphism. The technique of taking an abstract view of an object or function and using any concrete objects or arguments that are derived from this abstract view.

portable character set. The set of characters specified in POSIX 1003.2, section 2.4:

```
<NUL>
<alert>
<backspace>
<tab>
<newline>
<vertical-tab>
<form-feed>
<carriage-return>
<space>
<exclamation-mark>     !
<quotation-mark>       "
<number-sign>          #
<dollar-sign>          $
<percent-sign>         %
<ampersand>            &
<apostrophe>           '
<left-parenthesis>     (
<right-parenthesis>    )
<asterisk>             *
<plus-sign>            +
<comma>                ,
<hyphen>               -
<hyphen-minus>         -
<period>               .
<slash>                /
<zero>                 0
<one>                  1
<two>                  2
<three>                3
<four>                 4
<five>                 5
<six>                  6
<seven>                7
<eight>                8
<nine>                 9
<colon>                :
<semicolon>            ;
<less-than-sign>       <
<equals-sign>          =
<greater-than-sign>    >
<question-mark>        ?
<commercial-at>        @

<A>                    A
<B>                    B
<C>                    C
<D>                    D
<E>                    E
<F>                    F
<G>                    G
<H>                    H
<I>                    I
```

```
<J>                     J
<K>                     K
<L>                     L
<M>                     M
<N>                     N
<O>                     O
<P>                     P
<Q>                     Q
<R>                     R
<S>                     S
<T>                     T
<U>                     U
<V>                     V
<W>                     W
<X>                     X
<Y>                     Y
<Z>                     Z

<left-square-bracket>   [
<backslash>             \
<reverse-solidus>       \
<right-square-bracket>  ]
<circumflex>            ^
<circumflex-accent>     ^
<underscore>            _
<low-line>              _
<grave-accent>          `
<a>                     a
<b>                     b
<c>                     c
<d>                     d
<e>                     e
<f>                     f
<g>                     g
<h>                     h
<i>                     i
<j>                     j
<k>                     k
<l>                     l

<m>                     m
<n>                     n
<o>                     o
<p>                     p
<q>                     q
<r>                     r
<s>                     s
<t>                     t
<u>                     u
<v>                     v
<w>                     w
<x>                     x
<y>                     y
<z>                     z

<left-brace>            {
<left-curly-bracket>    {
<vertical-line>         |
<right-brace>           }
<right-curly-bracket>   }
<tilde>                 ~
```

portable file name character set. The set of characters from which portable file names are constructed. For a file name to be portable across implementations conforming to the ISO POSIX-1 standard and to ISO/IEC 9945, it must consists only of the following characters:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -
```

The last three characters are the period, underscore, and hyphen characters, respectively. The hyphen must not be used as the first character of a portable file name. Upper- and lower-case letters retain their unique identities between conforming implementations. In the case of a portable path name, the slash character may also be used. *X/Open*. *ISO.1*.

portability. The ability of a programming language to compile successfully on different operating systems without requiring changes to the source code.

positional parameter. A parameter that must appear in a specified location relative to other positional parameters. *IBM*.

precedence. The priority system for grouping different types of operators with their operands.

predefined macros. Frequently used routines provided by an application or language for the programmer.

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 provided with z/OS Language Environment that you can use to process application programs that require DLL support, or contain either constructed reentrancy or external symbol names that are longer than 8 characters. You require the prelinker, or its equivalent function which is provided by the binder, to process all C++ applications, or C applications that are compiled with the RENT, DLL, LONGNAME or IPA options. As of Version 2 Release 4, the prelinker was superseded by the binder. See also *binder*.

preprocessor. A phase of the compiler that examines the source program for preprocessor statements that are then executed, resulting in the alteration of the source program.

preprocessor statement. In the C and C++ languages, a statement that begins with the symbol # and is interpreted by the preprocessor during compilation. *IBM*.

primary expression. (1) An identifier, parenthesized expression, function call, array element specification, structure member specification, or union member specification. *IBM*. (2) Literals, names, and names qualified by the :: (scope resolution) operator.

printable character. One of the characters included in the print character classification of the LC_CTYPE category in the current locale. *X/Open*.

private. Pertaining to a class member that is only accessible to member functions and friends of that class.

process. (1) An instance of an executing application and the resources it uses. (2) 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. *X/Open*. *ISO.1*.

process group. A collection of processes that permits the signaling of related processes. Each process in the system is a member of a process group that is identified by the process group ID. A newly created process joins the process group of its creator. *IBM*. *X/Open*. *ISO.1*.

process group ID. The unique identifier representing a process group during its lifetime. A process group ID is a positive integer. (Under ISO only, it is a positive integer *that can be contained in a pid_t*.) A process group ID will not be reused by the system until the process group lifetime ends. *X/Open*. *ISO.1*.

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. The unique identifier representing a process. A process ID is a positive integer. (Under ISO only, it is a positive integer *that can be contained in a pid_t*.) A process ID will not be reused by the system until the process lifetime ends. In addition, if there exists a process group whose process group ID is equal to that process ID, the process ID will not be reused by the system until the process group lifetime ends. A process that is not a system process will not have a process ID of 1. *X/Open*. *ISO.1*.

process lifetime. The period of time that begins when a process is created and ends when the process ID is returned to the system. 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. *X/Open*. *ISO.1*.

program object. All or part of a computer program in a from suitable for loading into main storage for execution. A program object is the output of the z/OS Binder and is a newer more flexible format (e.g. longer external names) than a load module.

protected. Pertaining to a class member that is only accessible to member functions and friends of that class, or to member functions and friends of classes derived from that class.

prototype. A function declaration or definition that includes both the return type of the function and the types of its parameters. See *function prototype*.

public. Pertaining to a class member that is accessible to all functions.

pure virtual function. A virtual function that has a function definition of `= 0;`. See also *abstract classes*.

Q

qualified class name. Any class name or class name qualified with one or more :: (scope resolution) operators.

qualified name. Used to qualify a non-class type name such as a member by its class name.

qualified type name. Used to reduce complex class name syntax by using typedefs to represent qualified class names.

Query Management Facility (QMF). Pertaining to an IBM query and report writing facility that enables a variety of tasks such as data entry, query building, administration, and report analysis. *IBM*.

queue. A sequence with restricted access in which elements can only be added at the back end (or bottom) and removed from the front end (or top). A queue is characterized by first-in, first-out behavior and chronological order.

quotation marks. The characters " and ‘, also known as *double-quote* and *single-quote* respectively. *X/Open*.

R

radix character. The character that separates the integer part of a number from the fractional part. *X/Open*.

real group ID. The attribute of a process that, at the time of process creating, identifies the group of the user who created the process. This value is subject to change during the process lifetime, as describe in `setgid()`. *X/Open*. *ISO.1*.

real user ID. The attribute of a process that, at the time of process creation, identifies the user who created the process. This value is subject to change during the process lifetime, as described in `setuid()`. *X/Open*. *ISO.1*.

reason code. A code that identifies the reason for a detected error. *IBM*.

reassociation. An optimization technique that rearranges the sequence of calculations in a subscript expression producing more candidates for common expression elimination.

redirection. In the shell, a method of associating files with the input or output of commands. *X/Open*.

reentrant. The attribute of a program or routine that allows the same copy of a 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. Synonymous with *indirection class*.

refresh. To ensure that the information on the user's terminal screen is up-to-date. *X/Open*.

register storage class specifier. A specifier that indicates to the compiler within a block scope data definition, or a parameter declaration, that the object being described will be heavily used.

register variable. A variable defined with the register storage class specifier. Register variables have automatic storage.

regular expression. (1) A mechanism to select 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. *X/Open*. *ISO.1*.

relation. An unordered flat collection class that uses keys, allows for duplicate elements, and has element equality.

relative path name. The name of a directory or file expressed as a sequence of directories followed by a file name, beginning from the current directory. See *path name resolution*. *IBM*.

reserved word. (1) In programming languages, a keyword that may not be used as an identifier. *ISO-JTC1*. (2) A word used in a source program to describe an action to be taken by the program or compiler. It must not appear in the program as a user-defined name or a system name. *IBM*.

RMODE (residency mode). 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.

RTTI. Use the RTTI option to generate run-time type identification (RTTI) information for the typeid operator and the dynamic_cast operator.

run-time library. A compiled collection of functions whose members can be referred to by an application program during run-time execution. Typically used to refer to a dynamic library that is provided in object code, such that references to the library are resolved during the linking step. The run-time library itself is not statically bound into the application modules.

S

saved set-group-ID. An attribute of a process that allows some flexibility in the assignment of the effective group ID attribute, as described in the `exec()` family of functions and `setgid()`. *X/Open*. *ISO.1*.

saved set-user-ID. An attribute of a process that allows some flexibility in the assignment of the effective user ID attribute, as described in `exec()` and `setuid()`. *X/Open*. *ISO.1*.

scalar. An arithmetic object, or a pointer to an object of any type.

scope. (1) That part of a source program in which a variable is visible. (2) That part of a source program in which an object is defined and recognized.

scope operator (::). 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, the scope is within that class. Synonymous with *scope resolution operator*.

scope resolution operator (::). Synonym for *scope operator*.

semaphore. An object used by multi-threaded applications for signalling 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 concatenation. Multiple sequential data sets or partitioned data-set members are treated as one long sequential data set. In the case of sequential data sets, you can access or update the data sets in order. In the case of partitioned data-set members, you can access or update the members in order. Repositioning is possible if all of the data sets in the concatenation support repositioning.

sequential data set. A data set whose records are organized on the basis of their successive physical positions, such as on magnetic tape. *IBM*.

session. A collection of process groups established for job control purposes. Each process group is a member of a session. A process is a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership; see `setsid()`. There can be multiple process groups in the same session. *X/Open*. *ISO.1*.

shell. A program that interprets sequences of text input as commands. It may operate on an input stream or it may interactively prompt and read commands from a terminal. *X/Open*.

This feature is provided as part of the z/OS Shell and Utilities feature licensed program.

Short name. 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.

signal. (1) A condition that may or may not be reported during program execution. For example, `SIGFPE` is the signal used to represent erroneous arithmetic operations such as a division by zero. (2) A mechanism by which a process may 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. *X/Open*. *ISO.1*. (3) A method of interprocess communication that simulates software interrupts. *IBM*.

signal handler. A function to be called when the signal is reported.

single-byte character set (SBCS). A set of characters in which each character is represented by a one-byte code. *IBM*.

single-precision. Pertaining to the use of one computer word to represent a number in accordance with the required precision. *ISO-JTC1*. *ANSI/ISO*.

single-quote. The character ‘, also known as *apostrophe*. This character is named <quotation-mark> in the portable character set.

slash. The character /, also known as *solidus*. This character is named <slash> in the portable character set.

socket. (1) A unique host identifier created by the concatenation of a port identifier with a transmission control protocol/Internet protocol (TCP/IP) address. (2) A port identifier. (3) A 16-bit port-identifier. (4) A port on a specific host; a communications end point that is accessible though a protocol family's addressing mechanism. A socket is identified by a socket address. *IBM*.

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. A file that contains source statements for such items as high-level language programs and data description specifications. *IBM*.

source program. A set of instructions written in a programming language that must be translated to machine language before the program can be run. *IBM*.

space character. The character defined in the portable character set as <space>. The space character is a member of the space character class of the current locale, but represents the single character, and not all of the possible members of the class. *X/Open*.

spanned record. A logical record contained in more than one block. *IBM*.

specialization. A user-supplied definition which replaces a corresponding template instantiation.

specifiers. Used in declarations to indicate storage class, fundamental data type and other properties of the object or function being declared.

spill area. A storage area used to save the contents of registers. *IBM*.

SQL (Structured Query Language). A language designed to create, access, update and free data tables.

square brackets. The characters [(left bracket) and] (right bracket). Also see *brackets*.

stack frame. The physical representation of the activation of a routine. The stack frame is allocated and freed on a LIFO (last in, first out) basis. A stack is a collection of one or more stack segments consisting of an initial stack segment and zero or more increments.

stack storage. Synonym for *automatic storage*.

standard error. An output stream usually intended to be used for diagnostic messages. *X/Open*.

standard input. (1) An input stream usually intended to be used for primary data input. *X/Open*. (2) The primary source of data entered into a command. Standard input comes from the keyboard unless

redirection or piping is used, in which case standard input can be from a file or the output from another command. *IBM*.

standard output. (1) An output stream usually intended to be used for primary data output. *X/Open*. (2) The primary destination of data coming from a command. Standard output goes to the display unless redirection or piping is used, in which case standard output can go to a file or to another command. *IBM*.

statement. An instruction that ends with the character **;** (semicolon) or several instructions that are surrounded by the characters { and }.

static. A keyword used for defining the scope and linkage of variables and functions. For internal variables, the variable has block scope and retains its value between function calls. For external values, the variable has file scope and retains its value within the source file. For class variables, the variable is shared by all objects of the class and retains its value within the entire program.

static binding. The act of resolving references to external variables and functions before run time.

storage class specifier. One of the terms used to specify a storage class, such as auto, register, static, or extern.

stream. (1) A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format. (2) 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. *X/Open*.

string. A contiguous sequence of bytes terminated by and including the first null byte. *X/Open*.

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. A special data set organization that spreads a data set over a specified number of volumes so that I/O parallelism can be exploited. Record *n* in a striped data set is found on a volume separate from the volume containing record *n* - *p*, where *n* > *p*.

struct. An aggregate of elements having arbitrary types.

structure. A construct (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. A structure can be used in all places a class is used. The initial projection is public.

structure tag. The identifier that names a structure data type.

Structured Query Language. See *SQL*.

stub routine. A routine, within a run-time library, that contains the minimum lines of code required to locate a given routine at run time.

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.

subsystem. A secondary or subordinate system, usually capable of operating independently of or asynchronously with, a controlling system. *ISO Draft*.

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. *IBM*.

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. *IBM*.

system process. (1) An implementation-dependent object, other than a process executing an application, that has a process ID. *X/Open*. (2) An object, other than a process executing an application, that is defined by the system, and has a process ID. *ISO.1*.

T

tab character. A character that in the output stream indicates that printing or displaying should start at the next horizontal tabulation position on the current line. The tab is the character designated by '\t' in the C language. If the current position is at or past the last defined horizontal tabulation position, the behavior is unspecified. It is unspecified whether the character is the exact sequence transmitted to an output device by the system to accomplish the tabulation. *X/Open*.

This character is named <tab> in the portable character set.

task library. A class library that provides the facilities to write programs that are made up of tasks.

template. A family of classes or functions with variable types.

template class. A class instance generated by a class template.

template function. A 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.

terminals. Synonym for *leaves*.

text file. A file that contains characters organized into one or more lines. The lines must not contain NUL characters and none can exceed {LINE_MAX}—which is defined in limits.h—bytes in length, including the new-line character. The term *text file* does not prevent the inclusion of control or other unprintable characters (other than NUL). *X/Open*.

thread. The smallest unit of operation to be performed within a process. *IBM*.

throw expression. An argument to the C++ exception being thrown.

tilde. The character ~. This character is named <tilde> in the portable character set.

token. The smallest independent unit of meaning of a program as defined either by a parser or a lexical analyzer. A token can contain data, a language keyword, an identifier, or other parts of language syntax. *IBM*.

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 the status of the routines on the call-chain at the time the traceback was produced.

trigraph sequence. An alternative spelling of some characters to allow the implementation of C in character sets that do not provide a sufficient number of non-alphabetic graphics. *ANSI/ISO*.

Before preprocessing, each trigraph sequence in a string or literal is replaced by the single character that it represents.

truncate. To shorten a value to a specified length.

try block. A block in which a known C++ exception is passed to a handler.

type definition. A definition of a name for a data type. *IBM*.

type specifier. Used to indicate the data type of an object or function being declared.

U

ultimate consumer. The target of data in an I/O operation. An ultimate consumer can be a file, a device, or an array of bytes in memory.

ultimate producer. The source of data in an I/O operation. An ultimate producer can be a file, a device, or an array of byes in memory.

unary expression. An expression that contains one operand. *IBM*.

undefined behavior. Action by the compiler and library when the program uses erroneous constructs or contains erroneous data. Permissible undefined behavior includes ignoring the situation completely with unpredictable results. It also includes behaving in a documented manner that is characteristic of the environment, during translation or program execution, with or without issuing a diagnostic message. It can also include terminating a translation or execution, while issuing a diagnostic message. Contrast with *unspecified behavior* and *implementation-defined behavior*.

underflow. (1) A condition that occurs when the result of an operation is less than the smallest possible nonzero number. (2) Synonym for arithmetic underflow, monadic operation. *IBM.*

union. (1) In the C or C++ language, a variable that can hold any one of several data types, but only one data type at a time. *IBM*. (2) For bags, there is an additional rule for duplicates: If bag P contains an element *m* times and bag Q contains the same element *n* times, then the union of P and Q contains that element *m+n* times.

union tag. The identifier that names a union data type.

unnamed pipe. A pipe that is accessible only by the process that created the pipe and its child processes. An unnamed pipe does not have to be opened before it can be used. It is a temporary file that lasts only until the last file descriptor that uses it is closed.

unique collection. A collection in which the value of an element only occurs once; that is, there are no duplicate elements.

unrecoverable error. An error for which recovery is impossible without use of recovery techniques external to the computer program or run.

unspecified behavior. Action by the compiler and library when the program uses correct constructs or data, for which the standards impose no specific

requirements. Such action should not cause compiler or application failure. You should not, however, write any programs to rely on such behavior as they may not be portable to other systems. Contrast with *implementation-defined behavior* and *undefined behavior*.

user-defined data type. (1) 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. (2) See also *abstract data type*.

user ID. A nonnegative integer that is used to identify a system user. (Under ISO only, a nonnegative integer, which can be contained in an object of type *uid_t*.) When the identity of a user is associated with a process, a user ID value is referred to as a real user ID, an effective user ID, or (under ISO only, and there optionally) a saved set-user ID. *X/Open*. *ISO.1*.

user name. A string that is used to identify a user. *ISO.1*.

user prefix. In the z/OS environment, the user prefix is typically the user's logon user identification.

V

value numbering. An optimization technique that involves local constant propagation, local expression elimination, and folding several instructions into a single instruction.

variable. In programming languages, a language object that may take different values, one at a time. The values of a variable are usually restricted to a certain data type. *ISO-JTC1*.

variant character. A character whose hexadecimal value differs between different character sets. On EBCDIC systems, such as S/390, these 13 characters are an exception to the portability of the portable character set.

```
<left-square-bracket>  [
<right-square-bracket> ]
<left-brace>           {
<right-brace>          }
<backslash>            \
<circumflex>           ^
<tilde>                ~
<exclamation-mark>     !
<number-sign>          #
<vertical-line>        |
<grave-accent>         `
<dollar-sign>          $
<commercial-at>        @
```

vertical-tab character. A character that in the output stream indicates that printing should start at the next vertical tabulation position. The vertical-tab is the character designated by '\v' in the C or C++ languages. If the current position is at or past the last defined vertical tabulation position, the behavior is unspecified. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the tabulation. *X/Open*. This character is named <vertical-tab> in the portable character set.

virtual address space. In virtual storage systems, the virtual storage assigned to a batched or terminal job, a system task, or a task initiated by a command.

virtual function. A function of a class that is declared with the keyword `virtual`. The implementation that is executed when you make a call to a virtual function depends on the type of the object for which it is called, which is determined at run time.

Virtual Storage Access Method (VSAM). An access method for direct or sequential processing of fixed and variable length records on direct access devices. The records in a VSAM data set or file can be organized in logical sequence by a key field (key sequence), in the physical sequence in which they are written on the data set or file (entry-sequence), or by relative-record number.

visible. Visibility of identifiers is based on scoping rules and is independent of *access.*

volatile attribute. (1) In the C or C++ language, the keyword *volatile*, used in a definition, declaration, or cast. It causes the compiler to place the value of the data object in storage and to reload this value at each reference to the data object. *IBM*. (2) 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 contains the keyword *while* followed by an expression in parentheses (the condition) and a statement (the action). *IBM*.

white space. (1) Space characters, tab characters, form-feed characters, and new-line characters. (2) A sequence of one or more characters that belong to the space character class as defined via the LC_CTYPE category in the current locale. In the POSIX locale, white space consists of one or more blank characters (space and tab characters), new-line characters, carriage-return characters, form-feed characters, and vertical-tab characters. *X/Open*.

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 corresponding to a single graphic symbol or control code. *X/Open*.

wide-character string. A contiguous sequence of wide-character codes terminated by and including the first null wide-character code. *X/Open*.

wide-oriented stream. See *orientation of a stream*.

word. A character string considered as a unit for a given purpose. In z/OS, a word is 32 bits or 4 bytes.

working directory. Synonym for *current working directory*.

writable static area. See WSA.

write. (1) 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*. *X/Open*. (2) To make a permanent or transient recording of data in a storage device or on a data medium. *ISO-JTC1*. *ANSI/ISO*.

WSA (writable static area). An area of memory in the program that is modifyable during program execution. Typically, this area contains global variables and function and variable descriptors for DLLs.

X

XPLINK (Extra Performance Linkage). A new call linkage between functions that 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 nonessential instructions from the main path. When all functions are compiled with the XPLINK option, pointers can be used without restriction, which makes it easier to port new applications to z/OS.

Z

z/OS UNIX System Services (z/OS UNIX). An element of the z/OS operating system, (formerly known as OpenEdition). z/OS UNIX includes a POSIX system Application Programming Interface for the C language, a shell and utilities component, and a dbx debugger. All the components conform to IEEE POSIX standards (ISO 9945-1: 1990/IEEE POSIX 1003.1-1990, IEEE POSIX 1003.1a, IEEE POSIX 1003.2, and IEEE POSIX 1003.4a).

Bibliography

This bibliography lists the publications for IBM products that are related to the z/OS C/C++ product. 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 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 Changes*, SA22-7505
- *z/OS Information Roadmap*, SA22-7500

z/OS C/C++

- *z/OS C/C++ Programming Guide*, SC09-4765
- *z/OS C/C++ User's Guide*, SC09-4767
- *C/C++ Language Reference*, SC09-4815
- *z/OS C/C++ Messages*, GC09-4819
- *z/OS C/C++ Run-Time Library Reference*, SA22-7821
- *z/OS C Curses*, SA22-7820
- *z/OS C/C++ Compiler and Run-Time Migration Guide*, GC09-4913
- *IBM Open Class Library User's Guide*, SC09-4811
- *IBM Open Class Library Reference*, SC09-4812
- *Debug Tool User's Guide and Reference*, SC09-2137
- *Standard C++ Library Reference*, which is available at: `http://www.ibm.com/software/ad/c390/czos/czosdocs.html`

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 Migration Guide*, GA22-7565
- *z/OS Language Environment Writing Interlanguage 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 for OS/390 & VM Compiler and Run-Time Migration Guide*, GC26-4764
- *COBOL for OS/390 & VM Programming Guide*, SC26-9049
- *COBOL for OS/390 & VM Language Reference*, SC26-9046
- *COBOL for OS/390 & VM Diagnosis Guide*, GC26-9047
- *COBOL for OS/390 & VM Licensed Program Specifications*, GC26-9044
- *COBOL for OS/390 & VM Customization under OS/390*, GC26-9045
- *COBOL Millenium Language Extensions Guide*, GC26-9266

PL/I

- *VisualAge PL/I Language Reference*, SC26-9476
- *PL/I for MVS & VM Language Reference*, SC26-3114
- *PL/I for MVS & VM Programming Guide*, SC26-3113
- *PL/I for MVS & VM Compiler and Run-Time Migration Guide*, SC26-3118

VS FORTRAN

- *Language and Library Reference*, SC26-4221
- *Programming Guide*, SC26-4222

CICS

- *CICS Application Programming Guide*, SC34-5702
- *CICS Application Programming Reference*, SC34-5703
- *CICS Distributed Transaction Programming Guide*, SC34-5708
- *CICS Front End Programming Interface User's Guide*, SC34-5710
- *CICS Messages and Codes*, GC33-5716
- *CICS Resource Definition Guide*, SC34-5722
- *CICS System Definition Guide*, SC34-5725
- *CICS System Programming Reference*, SC34-5726
- *CICS User's Handbook*, SX33-6116
- *CICS Family: Client/Server Programming*, SC34-1435
- *CICS Transaction Server for OS/390 Migration Guide*, GC34-5699
- *CICS Transaction Server for OS/390 Release Guide*, GC34-5701
- *CICS Transaction Server for OS/390: Planning for Installation*, GC34-5700

DB2

- *DB2 Administration Guide*, SC26-9931
- *DB2 Application Programming and SQL Guide*, SC26-9933
- *DB2 ODBC Guide and Reference*, GC26-9941
- *DB2 Command Reference*, SC26-9934
- *DB2 Data Sharing: Planning and Administration*, SC26-9935
- *DB2 Installation Guide*, GC26-9936
- *DB2 Messages and Codes*, GC26-9940
- *DB2 Reference for Remote DRDA Requesters and Servers*, SC26-9942
- *DB2 SQL Reference*, SC26-9944

- *DB2 Utility Guide and Reference*, SC26-9945

IMS/ESA

- *IMS/ESA Application Programming: Design Guide*, SC26-8728
- *IMS/ESA Application Programming: Transaction Manager*, SC26-8729
- *IMS/ESA Application Programming: Database Manager*, SC26-8727
- *IMS/ESA Application Programming: EXEC DLI Commands for CICS and IMS*, SC26-8726

QMF

- *Introducing QMF*, GC26-9576
- *Using QMF*, SC26-9578
- *Developing QMF Applications*, SC26-9579
- *Reference*, SC26-9577
- *Installing and Managing QMF on MVS*, SC26-9575
- *Messages and Codes*, SC26-9580

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*, SC26-7394
- *z/OS DFSMS Program Management*, SC27-1130

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

W

X

Z

Program Number: 5694-A01

Printed in the United States of America

SC09-4767-01