

z/OS



C/C++ User's Guide

z/OS



C/C++ User's Guide

Note!

Before using this information and the product it supports, be sure to read the information in "Notices" on page 623.

Fourth Edition (September 2004)

This edition applies to Version 1 Release 6 of z/OS C/C++ (5694-A01), Version 1 Release 6 of z/OS.e (5655-G52), and to all subsequent releases until otherwise indicated in new editions. This edition replaces SC09-4767-02 . 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/OShome 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 to the following Internet address: compinfo@ca.ibm.com. Be sure to include your e-mail address if you want a reply.

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

© Copyright International Business Machines Corporation 1996, 2004. All rights reserved.

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

Contents

About this document	xv
How to read syntax diagrams.	xv
Symbols	xv
Syntax items	xv
Syntax examples	xvi
z/OS C/C++ and related publications	xvii
Softcopy documents.	xxii
Softcopy examples	xxii
z/OS C/C++ on the World Wide Web	xxiii
Where to find more information	xxiii

Part 1. Introduction 1

Chapter 1. About IBM z/OS C/C++	3
Changes for z/OS V1R6	3
The C/C++ compilers	5
The C language	5
The C++ language	5
Common features of the z/OS C and C++ compilers	5
z/OS C Compiler specific features	7
z/OS C++ Compiler specific features	7
Class libraries.	7
Utilities	7
dbx	8
z/OS Language Environment	8
z/OS Language Environment downward compatibility	9
About prelinking, linking, and binding	10
Notes on the prelinking process.	10
File format considerations	11
The program management binder	11
z/OS UNIX System Services	12
z/OS C/C++ Applications with z/OS UNIX System Services C/C++ functions	13
Input and output	14
I/O interfaces	14
File types	14
Additional I/O features	15
The System Programming C facility	16
Interaction with other IBM products	16
Additional features of z/OS C/C++	18

Part 2. User's reference 21

Chapter 2. z/OS C example	23
Example of a C program	23
CCNUAAM	23
CCNUAAN	24
Compiling, binding, and running the z/OS C example	25
Under z/OS batch.	25
Non-XPLINK and XPLINK under TSO	26
Non-XPLINK and XPLINK under the z/OS UNIX System Services shell	27
Chapter 3. z/OS C++ examples	29

Example of a C++ program	29
CCNUBRH	30
CCNUBRC	32
Compiling, binding, and running the z/OS C++ example	34
Under z/OS batch	34
Non-XPLINK and XPLINK under TSO	35
Non-XPLINK and XPLINK under the z/OS UNIX shell	37
Example of a C++ template program	37
CLB3ATMP.CXX	38
Compiling, binding, and running the C++ template example	39
Under z/OS batch	39
Under TSO	41
Under the z/OS UNIX shell	42
Chapter 4. Compiler Options	43
Specifying compiler options	43
IPA considerations	44
Using special characters	46
Specifying z/OS C compiler options using #pragma options	47
Specifying compiler options under z/OS UNIX System Services	47
Compiler option defaults	48
Summary of compiler options	50
Compiler options for file management	54
Options that control the preprocessor	55
Options that control the processing of an input source file	55
Options that control the compiler listing	56
Options that control the IPA object	57
Options that control the IPA Link step	57
Options for debugging and diagnosing errors	58
Options that control the programming language characteristics	59
Options that control object code generation	60
Options that control program execution	63
Portability options	63
Description of compiler options	63
AGGRCOPY	64
AGGREGATE NOAGGREGATE	65
ALIAS NOALIAS	66
ANSIALIAS NOANSIALIAS	67
ARCHITECTURE	70
ARGPARSE NOARGPARSE	73
ASCII NOASCII	74
ATTRIBUTE NOATTRIBUTE	75
BITFIELD(SIGNED) BITFIELD(UNSIGNED)	76
CHARS(SIGNED) CHARS(UNSIGNED)	76
CHECKOUT NOCHECKOUT	77
COMPACT NOCOMPACT	79
COMPRESS NOCOMPRESS	81
CONVLIT NOCONVLIT	82
CSECT NOCSECT	84
CVFT NOCVFT	86
DBRMLIB	88
DEBUG NODEBUG	88
DEFINE	92
DIGRAPH NODIGRAPH	93
DLL NODLL	95
ENUMSIZE	97

EVENTS NOEVENTS	99
EXECOPS NOEXECOPS	100
EXH NOEXH	101
EXPMAC NOEXPMAC	102
EXPORTALL NOEXPORTALL	103
FASTTEMPINC NOFASTTEMPINC	103
FLAG NOFLAG	104
FLOAT	105
GOFF NOGOFF	110
GONUMBER NOGONUMBER	111
HALT(num)	112
HALTONMSG NOHALTONMSG	113
IGNERRNO NOIGNERRNO	114
INFO NOINFO	115
INITAUTO NOINITAUTO	116
INLINE NOINLINE	118
INLRPT NOINLRPT	121
IPA NOIPA	122
KEYWORD NOKEYWORD	130
LANGLVL	131
LIBANSI NOLIBANSI	142
LIST NOLIST	143
LOCALE NOLOCALE	145
LONGNAME NOLONGNAME	147
LP64 ILP32	148
LSEARCH NOLSEARCH	150
MARGINS NOMARGINS	156
MAXMEM NOMAXMEM	157
MEMORY NOMEMORY	158
NAMEMANGLING	159
NESTINC NONESTINC	161
OBJECT NOOBJECT	162
OBJECTMODEL	163
OE NOOE	165
OFFSET NOOFFSET	166
OPTFILE NOOPTFILE	167
OPTIMIZE NOOPTIMIZE	169
PHASEID NOPHASEID	172
PLIST	173
PORT NOPORT	174
PPONLY NOPPONLY	175
REDIR NOREDIR	178
RENT NORENT	179
ROCONST NOROCONST	180
ROSTRING NOROSTRING	181
ROUND	182
RTTI NORTTI	183
SEARCH NOSEARCH	184
SEQUENCE NOSEQUENCE	185
SERVICE NOSERVICE	186
SHOWINC NOSHOWINC	187
SOURCE NOSOURCE	188
SPILL NOSPILL	189
SQL NOSQL	191
SSCOMM NOSSCOMM	192
START NOSTART	193

STATICINLINE NOSTATICINLINE	194
STRICT NOSTRICT	195
STRICT_INDUCTION NOSTRICT_INDUCTION	196
SUPPRESS NOSUPPRESS	197
TARGET	197
TEMPINC NOTEMPINC	203
TEMPLATERECOMPILE NOTEMPLATERECOMPILE	204
TEMPLATEREGISTRY NOTEMPLATEREGISTRY	205
TERMINAL NOTERMINAL	206
TEST NOTEST	206
TMPLPARSE	211
TUNE	212
UNDEFINE	214
UNROLL	215
UPCONV NOUPCONV	216
WARN64 NOWARN64	216
WSIZEOF NOWSIZEOF	217
XPLINK NOXPLINK	218
XREF NOXREF	222
Using the z/OS C Compiler Listing	223
IPA Considerations	223
Example of a C Compiler Listing	223
z/OS C Compiler Listing Components	254
Using the z/OS C++ Compiler Listing	257
IPA Considerations	258
Example of a C++ Compiler Listing	258
z/OS C++ Compiler Listing Components	268
Using the IPA Link Step Listing	272
Example of an IPA Link Step Listing	272
IPA Link Step Listing Components	279
Chapter 5. Binder options and control statements	285
Chapter 6. Run-Time options	287
Specifying run-time options	287
Using the #pragma runopts preprocessor directive	287

Part 3. Compiling, binding, and running z/OS C/C++ programs 289

Chapter 7. Compiling	291
Input to the z/OS C/C++ compiler	291
Primary input	291
Secondary input	292
Output from the compiler	292
Specifying output files	292
Valid input/output file types	295
Compiling under z/OS batch	296
Using cataloged procedures for z/OS C	297
Using cataloged procedures for z/OS C++	298
Using special characters	298
Examples of compiling programs using your own JCL	299
Specifying source files	301
Specifying include files	302
Specifying output files	302
Compiling under TSO	303
Using the CC and CXX REXX EXECs	303

Specifying sequential and partitioned data sets	304
Specifying HFS files or directories	304
Compiling and binding in the UNIX System Services environment.	305
Compiling without binding using compiler invocation command names supported by c89 and xlc	307
Compiling and binding in one step using compiler invocation command names supported by c89 and xlc	309
Building an application with XPLINK using the c89 or xlc utilities	310
Building a 64-bit application using the c89 or xlc utilities	310
Invoking IPA using the c89 or xlc utilities	310
Using IPA(OBJONLY) with the c89 or xlc utilities	311
Using the make utility	311
Compiling with IPA	312
The IPA Compile step	312
The IPA Link step	313
Compiling with IPA(OBJONLY).	314
Working with object files	315
Browsing object files	315
Identifying object file variations	316
Using feature test macros	316
Using include files	316
Specifying include file names	317
Forming file names	317
Forming data set names with LSEARCH SEARCH options.	318
Search sequence	320
Determining whether the file name is in absolute form	321
Using SEARCH and LSEARCH	323
Search sequences for include files	324
With the NOOE option.	325
With the OE option	325
Compiling z/OS C source code using the SEARCH option	326
Compiling z/OS C++ source code using the SEARCH option	327
Chapter 8. Using the IPA Link step with z/OS C/C++ programs	329
Creating a module with IPA	329
Example 1. all C parts.	329
Example 2. all C parts built with XPLINK	340
Creating a DLL with IPA	341
Example 1. a mixture of C and C++.	342
Example 2. using the IPA control file	344
Using Profile-Directed Feedback (PDF)	345
Steps for completing the PDF process.	345
Steps for building a module in USS using PDF.	347
Reference Information.	348
Invoking IPA from the c89 utility	348
IPA Link step control file	349
Object file directives understood by IPA	352
Troubleshooting	352
Chapter 9. Binding z/OS C/C++ programs.	355
When you can use the binder	355
When you cannot use the binder	355
Your output is a PDS, not a PDSE	355
CICS	355
MTF	355
IPA.	355

Using different methods to bind	356
Single final bind	356
Bind each compile unit	357
Build and use a DLL	358
Rebind a changed compile unit	360
Binding under z/OS UNIX System Services	360
z/OS UNIX System Services example	361
Steps for single final bind using c89.	361
Steps for binding each compile unit using c89	362
Steps for building and using a DLL using c89	363
Steps for rebinding a changed compile unit using c89	364
Using the non-XPLINK version of the Standard C++ Library and c89	365
Binding under z/OS batch	366
z/OS batch example	366
Steps for single final bind under z/OS batch.	367
Steps for binding each compile unit under z/OS batch	368
Steps for building and using a DLL under z/OS batch	369
Build and use a 64-bit application under z/OS batch.	370
Build and use a 64-bit application with IPA under z/OS batch	371
Using the non-XPLINK version of the Standard C++ Library and z/OS batch	374
Steps for rebinding a changed compile unit under z/OS batch	375
Writing JCL for the binder	375
Binding under TSO using CXXBIND.	377
TSO example	379
Steps for single final bind under TSO	379
Steps for binding each compile unit under TSO	380
Steps for building and using a DLL under TSO.	380
Steps for rebinding a changed compile unit under TSO	381
Chapter 10. Binder processing	383
Linkage considerations	384
Primary input processing	385
C or C++ object module as input	385
Secondary input processing.	385
Load module as input	385
Program object as input	385
Autocall input processing (library search)	386
Incremental autocall processing (AUTOCALL control statement)	386
Final autocall processing (SYSLIB)	386
LP64 libraries	387
Rename processing	388
Generating aliases for automatic library call (library search)	388
Dynamic Link Library (DLL) processing	389
Statically bound functions	389
Imported variables	390
Imported functions	390
Output program object.	390
Output IMPORT statements.	390
Output listing	390
Header	392
Input Event Log	392
Module Map	393
Cross Reference Table	395
Imported and Exported Symbols Listing	395
Mangled to Demangled Symbol Cross Reference.	396
Processing Options.	397

Save Operation Summary	397
Save Module Attributes	397
Entry Point and Alias Summary	398
Long Symbol Abbreviation Table	398
DDname vs Pathname Cross Reference Table	399
Message Summary Report	399
Binder processing of C/C++ object to program object	400
Rebindability	401
Error recovery	403
Unresolved symbols	403
Significance of library search order	404
Duplicates	405
Duplicate functions from autocall	407
Hunting down references to unresolved symbols	407
Incompatible linkage attributes	407
Non-reentrant DLL problems	408
Code that has been prelinked	408
Chapter 11. Running a C or C++ application.	409
Setting the region size for z/OS C/C++ applications	409
Running an application under z/OS batch	410
Specifying run-time options under z/OS batch	410
Specifying run-time options in the EXEC statement	411
Using cataloged procedures	411
Running an application under TSO	412
Specifying run-time options under TSO	413
Passing arguments to the z/OS C/C++ application	413
Running an application under z/OS UNIX System Services	414
z/OS UNIX System Services Application environments	414
Specifying run-time options under z/OS UNIX System Services	415
Restriction on using 24-bit AMODE programs	415
Copying applications between a PDS and HFS	415
Running a data Set member from the z/OS Shell	415
Running z/OS UNIX System Services applications under z/OS batch	415

Part 4. Utilities and tools 419

Chapter 12. Object Library Utility	421
Creating an object library under z/OS batch	421
Creating an object library under TSO	422
Object Library Utility Map	424
Chapter 13. Filter Utility	433
CXXFILT options	434
SYMMAP NOSYMMAP	434
SIDEBYSIDE NOSIDEBYSIDE	434
WIDTH(width) NOWIDTH	434
REGULARNAME NOREGULARNAME	434
CLASSNAME NOCLASSNAME	435
SPECIALNAME NOSPECIALNAME	435
Unknown type of name	435
Under z/OS batch	435
Under TSO	436
Chapter 14. DSECT Conversion Utility	439
DSECT Utility options	439

SECT	439
BITF0XL NOBITF0XL	440
COMMENT NOCOMMENT	441
DECIMAL NODECIMAL.	441
DEFSUB NODEFSUB	442
EQUATE NOEQUATE	443
HDRSKIP NOHDRSKIP.	445
INDENT NOINDENT	445
LOCALE NOLOCALE	445
LOWERCASE NOLOWERCASE	446
LP64 NOLP64	446
OPTFILE NOOPTFILE	446
PPCOND NOPPCOND	446
SEQUENCE NOSEQUENCE.	447
UNIQUE NOUNIQUE	447
UNNAMED NOUNNAMED	448
OUTPUT	448
RECFM	448
LRECL	448
BLKSIZE	448
Generation of structures	448
Under z/OS batch	451
Under TSO	452
Chapter 15. Coded Character Set and Locale Utilities	453
Coded Character Set Conversion Utilities.	453
iconv Utility	453
genxlt Utility	455
localedef Utility	457

Part 5. z/OS UNIX System Services utilities 463

Chapter 16. Archive and Make Utilities	465
Archive libraries	465
Creating archive libraries.	465
Creating makefiles	466
Makedepend Utility	466
Chapter 17. BPXBATCH Utility	467
BPXBATCH usage	467
Parameter	468
Usage notes	469
Files	469
Chapter 18. c89 — Compiler invocation using host environment variables	471
Format	471
Description	471
Options	473
Operands	483
Environment variables.	486
Files	504
Usage notes	505
Localization.	511
Exit values	511
Portability	511
Related information.	512

	Chapter 19. xlc — Compiler invocation using a customizable	
	configuration file	513
	Format	513
	Description	513
	Invocation commands	514
	Setting up the compilation environment	515
	Environment variables	515
	Setting up a configuration file	516
	Configuration file attributes	517
	Tailoring a configuration file	519
	Default configuration file	519
	Invoking the compiler	522
	Invoking the binder	523
	Supported options	523
	–q options syntax	523
	Flag options syntax	525
	Specifying compiler options	528
	Specifying compiler options on the command line.	529
	Specifying flag options	529
	Specifying compiler options in a configuration file	530
	Specifying compiler options in your program source files	530
	Specifying compiler options for architecture-specific 32-bit or 64-bit compilation	530

Part 6. Appendixes 533

	Appendix A. Prelinking and linking z/OS C/C++ programs	535
	Restrictions on using the prelinker	535
	Prelinking an application	535
	Using DD Statements for the standard data sets - prelinker	536
	Input to the prelinker	538
	Prelinker output	538
	Mapping long names to short names	539
	Linking an application	540
	Using DD statements for standard data sets—linkage editor	540
	Input to the linkage editor	541
	Output from the linkage editor	542
	Link-editing multiple object modules.	543
	Building DLLs	544
	Linking your code	545
	Using DLLs.	545
	Prelinking and linking an application under z/OS batch and TSO	549
	z/OS Language Environment Prelinker Map	550
	Processing the prelinker automatic library call	555
	References to currently undefined symbols (external references)	555
	Prelinking and linking under z/OS batch	555
	Writing JCL for the prelinker and linkage editor.	557
	Secondary input to the linker	558
	Using additional input object modules under z/OS batch	559
	Under TSO.	559
	Using CPLINK.	562
	Using LINK.	564
	Prelinking and link-editing under the z/OS Shell	566
	Using your JCL	566
	Setting c89 to invoke the prelinker	568
	Using the c89 utility.	568

Prelinker control statement processing	568
IMPORT control statement	569
INCLUDE control statement	569
LIBRARY control statement	570
RENAME control statement	571
Reentrancy	572
Natural or constructed reentrancy	572
Using the prelinker to make your program reentrant	572
Steps for generating a reentrant load module in C	573
Steps for generating a reentrant load module in C++	574
Resolving multiple definitions of the same template function	574
External variables	575
Appendix B. Prelinker and linkage editor options	577
Prelinker options	577
DLLNAME(dll-name)	577
DUP NODUP	577
DYNAM NODYNAM	577
ER NOER	577
MAP NOMAP	578
MEMORY NOMEMORY	578
NCAL NONCAL	578
OMVS NOOMVS	578
UPCASE NOUPCASE	579
Linkage editor options	579
Appendix C. Diagnosing problems	581
Problem checklist	581
When does the error occur?	582
Steps for problem diagnosis using optimization levels	583
Steps for diagnosing errors that occur at compile time	584
Steps for diagnosing errors that occur at IPA Link time	585
The error occurs at bind time	586
The error occurs at prelink time	586
The error occurs at link time	587
Steps for diagnosing errors that occur at run time	587
Steps for avoiding installation problems	589
Appendix D. Cataloged procedures and REXX EXECs	591
Tailoring PROCs, REXX EXECs, and EXECs	593
Data sets used	594
Description of data sets used	595
Examples using cataloged procedures	601
Other z/OS C utilities	601
Using the old syntax for CC	601
Using CMOD	602
Appendix E. Calling the Compiler from Assembler	605
Example of using the Assembler ATTACH macro (CCNUAAP)	607
Example of JCL for the Assembler ATTACH macro (CCNUAAQ)	609
Example of using the Assembler LINK macro (CCNUAAR)	610
Example of JCL for the Assembler LINK macro (CCNUAAS)	612
Example of using the Assembler CALL macro (CCNUAAT)	613
Example of JCL for Assembler CALL macro (CCNUAAU)	615
Appendix F. Layout of the Events file	617

Description of the Fileid field	617
Description of the Filend field	618
Description of the Error field	618
Appendix G. Accessibility	621
Accessibility	621
Using assistive technologies	621
Keyboard navigation of the user interface.	621
z/OS information	621
Notices	623
Programming interface information	624
Trademarks.	624
Standards	625
Glossary	627
Bibliography	655
z/OS	655
z/OS C/C++	655
z/OS Run-Time Library Extensions	655
Debug Tool	655
z/OS Language Environment	655
Assembler	656
COBOL	656
PL/I	656
VS FORTRAN.	656
CICS	656
DB2	657
IMS/ESA.	657
QMF	657
DFSMS	657
INDEX	659

About this document

This edition of *z/OS C/C++ User's Guide* is intended for users of the z/OS® or z/OS.e C/C++ compiler with the z/OS or z/OS.e 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 or z/OS.e operating systems. References to z/OS in this document refer to both z/OS and z/OS.e.

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

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

How to read syntax diagrams

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

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

Symbols

The following symbols may be displayed in syntax diagrams:

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

Syntax items

Syntax diagrams contain many different items. Syntax items include:

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

- Separators - a separator separates keywords, variables or operators. For example, a comma (,) is a separator.

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

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




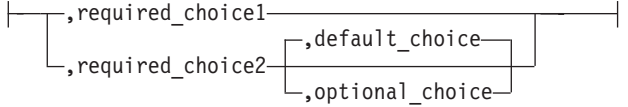
Syntax examples

The following table provides syntax examples.

Table 1. Syntax examples

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

Table 1. Syntax examples (continued)

Item	Syntax example
Variable.	
Variables appear in lowercase italics. They represent names or values.	
Repeatable item.	
An arrow returning to the left above the main path of the horizontal line indicates an item that can be repeated.	
An arrow returning to the left above a group of repeatable items indicates that one of the items can be selected, or a single item can be repeated.	
Fragment.	
The <code> fragment </code> symbol indicates that a labelled group is described below the main syntax diagram. Syntax is occasionally broken into fragments if the inclusion of the fragment would overly complicate the main syntax diagram.	<p data-bbox="829 699 954 724">fragment:</p> 

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 2. z/OS C/C++ publications

Document Title and Number	Key Sections/Chapters in the Document
<p><i>z/OS C/C++ Programming Guide</i>, SC09-4765</p>	<p>Guidance information for:</p> <ul style="list-style-type: none"> • 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[®] System Services applications • Reentrancy • Handling exceptions, error conditions, and signals • Performance optimization • Network communications under z/OS UNIX System Services • Interprocess communications using z/OS UNIX System Services • Structuring a program that uses C++ templates • Using environment variables • Using System Programming C facilities • Library functions for the System Programming C facilities • Using run-time user exits • Using the z/OS 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 System Services C/C++
<p><i>z/OS C/C++ User's Guide</i>, SC09-4767</p>	<p>Guidance information for:</p> <ul style="list-style-type: none"> • z/OS C/C++ examples • Compiler options • Binder options and control statements • Specifying Language Environment run-time options • Compiling, IPA Linking, binding, and running z/OS C/C++ programs • Utilities (Object Library, CXXFILT, DSECT Conversion, Code Set and Locale, ar and make, BPXBATCH) • Diagnosing problems • Cataloged procedures and REXX EXECs supplied by IBM
<p><i>z/OS C/C++ Language Reference</i>, SC09-4815</p>	<p>Reference information for:</p> <ul style="list-style-type: none"> • 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
<p><i>z/OS C/C++ Messages</i>, GC09-4819</p>	<p>Provides error messages and return codes for the compiler, and its related application interface libraries and utilities. For the C/C++ run-time library messages, refer to <i>z/OS Language Environment Run-Time Messages</i>, SA22-7566. For the c89 and xlc utility messages, refer to <i>z/OS UNIX System Services Messages and Codes</i>, SA22-7807.</p>

Table 2. z/OS C/C++ publications (continued)

Document Title and Number	Key Sections/Chapters in the Document
<i>z/OS C/C++ Run-Time Library Reference</i> , SA22-7821	Reference information for: <ul style="list-style-type: none"> • header files • library functions
<i>z/OS C Curses</i> , SA22-7820	Reference information for: <ul style="list-style-type: none"> • 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
<i>z/OS C/C++ Compiler and Run-Time Migration Guide for the Application Programmer</i> , GC09-4913	Guidance and reference information for: <ul style="list-style-type: none"> • 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
<i>Standard C++ Library Reference</i> , SC09-4949	The documentation describes how to use the following three main components of the Standard C++ Library to write portable C/C++ code that complies with the ISO standards: <ul style="list-style-type: none"> • ISO Standard C Library • ISO Standard C++ Library • Standard Template Library (C++) <p>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.</p>
<i>C/C++ Legacy Class Libraries Reference</i> , SC09-7652	Reference information for: <ul style="list-style-type: none"> • UNIX System Laboratories (USL) I/O Stream Library • USL Complex Mathematics Library <p>As of z/OS V1R6, this reference is part of the Run-Time Library Extensions documentation.</p>
<i>IBM Open Class Library Transition Guide</i> , SC09-4948	The documentation explains the various options to application owners and users for migrating from the IBM Open Class® library to the Standard C++ Library.
<i>z/OS Common Debug Architecture User's Guide</i> , SC09-7653	This documentation is the user's guide for IBM's libddpi library. It includes: <ul style="list-style-type: none"> • Overview of the architecture • Information on the order and purpose of API calls for model user applications and for accessing DWARF information • Information on using CDA with C/C++ source <p>This user's guide is part of the Run-Time Library Extensions documentation.</p>

Table 2. z/OS C/C++ publications (continued)

Document Title and Number	Key Sections/Chapters in the Document
<i>z/OS Common Debug Architecture Library Reference</i> , SC09-7654	This documentation is the reference for IBM's libddpi library. It includes: <ul style="list-style-type: none"> • General discussion of Common Debug Architecture • Description of APIs and data types related to stacks, processes, operating systems, machine state, storage, and formatting <p>This reference is part of the Run-Time Library Extensions documentation.</p>
<i>DWARF/ELF Extensions Library Reference</i> , SC09-7655	This documentation is the reference for IBM's extensions to the libdwarf and libelf libraries. It includes information on: <ul style="list-style-type: none"> • Consumer APIs • Producer APIs <p>This reference is part of the Run-Time Library Extensions documentation.</p>
Debug Tool documentation, available on the Debug Tool for z/OS and OS/390® library page on the World Wide Web	The documentation, which is available at http://www.ibm.com/software/awdtools/debugtool/library/ , provides guidance and reference information for debugging programs, using Debug Tool in different environments, and language-specific information.
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: <ul style="list-style-type: none"> • 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 Language Environment and using the Language Environment run-time options, refer to *z/OS Language Environment Programming Guide*, SA22-7561. For complete and detailed information on using interlanguage calls, refer to *z/OS Language Environment Writing Interlanguage Communication Applications*, SA22-7563.

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

Table 3. Publications by task

Tasks	Documents
Planning, preparing, and migrating to z/OS C/C++	<ul style="list-style-type: none"> • <i>z/OS C/C++ Compiler and Run-Time Migration Guide for the Application Programmer</i>, GC09-4913 • <i>z/OS Language Environment Customization</i>, SA22-7564 • <i>z/OS Language Environment Run-Time Application Migration Guide</i>, GA22-7565 • <i>z/OS UNIX System Services Planning</i>, GA22-7800 • <i>z/OS and z/OS.e Planning for Installation</i>, GA22-7504
Installing	<ul style="list-style-type: none"> • <i>z/OS Program Directory</i> • <i>z/OS and z/OS.e Planning for Installation</i>, GA22-7504 • <i>z/OS Language Environment Customization</i>, SA22-7564
Coding programs	<ul style="list-style-type: none"> • <i>z/OS C/C++ Run-Time Library Reference</i>, SA22-7821 • <i>z/OS C/C++ Language Reference</i>, SC09-4815 • <i>z/OS C/C++ Programming Guide</i>, SC09-4765 • <i>z/OS Language Environment Concepts Guide</i>, SA22-7567 • <i>z/OS Language Environment Programming Guide</i>, SA22-7561 • <i>z/OS Language Environment Programming Reference</i>, SA22-7562

Table 3. Publications by task (continued)

Tasks	Documents
Coding and binding programs with interlanguage calls	<ul style="list-style-type: none"> • <i>z/OS C/C++ Programming Guide</i>, SC09-4765 • <i>z/OS C/C++ Language Reference</i>, SC09-4815 • <i>z/OS Language Environment Programming Guide</i>, SA22-7561 • <i>z/OS Language Environment Writing Interlanguage Communication Applications</i>, SA22-7563 • <i>z/OS MVS Program Management: User's Guide and Reference</i>, SA22-7643 • <i>z/OS MVS Program Management: Advanced Facilities</i>, SA22-7644
Compiling, binding, and running programs	<ul style="list-style-type: none"> • <i>z/OS C/C++ User's Guide</i>, SC09-4767 • <i>z/OS Language Environment Programming Guide</i>, SA22-7561 • <i>z/OS Language Environment Debugging Guide</i>, GA22-7560 • <i>z/OS MVS Program Management: User's Guide and Reference</i>, SA22-7643 • <i>z/OS MVS Program Management: Advanced Facilities</i>, SA22-7644
Compiling and binding applications in the z/OS UNIX System Services environment	<ul style="list-style-type: none"> • <i>z/OS C/C++ User's Guide</i>, SC09-4767 • <i>z/OS UNIX System Services User's Guide</i>, SA22-7801 • <i>z/OS UNIX System Services Command Reference</i>, SA22-7802 • <i>z/OS MVS Program Management: User's Guide and Reference</i>, SA22-7643 • <i>z/OS MVS Program Management: Advanced Facilities</i>, SA22-7644
Debugging programs	<ul style="list-style-type: none"> • README file • <i>z/OS C/C++ User's Guide</i>, SC09-4767 • <i>z/OS C/C++ Messages</i>, GC09-4819 • <i>z/OS C/C++ Programming Guide</i>, SC09-4765 • <i>z/OS Language Environment Programming Guide</i>, SA22-7561 • <i>z/OS Language Environment Debugging Guide</i>, GA22-7560 • <i>z/OS Language Environment Run-Time Messages</i>, SA22-7566 • <i>z/OS UNIX System Services Messages and Codes</i>, SA22-7807 • <i>z/OS UNIX System Services User's Guide</i>, SA22-7801 • <i>z/OS UNIX System Services Command Reference</i>, SA22-7802 • <i>z/OS UNIX System Services Programming Tools</i>, SA22-7805 • <i>z/OS Common Debug Architecture User's Guide</i>, SC09-7653 • <i>z/OS Common Debug Architecture Library Reference</i>, SC09-7654 • <i>DWARF/ELF Extensions Library Reference</i>, SC09-7655 • Debug Tool documentation, available on the Debug Tool Library page on the World Wide Web (http://www.ibm.com/software/awdtools/debugtool/library/) • z/OS Messages Database, available on the z/OS Library page at http://www.ibm.com/servers/eserver/zseries/zos/bkserv/
Using shells and utilities in the z/OS UNIX System Services environment	<ul style="list-style-type: none"> • <i>z/OS C/C++ User's Guide</i>, SC09-4767 • <i>z/OS UNIX System Services Command Reference</i>, SA22-7802 • <i>z/OS UNIX System Services Messages and Codes</i>, SA22-7807
Using sockets library functions in the z/OS UNIX System Services environment	<ul style="list-style-type: none"> • <i>z/OS C/C++ Run-Time Library Reference</i>, SA22-7821
Using the ISO Standard C++ Library to write portable C/C++ code that complies with ISO standards	<ul style="list-style-type: none"> • <i>Standard C++ Library Reference</i>, SC09-4949
Migrating from the IBM Open Class Library to the Standard C++ Library	<ul style="list-style-type: none"> • <i>IBM Open Class Library Transition Guide</i>, SC09-4948

Table 3. Publications by task (continued)

Tasks	Documents
Porting a z/OS UNIX System Services application to z/OS	<ul style="list-style-type: none"> • <i>z/OS UNIX System Services Porting Guide</i> This guide contains useful information about supported header files and C functions, sockets in z/OS UNIX System Services, process management, compiler optimization tips, and suggestions for improving the application's performance after it has been ported. The <i>Porting Guide</i> 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.ibm.com/servers/eserver/zseries/zos/unix/bpxa1por.html
Working in the z/OS UNIX System Services Parallel Environment	<ul style="list-style-type: none"> • <i>z/OS UNIX System Services Parallel Environment: Operation and Use</i>, SA22-7810 • <i>z/OS UNIX System Services Parallel Environment: MPI Programming and Subroutine Reference</i>, SA22-7812
Performing diagnosis and submitting an Authorized Program Analysis Report (APAR)	<ul style="list-style-type: none"> • <i>z/OS C/C++ User's Guide</i>, SC09-4767 • CBC.SCCNDOC(APAR) on z/OS C/C++ product tape
Tuning Large C/C++ Applications on OS/390 UNIX System Services	<ul style="list-style-type: none"> • IBM Redbook called <i>Tuning Large C/C++ Applications on OS/390 UNIX System Services</i>, which is available at: http://www.redbooks.ibm.com/abstracts/sg245606.html
C/C++ Applications on z/OS and OS/390 UNIX	<ul style="list-style-type: none"> • IBM Redbook called <i>C/C++ Applications on z/OS and OS/390 UNIX</i>, which is available at: http://www.redbooks.ibm.com/abstracts/sg245992.html
Performance considerations for XPLINK	<ul style="list-style-type: none"> • IBM Redbook called <i>XPLink: OS/390 Extra Performance Linkage</i>, 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 535. 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*.

Softcopy documents

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 <http://www.ibm.com/software/awdtools/czos/library>.

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 at <http://www.adobe.com>.

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

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

Softcopy examples

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

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

- *z/OS C/C++ Programming Guide*, SC09-4765

In the following documents, 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. The labels have the form CCNxxyy or CLBxxyy, where x refers to a publication:

- R and X refer to *z/OS 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 CCNxxyy appear in *z/OS C/C++ Language Reference*, *z/OS C/C++ Programming Guide*, and *z/OS C/C++ User's Guide*. Examples labelled as CLBxxyy appear in *z/OS C/C++ User's Guide*.

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/awdtools/czos/>

This page contains late-breaking information about the z/OS C/C++ product, including the compiler, the class libraries, and utilities. 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 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 z/OS licensed documents on the Internet

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

<http://www.ibm.com/servers/resourceLink>

Licensed documents are available only to customers with a z/OS license. Access to these documents requires an IBM Resource Link user ID and password, and a key code. With your z/OS order you received a Memo to Licensees, (GI10-0671), that includes this key code.¹

To obtain your IBM Resource Link user ID and password, log on to:

<http://www.ibm.com/servers/resourceLink>

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

1. Sign in to Resource Link using your Resource Link user ID and password.
2. Select **User Profiles** located on the left-hand navigation bar.

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

Printed licensed documents are not available from IBM.

1. z/OS.e customers received a Memo to Licensees, (GI10-0684) that includes this key code.

You can use the PDF format on either **z/OS Licensed Product Library CD-ROM** or IBM Resource Link to print licensed documents.

Using LookAt to look up message explanations

LookAt is an online facility that lets you look up explanations for most of the IBM messages you encounter, as well as for some system abends and codes. Using LookAt to find information is faster than a conventional search because in most cases LookAt goes directly to the message explanation.

You can use LookAt from the following locations to find IBM message explanations for z/OS elements and features, z/VM[®], VSE/ESA[™], and Clusters for AIX[®] and Linux:

- The Internet. You can access IBM message explanations directly from the LookAt Web site at <http://www.ibm.com/eserver/zseries/zos/bkserv/lookat/>.
- Your z/OS TSO/E host system. You can install code on your z/OS or z/OS.e systems to access IBM message explanations, using LookAt from a TSO/E command line (for example, TSO/E prompt, ISPF, or z/OS UNIX System Services running OMVS).
- Your Microsoft[®] Windows[®] workstation. You can install code to access IBM message explanations on the *z/OS Collection* (SK3T-4269), using LookAt from a Microsoft Windows DOS command line.
- Your wireless handheld device. You can use the LookAt Mobile Edition with a handheld device that has wireless access and an Internet browser (for example, Internet Explorer for Pocket PCs, Blazer, or Eudora for Palm OS, or Opera for Linux handheld devices). Link to the LookAt Mobile Edition from the LookAt Web site.

You can obtain code to install LookAt on your host system or Microsoft Windows workstation from a disk on your *z/OS Collection* (SK3T-4269), or from the LookAt Web site (click **Download**, and select the platform, release, collection, and location that suit your needs). More information is available in the LOOKAT.ME files available during the download process.

Part 1. Introduction

This part discusses introductory concepts for the z/OS C/C++ product. Specifically, it discusses the following:

- Chapter 1, “About IBM z/OS C/C++,” on page 3
- “About prelinking, linking, and binding” on page 10

Chapter 1. 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.

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)
- 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

Notes:

1. The Run-Time Library Extensions base element was introduced in z/OS V1R5. It includes the Common Debug Architecture (CDA) Libraries, the c89 utility, and the x1c utility. The Common Debug Architecture provides a consistent and common format for debugging information across the various languages and operating systems that are supported on the IBM eServer™ zSeries® platform. Run-Time Library Extensions also includes legacy libraries to support existing programs. These are the UNIX System Laboratories (USL) I/O Stream Library, USL Complex Mathematics Library, and IBM Open Class DLLs. Application development using the IBM Open Class Library is not supported.
2. The Standard C++ Library is included with the Language Environment.
3. The z/OS C/C++ compiler works with the mainframe interactive Debug Tool product.

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

Changes for z/OS V1R6

z/OS V1R6 C/C++ supports compilation of 64-bit programs, which is enabled by the LP64 compiler option. z/OS C/C++ has made the following performance and usability enhancements for the V1R6 release:

New compiler suboptions

z/OS V1R6 C/C++ introduces the following new compiler suboptions:

- ARCH(6)
- TARGET(zOSV1R6)
- TUNE(6)

New cataloged procedures

z/OS V1R6 C/C++ introduces the following new cataloged procedures:

- CBCQB - to bind a 64-bit C++ program
- CBCQBG - to bind and run a 64-bit C++ program
- CBCQCB - to compile and bind a 64-bit program
- CBCQCBG - to compile, bind, and run a 64-bit program
- CCNQP1B - to bind the results of IPA(LINK,PDF1) for 64-bit applications
- EDCQB - to bind 64-bit C programs

- EDCQBG - to bind and run 64-bit C programs
- EDCQCB - to compile and bind a 64-bit C program
- EDCQCBG - to compile, bind, and run a 64-bit C program

New keyword z/OS V1R6 C/C++ introduces the following new keyword, which is used in a declaration to specify an alignment for a declared variable:

- `__attribute__((aligned(n)))`

For further information on this keyword, see <http://gcc.gnu.org/onlinedocs>.

New xlc compiler invocation utility

This release includes the new xlc compiler invocation utility, which supports the following invocation commands that accept AIX option syntax:

- `c89` - to compile ANSI compliant C programs
- `cc` - to compile non-standard C programs
- `xlc` - to compile any C programs

It also includes the following invocation commands, which are used to compile C++ code:

- `c++`
- `cxx`
- `xlc`
- `xlc++`

All of the invocation commands listed above can have the following suffixes:

- `_x` - to compile the program with XPLINK (for example, `c89_x`)
- `_64` - to compile the program in 64-bit mode (for example, `c89_64`)

For z/OS Version 1 Release 6, the Language Environment provides the following:

64-bit virtual addressing mode support

With 64-bit virtual storage support, the maximum virtual addressability is up to 16 exabytes. For programs that need large data caches, migrating to 64-bit virtual addressing mode (AMODE 64) is reasonable. This enables applications to access storage and helps in porting applications from other platforms.

Language Environment provides 64-bit virtual support for XPLINK applications only. Applications compiled for AMODE 64 will run in this new form of Language Environment. Mixing of AMODE 31 and AMODE 64 applications is not supported.

Assembler dynamic link libraries (DLL) support

This support allows Language Environment-conforming assembler applications to create and use DLLs.

C/C++ run-time enhancements

These enhancements include:

- The `dlopen()` family of functions is provided.
- Enhanced ASCII support is added for `__getenv()`, `rexec()`, and `rexec_af()`.

- The `popen()` function can now use `fork()` for `spawn()` based on the setting of the `_EDC_POOPEN` environment variable.
- The `snprintf()` family of functions is provided.

Support for Euro Phase III and G11N currency

Phase III of Euro support in locales is provided for National Language Standards.

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, generic types or templates, and an extensive library. For a detailed description of the differences between z/OS C++ and z/OS C, refer to *z/OS 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 z/Series 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/C++ compiler always uses the constructed reentrancy algorithms.

- 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 System Services technology.

z/OS UNIX System Services 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, Networking Services, Issue 4*

- 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 Languages, 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 V1R6 C/C++ uses the following thread-safe class libraries that are available with z/OS:

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

Note: Starting with z/OS V1R5, all application development using the C/C++ IBM Open Class Library (Application Support Class and Collection Class Libraries) is not supported. Run-time support for the execution of existing applications, which use the IBM Open Class, is provided with z/OS V1R6 but is planned to be removed in a future release. For additional information, see *IBM Open Class Library Transition Guide*.

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

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

Utilities

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

- The `x1c` utility to invoke the compiler using a customizable configuration file.
- The `c89` utility to invoke the compiler using host environment variables.
- The `CXXFILT` utility to map z/OS C++ mangled names to the original source.
- The `DSECT` Conversion Utility to convert descriptive assembler DSECTs into z/OS C/C++ data structures.

- 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 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.

dbx

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

Please refer to <http://www.ibm.com/servers/eserver/zseries/zos/unix/bpxa1dbx.html> for further information on dbx.

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.

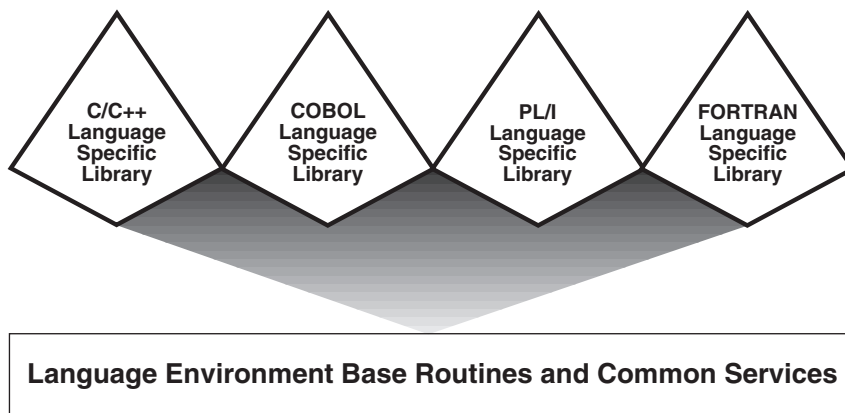


Figure 1. Libraries in z/OS Language Environment

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 includes 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 System 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.

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

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 *z/OS Language Environment Programming Guide*, this support enables you to develop applications on higher release levels of z/OS for use on platforms that are running lower release levels of z/OS. In C and C++, downward compatibility support is provided through the C/C++ TARGET compiler option. See “TARGET” on page 197 for details on this compiler option.

For example, a company may use z/OS V1R6 with Language Environment on a development system where applications are coded, link-edited, and tested, while using any supported lower release of 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 z/OS where the application will be used.

The downward compatibility support includes toleration PTFs for lower releases of 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 diagnosis assistance that will be provided by the toleration PTFs includes detection of an unsupported program object format. If the program object format is at a level which is not supported by the target deployment system, then the deployment system will produce an abend when trying to load the application program. The abend will indicate that DFSMS was unable to find or load the application program. Correcting this problem does not require the installation of any toleration PTFs. Instead, the application developer will need to recreate the program object which is compatible with the older deployment system.

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 z/OS Language Environment will continue to run on later releases of z/OS Language Environment without the need to recompile or re-link edit the application, independent of the downward compatibility support.

The current z/OS 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).

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

About prelinking, linking, and binding

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

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

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

- With respect to building an application:

In both instances, the program management binder is performing the actual processing of converting the object file(s) into the application executable module. Object files with longname symbols, reentrant writable static symbols, and DLL-style function calls require additional processing to build global data for the application.

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

- The processing is not required, because none of the object files in the application use constructed reentrancy, use long names, are DLL or are C++.
- The processing is handled by executing the prelinker step before running the binder.

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

- With respect to executing code in an application:

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

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

Notes on the prelinking process

Note that you cannot use the prelinker if you are using the XPLINK, G0FF, or LP64 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 document discussed concepts of *prelinking* and *linking*. The prelinker was designed to process long names and support constructed reentrancy in earlier versions of the C compiler 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. In particular, the binder supports:

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

Note: You need to use the binder for 64-bit objects.

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 9, “Binding z/OS C/C++ programs,” on page 355.

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

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).

z/OS UNIX System Services

z/OS UNIX System Services provides capabilities under z/OS to make it easier to implement or port applications in an open, distributed environment. z/OS UNIX System 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 System Services provides support for both existing z/OS applications and new z/OS UNIX System Services 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 System Services extensions that provide z/OS-specific support beyond the defined standards
- The z/OS UNIX System Services 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	Uses host environment variables to compile, assemble, and bind z/OS UNIX System Services 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
xlc	Allows you to invoke the compiler using a customizable configuration file
yacc	Allows you to write compilers and other programs that parse input according to strict grammar rules

– Support for other utilities such as:

dspcat	Displays all or part of a message catalog
dspmsg	Displays a selected message from a message catalog
mkcatdefs	Preprocesses a message source file for input to the gencat utility
runcat	Invokes mkcatdefs and pipes the message catalog source data (the output from mkcatdefs) to gencat

- Access to a hierarchical file system (HFS), with support for the POSIX.1 and XPG4 standards
- Access to zServer File System (zFS), which provides performance improvements over HFS
- 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 System Services offers program portability across multivendor operating systems, with support for POSIX.1, POSIX.1a (draft 6), POSIX.2, POSIX.4a (draft 6), and XPG4.2.

For application developers who have worked with other UNIX environments, the z/OS UNIX System Services 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 System Services 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 System Services C/C++ functions

All z/OS UNIX System Services C functions are available at all times. In some situations, you must specify the `POSIX(0N)` 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 System Services 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(0N)` 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 Standard C++ 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 System Services provides support for an enhanced version of an industry-accepted protocol for client/server communication that is known as *sockets*. A set of C language functions provides support for z/OS UNIX System Services sockets. z/OS UNIX System Services 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 System Services socket application program interface (API) provides support for both UNIX domain sockets and Internet domain sockets. UNIX domain sockets, or *local sockets*, allow interprocess communication within z/OS, independent of TCP/IP. Local sockets behave like traditional UNIX sockets and allow processes to communicate with one another on a single system. With Internet sockets, application programs can communicate with each other in the network using TCP/IP.

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

File types

In addition to conventional files, such as sequential files and partitioned data sets, the 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 *Performing VSAM I/O operations in 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. See *Opening Files in z/OS C/C++ Programming Guide* for the details of these rules. You can create regular HFS files, special character HFS files, or FIFO HFS files. You can also create links or directories.

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).

zServer File System

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

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 (see large file support in *z/OS C/C++ Language Reference*)
- User error handling for serious I/O failures (SIGIOERR)
- Improved sequential data access performance through enablement of the DFSMS 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 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:

- Customer Information Control System (CICS)
You can use the CICS Command-Level Interface to write C/C++ application programs. The CICS Command-Level Interface provides data, job, and task management facilities that are normally provided by the operating system.
- 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.

Note: Alternatively, use the SQL compiler option to compile a DB2 program without using the DB2 preprocessor.

The DB2 program processes requests, then returns control to the application program.

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

- Frequency analysis
- Dynamic patching

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

You can specify either data sets or Hierarchical File System (HFS) files as source files.

For further information, see <http://www.ibm.com/software/awdtools/debugtool/>.

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

Note: Starting with z/OS V1R5, both the z/OS C/C++ compiler optional feature and the Debug Tool product will need to be installed if you wish to use IBM C/C++ Productivity Tools for OS/390. For more information on Debug Tool, refer to <http://www.ibm.com/software/awdtools/debugtool/>.

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

- 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 breakpoint with the simple click of the mouse. Use the windowing capabilities of your workstation to view multiple segments of your source and your storage, while monitoring a variable at the same time.

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

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.

- Fault Analyzer for z/OS and OS/390

The IBM Fault Analyzer helps developers analyze and fix application and system failures. It gathers information about an application and the surrounding environment at the time of the abend, providing the developer with valuable information needed for developing and testing new and existing applications. For

more information, please refer to:
<http://www.ibm.com/software/awdtools/faultanalyzer/>

- Application Monitor for z/OS and OS/390

The IBM Application Monitor provides resource utilization information for your applications. This resource information can be the current system data (online analysis) or data collected over a certain time period (historical analysis). It helps you to isolate performance problems in applications, improve response time in online transactions and improve batch turnaround time. It also collects samples from the monitored address space and analyzes the system or resource application. For more information please refer to:
<http://www.ibm.com/software/awdtools/applicationmonitor/>

- Software Configuration and Library Manager facility (SCLM)

The ISPF Software Configuration and Library Manager facility (SCLM) maintains information about the source code, objects and load modules. It also keeps track of other relationships in your application, such as test cases, JCL, and publications. The SCLM Build function translates input to output, managing not only compilation and linking, but all associating processes required to build an application. This facility helps to ensure that your production load modules match the source in your production source libraries.

- 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 JNI, which is supported by both compiled and interpreted Java code. Calls to C or C++ do not distinguish between these two.

Additional features of z/OS 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.

Feature	Description
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 <code>wcstombs()</code> , <code>mbstowcs()</code> , <code>wcsrtoombs()</code> , and <code>mbsrtowcs()</code> , as well as the family of wide-character I/O functions. Wide-character data can be represented by the <code>wchar_t</code> data type.
Extended Precision Floating-Point Numbers	<p>z/OS C/C++ provides three S/390 floating-point number data types: single precision (32 bits), declared as <code>float</code>; double precision (64 bits), declared as <code>double</code>; and extended precision (128 bits), declared as <code>long double</code>.</p> <p>Extended precision floating-point numbers give greater accuracy to mathematical calculations.</p> <p>As of OS/390 V2R6, C/C+ also supports IEEE 754 floating-point representation. By default, <code>float</code>, <code>double</code>, and <code>long double</code> values are represented in IBM S/390 floating point format. However, the IEEE 754 floating-point representation is used if you specify the <code>FLOAT(IEEE754)</code> compiler option. For details on this support, see “FLOAT” on page 105.</p>
Command Line Redirection	You can redirect the standard streams <code>stdin</code> , <code>stderr</code> , and <code>stdout</code> from the command line or when calling programs using the <code>system()</code> 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.
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 <code>iconv</code> utility converts data or source from one code page to another.
Selected Built-in Library Functions	For selected library functions, the compiler generates an instruction sequence directly into the object code during optimization to improve execution performance. String and character functions are examples of these built-in functions. No actual calls to the library are generated when built-in functions are used.
Multi-threading	Threads are efficient in applications that allow them to take advantage of any underlying parallelism available in the host environment. This underlying parallelism in the host can be exploited either by forking a process and creating a new address space, or by using multiple threads within a single process. For more information, refer to <i>Using Threads in z/OS UNIX Applications in z/OS C/C++ Programming Guide</i> .
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 a z/OS C program or to define structures that are laid out according to COBOL or PL/I structure alignment rules.
Fixed-point (Packed) Decimal Data	z/OS 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 <code>COMP-3</code> or the PL/I data type <code>FIXED DEC</code> , with up to 31 digits of precision.
Long Name Support	For portability, external names can be mixed case and up to 32 K - 1 characters in length. For C++, the limit applies to the mangled version of the name.
System Calls	You can call commands or executable modules using the <code>system()</code> function under z/OS, z/OS UNIX System Services, and TSO. You can also use the <code>system()</code> function to call EXECs on z/OS and TSO, or Shell scripts using z/OS UNIX System Services.

Feature	Description
Exploitation of Hardware	<p>Use the ARCHITECTURE compiler option to select the minimum level of machine architecture on which your program will run. Note that certain features provided by the compiler require a minimum architecture level. The highest level currently supported is ARCH(6), which exploits instructions available on model 2084-xxx (z/900) in z/Architecture™ mode. For more information, refer to “ARCHITECTURE” on page 70.</p>
	<p>Use the TUNE compiler option to optimize your application for a specific machine architecture within the constraints imposed by the ARCHITECTURE option. The TUNE level must not be lower than the setting in the ARCHITECTURE option. For more information, refer to “TUNE” on page 212.</p>
Built-in Functions for Floating-Point and Other Hardware Instructions	<p>Use built-in functions for floating-point and other hardware instructions that are otherwise inaccessible to C/C++ programs. For more information, see the appendix on built-in functions in <i>z/OS C/C++ Programming Guide</i>.</p>

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 2, "z/OS C example," on page 23
- Chapter 3, "z/OS C++ examples," on page 29
- Chapter 4, "Compiler Options," on page 43
- Chapter 5, "Binder options and control statements," on page 285
- Chapter 6, "Run-Time options," on page 287

Chapter 2. 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 7, “Compiling,” on page 291, Chapter 9, “Binding z/OS C/C++ programs,” on page 355, and Chapter 11, “Running a C or C++ application,” on page 409 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 535.

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 324.
- 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 *z/OS C/C++ Language Reference* and *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. For more information, see Chapter 7, “Compiling,” on page 291, Chapter 9, “Binding z/OS C/C++ programs,” on page 355, and Chapter 11, “Running a C or C++ application,” on page 409.

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

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

Under z/OS batch

Copy the IBM-supplied sample program and header file into your data set. For example, if your user prefix is `PETE`, store the sample program (`CCNUAAM`) in `PETE.TEST.C(CTOF)` and the header file in `PETE.TESTHDR.H(CCNUAAM)`. 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 on page 26 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).

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

Before you begin: You need to have ensured that the z/OS Language Environment run-time libraries SCEERUN and SCEERUN2, and the z/OS C compiler are in the STEPLIB, LPALST, or LNKST concatenation.

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

1. 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 303.

-
2. Perform a bind:

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

```
-- or, for XPLINK --
```

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

CXXBIND binds the object module PETE.TEST.C.OBJ(CTOF) to create an executable module CTOF in the PDSE PETE.TEST.C.LOAD, with the default bind options. See Chapter 9, “Binding z/OS C/C++ programs,” on page 355 for more information.

-
3. Run the program:

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

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

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

CALL runs CTOF from PETE.TEST.C.LOAD with the default run-time options in effect. See Chapter 11, “Running a C or C++ application,” on page 409 for more information.

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

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

Before you begin: You need to have put the source in HFS and you need to have ensured 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.

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

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

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

2. Compile and bind:

```
c89 -o ctof ccnuaam.c

-- or, for XPLINK --

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

3. Run the program:

```
./ctof
```

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

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

Chapter 3. z/OS 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 7, “Compiling,” on page 291, Chapter 9, “Binding z/OS C/C++ programs,” on page 355, and Chapter 11, “Running a C or C++ application,” on page 409 for a detailed description on compiling, binding, and running a 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/cbclib/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 two files. File CCNUBRH contains the classes that are used in the main program. File CCNUBRC contains the remaining source code. The example files CCNUBRC and CCNUBRH are shipped with the z/OS 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 *z/OS C/C++ Language Reference* or *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  
  
class Date {  
public:  
    Date();  
    int DaysSince(const char *date);  
  
protected:  
    int curYear, curDay;  
    static const int dateLen = 10;  
    static const int numMonths = 12;  
    static const int numDays[];  
};  
  
class BirthDate : public Date {  
public:  
    BirthDate();  
    BirthDate(const char *birthText);  
    int DaysOld() { return(DaysSince(text)); }  
  
private:  
    char text[Date::dateLen+1];  
};
```

Figure 6. Header file for the biorhythm example (Part 1 of 2)


```

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

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

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

```

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

int main(void) {

    BioRhythm bio;
    int code;

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

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

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

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

    return(os);
}
```

Figure 7. z/OS C++ Biorhythm example program (Part 1 of 3)

```

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

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

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

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

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

Date::DaysSince(const char *text) {

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

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

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

```

Figure 7. z/OS C++ Biorhythm example program (Part 2 of 3)

```

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

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

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

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

```

Figure 7. z/OS C++ Biorhythm example program (Part 3 of 3)

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. For more information, see Chapter 7, “Compiling,” on page 291, Chapter 9, “Binding z/OS C/C++ programs,” on page 355, and Chapter 11, “Running a C or C++ application,” on page 409.

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

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

Under z/OS batch

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

```

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

```

Figure 8. JCL to compile, bind, and run the example program 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)',
// CPARAM='OPTFILE(DD:CCOPT)'
//COMPILE.CCOPT DD *
// LSEARCH('PETE.TESTHDR.H')
// SEARCH('CEE.SCEEH.+','CBC.SCLBH.+')
/*
/** ENTER A DATE IN THE FORM YYYY/MM/DD
//GO.SYSIN DD *
// 1997/10/19
/*

```

Figure 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 7, “Compiling,” on page 291, Chapter 9, “Binding z/OS C/C++ programs,” on page 355, and Chapter 11, “Running a C or C++ application,” on page 409.

Non-XPLINK and XPLINK under TSO

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

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

Before you begin: You need to have ensured 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.

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

1. 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 303.

2. Bind:

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

-- or, for XPLINK --

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

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

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

For more information see Chapter 9, “Binding z/OS C/C++ programs,” on page 355.

3. Run the program:

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

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

Result: The following information displays:

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

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

For more information see “Running an application under TSO” on page 412.

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

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

Before you begin: You need to have put the source in 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.

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.

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

1. Compile and bind:

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

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

2. Run the program:

```
./bio
```

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

Result: The following information displays:

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

Example of a C++ template program

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

The following example shows a simple z/OS C++ program that uses templates to perform simple operations on linked lists. It resides in HFS in the directory `/usr/lpp/cbclib/sample/clb3atmp`. The main program, `CLB3ATMP.CXX` (see “CLB3ATMP.CXX” on page 38), uses three header files that are from the Standard C++ Library: `list`, `string`, and `iostream`. It has one class template: `list`.

CLB3ATMP.CXX

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

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

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

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

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

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

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

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

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

Figure 10. z/OS C++ template program (Part 1 of 2)


```

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

return(0);
}

```

Figure 10. 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

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

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

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

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

CCNUNCL

```
//Jobcard info
//PROC JCLLIB ORDER=(CBC.SCCNPRC,
//  CEE.SCEEPROC)
//*
//* Compile MAIN program,creating an object deck
//*
//MAINCC EXEC CBCCL,
//  OUTFILE='<userhlq>.SAMPLE.OBJ(CLB3ATMP),DISP=SHR ',
//  CPARAM='XPLINK,OPTF(DD:COPTS)'
//SYSIN DD PATH='/usr/lpp/cbc1ib/sample/clb3atmp/clb3atmp.cpp'
//COPTS DD *
//  SEARCH('CEE.SCEEH.+','CBC.SCLBH.+')
//*
//*
//* Bind the program
//*
//BIND EXEC CBCXB,
//  INFILE='<userhlq>.SAMPLE.OBJ(CLB3ATMP)',
//  OUTFILE='<userhlq>.SAMPLE.LOAD(CLB3ATMP),DISP=SHR'
//*
//* Run the program
//*
//GO EXEC CBCXG,
//  INFILE='<userhlq>.SAMPLE.LOAD',
//  GOPGM=CLB3ATMP
//GO.SYSIN DD *
//  1 2 5 3 7 8 3 2 10 11
//*
```

Figure 11. JCL to compile, bind and run the template example

Under TSO

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

Before you begin: You need to have ensured 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 STEPLIB, LPALST, or the LNKLST concatenation.

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

1. Compile the source files:

- a. `cxx /usr/lpp/cbclib/sample/clb3atmp.cpp
(lsearch(/usr/lpp/cbclib/sample) search('cee.sceeh.+','cbc.sclbh.+')
obj(sample.obj(clb3atmp)) tempinc(//tempinc)`

This step compiles CLB3ATMP with the default compiler options, and stores the object module in *userhlq*.SAMPLE.OBJ(CLB3ATMP), where *userhlq* is your user prefix. The template instantiation files are written to the PDS *userhlq*.TEMPINC.

- b. `cxx TEMPINC (lsearch(/usr/lpp/cbclib/sample)
search('cee.sceeh.+','cbc.sclbh.+')`

This step compiles the PDS TEMPINC and creates the corresponding objects in the PDS *userhlq*.TEMPINC.OBJ.

See “Compiling under TSO” on page 303 for more information.

2. Create a library from the PDS *userhlq*.TEMPINC.OBJ:

```
C370LIB DIR LIB(TEMPINC.OBJ)
```

For more information see “Creating an object library under TSO” on page 422

3. Bind the program:

```
CXXBIND OBJ(SAMPLE.OBJ(CLB3ATMP)) LIB(TEMPINC.OBJ) LOAD(SAMPLE.LOAD(CLB3ATMP))
```

This step binds the *userhlq*.SAMPLE.OBJ(CLB3ATMP) text deck using the *userhlq*.TEMPINC.OBJ library and the default bind options. It also creates the executable module *userhlq*.SAMPLE.LOAD(CLB3ATMP).

Note: To avoid a binder error, the data set *userhlq*.SAMPLE.LOAD must be a PDSE.

For more information see “Binding under TSO using CXXBIND” on page 377.

4. Run the program:

```
CALL SAMPLE.LOAD(CLB3ATMP)
```

This step executes the module *userhlq*.SAMPLE.LOAD(CLB3ATMP) using the default run-time options. For more information see “Running an application under TSO” on page 412.

Under the z/OS UNIX shell

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

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

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

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

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

2. Then, to compile and bind:

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

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

3. Run the program:

```
./clb3atmp
```

Chapter 4. Compiler Options

This 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 7, “Compiling,” on page 291, and Appendix D, “Cataloged procedures and REXX EXECs,” on page 591 for details.

- In your own JCL procedure, by passing a parameter string to the compiler.
- In an options file. See “OPTFILE | NOOPTFILE” on page 167 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 47 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 `c89` utility, by using the `-wc` option to pass options to the compiler.
- In the `x1c` utility, by using the `-q` option or the `-wc` option to pass options to the compiler.

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

AGGRCOPY	
ALIAS	(C compile only)
ANSIALIAS	
ARCHITECTURE	
ARGPARSE	
ASCII	
BITFIELD	
CHARS	
COMPACT	
COMPRESS	
CONVLIT	
CSECT	
CVFT	(C++ compile only)
DEBUG	
DLL	
EXECOPS	
EXPORTALL	
FLOAT	
GOFF	
GONUMBER	
IGNERRNO	
ILP32	

INITAUTO	
INLINE	
IPA	
LANGLVL	
LIBANSI	
LOCALE	
LONGNAME	
LP64	
MAXMEM	
OBJECTMODEL	(C++ compile only)
OPTIMIZE	
PLIST	
REDIR	
RENT	(C compile only)
ROCONST	
ROSTRING	
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	
UNROLL	
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. Before you use the IPA compiler option, refer to an overview of IPA in *z/OS C/C++ Programming Guide*.

Applicability of compiler options under IPA

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

- Many compiler options do not have any special effect on IPA. For example, the PPOONLY 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 4 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 4. 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: ¹An 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 Linkstep, 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.

While the c89 utility uses compiler options, which have parentheses, x1c uses the -q syntax, which does not use parentheses and is more convenient for shell invocation.

See Chapter 18, “c89 — Compiler invocation using host environment variables,” on page 471 and Chapter 19, “x1c — Compiler invocation using a customizable configuration file,” on page 513 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,C Parm='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=CCNDVR,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 *z/OS 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 System Services

The c89 and x1c utilities invoke the z/OS C/C++ compiler with a set of compiler options.

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, x1c, and related utilities, refer to *z/OS UNIX System Services Command Reference*, Chapter 18, “c89 — Compiler invocation

using host environment variables,” on page 471 or Chapter 19, “xlc — Compiler invocation using a customizable configuration file,” on page 513.

For compiler options that take file names as suboptions, you can specify a sequential data set, a partitioned data set, or a partitioned data set member by prefixing the name with two slashes (//). The rest of the name follows the same syntax rule for naming data sets. Names that are not preceded with two slashes are 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`, and `xlc` utilities that run in the z/OS UNIX System Services shell specify certain compiler options in order to support POSIX standards. For a complete description of these utilities, refer to Chapter 18, “c89 — Compiler invocation using host environment variables,” on page 471, Chapter 19, “xlc — Compiler invocation using a customizable configuration file,” on page 513, 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 System Services utilities overrides the setting of the same option if it is specified using a `#pragma options` directive. The exception is CSECT, where the `#pragma csect` directive takes precedence.

In effect, invoking the compiler with the `c89`, and `xlc` utilities overrides the default values for many options, compared to running the compiler in MVS batch or TSO. For example, the `c89` utility specifies the RENT option, while the compiler default in MVS batch or TSO is NORENT. Any overrides of the defaults by the `c89`, and `xlc` utilities are noted in the DEFAULT category for the option. As the compiler defaults can always be changed during installation, you should always consult the compiler listing to verify the values passed to the compiler. See “Using the z/OS C Compiler Listing” on page 223 and “Using the z/OS C++ Compiler Listing” on page 257 for more information.

The `c89` utility remaps the following options to the values shown. Note that these values are set for a regular (non-IPA) compile. These values will change if you invoke IPA Compile, IPA link, or specify certain other options. For example, specifying the `c89 -V` option changes the settings of many of the compiler listing options. See Chapter 18, “c89 — Compiler invocation using host environment variables,” on page 471 or Chapter 19, “xlc — Compiler invocation using a

customizable configuration file,” on page 513 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 Chapter 18, “c89 — Compiler invocation using host environment variables,” on page 471 for more information.
- z/OS UNIX System Services 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 486 in Chapter 18, “c89 — Compiler invocation using host environment variables,” on page 471 for more information.

Refer to “SEARCH | NOSEARCH” on page 184 for more information on SEARCH.

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

Summary of compiler options

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

Note: For a description of the compiler options that can be specified with xlc, type xlc -help to access the help file.

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 5. Compiler options, abbreviations, and IBM supplied defaults

Compiler Option (Abbreviated Names are underlined)	IBM Supplied Default	C	C++	IPA Link	More Information
<u>AGGRCOPY</u> (<u>OVERLAP</u> <u>NOOVERLAP</u>)	AGGRC(NOOVERL)	✓	✓	✓	See 64
<u>AGGREGATE</u> <u>NOAGGREGATE</u>	NOAGG	✓		✓	See 65
<u>ALIAS</u> [(name)] <u>NOALIAS</u>	NOALI	✓			See 66
<u>ANSIALIAS</u> <u>NOANSIALIAS</u>	ANS	✓	✓	✓	See 67
<u>ARCHITECTURE</u> (n)	ARCH(5)	✓	✓	✓	See 70
<u>ARGPARSE</u> <u>NOARGPARSE</u>	ARG	✓	✓	✓	See 73
<u>ASCII</u> <u>NOASCII</u>	NOASCII	✓	✓	✓	See 74
<u>ATTRIBUTE</u> [(FULL)] <u>NOATTRIBUTE</u>	NOATT		✓	✓	See 75
<u>BITFIELD</u> (SIGNEDIUNSIGNED)	BITF(UNSIGNED)	✓	✓	✓	See 76
<u>CHARS</u> (SIGNED UNSIGNED)	CHARS(UNSIGNED)	✓	✓	✓	See 76
<u>CHECKOUT</u> (subopts) <u>NOCHECKOUT</u>	NOCHE	✓		✓	See 77
<u>COMPACT</u> <u>NOCOMPACT</u>	NOCOMPACT	✓	✓	✓	See 79

Table 5. Compiler options, abbreviations, and IBM supplied defaults (continued)

Compiler Option (Abbreviated Names are underlined)	IBM Supplied Default	C	C++	IPA Link	More Information
<u>COMPRESS</u> <u>NOCOMPRESS</u>	NOCOMPRESS	✓	✓	✓	See 81
<u>CONVLIT</u> [(subopts)] <u>NOCONVLIT</u> [(subopts)]	NOCONV (, NOWCHAR)	✓	✓	✓	See 82
<u>CSECT</u> [(qualifier)] <u>NOCSECT</u> [(qualifier)]	NOCSE for NOGOFF or CSE() for GOFF	✓	✓	✓	See 84
<u>CVFT</u> <u>NOCVFT</u>	CVFT		✓		See 86
<u>DBRMLIB</u> (filename)	DBRMLIB(DD:DBRMLIB)	✓	✓	✓	See “DBRMLIB” on page 88
<u>DEBUG</u> [(subopts)] <u>NODEBUG</u> [(subopts)]	NODEBUG	✓	✓		See “DEBUG NODEBUG” on page 88
<u>DEFINE</u> (name1[= =def1], name2[= =def2],...)	no default user definitions	✓	✓	✓	See 92
<u>DIGRAPH</u> <u>NODIGRAPH</u>	DIGR	✓	✓	✓	See 93
<u>DLL</u> (<u>CBA</u> <u>NOCBA</u>) <u>NODLL</u> (<u>CBA</u> <u>NOCBA</u>)	NODLL(NOCBA)	✓		✓	See 95
<u>DLL</u> (<u>CBA</u> <u>NOCBA</u>)	DLL(NOCBA)		✓	✓	See 95
<u>ENUMSIZE</u> (subopts)	ENUM(SMALL)	✓	✓	✓	See 97
<u>EVENTS</u> [(filename)] <u>NOEVENTS</u>	NOEVENT	✓	✓	✓	See 99
<u>EXECOPS</u> <u>NOEXECOPS</u>	EXEC	✓	✓	✓	See 100
<u>EXH</u> <u>NOEXH</u>	EXH		✓		See 101
<u>EXPMAC</u> <u>NOEXPMAC</u>	NOEXP	✓	✓	✓	See 102
<u>EXPORTALL</u> <u>NOEXPORTALL</u>	NOEXPO	✓	✓	✓	See 103
<u>FASTTEMPINC</u> <u>NOFASTTEMPINC</u>	NOFASTT		✓		See 103
<u>FLAG</u> (severity) <u>NOFLAG</u>	FL(I)	✓	✓	✓	See 104
<u>FLOAT</u> (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 105
<u>GOFF</u> <u>NOGOFF</u>	NOGOFF	✓	✓	✓	See 110
<u>GONUMBER</u> <u>NOGONUMBER</u>	NOGONUM	✓	✓	✓	See 111
<u>HALT</u> (num)	HALT(16)	✓	✓	✓	See 112
<u>HALTONMSG</u> (msgno) <u>NOHALTONMSG</u>	NOHALTON	✓	✓	✓	See 113
<u>IGNERRNO</u> <u>NOIGNERRNO</u>	NOIGNER	✓	✓	✓	See 114
<u>INFO</u> [(subopts)] <u>NOINFO</u>	For C++: IN(LAN) For C: NOIN	✓	✓		See 115
<u>INITAUTO</u> (number [,word]) <u>NOINITAUTO</u>	NOINITA	✓	✓	✓	See 116

Table 5. Compiler options, abbreviations, and IBM supplied defaults (continued)

Compiler Option (Abbreviated Names are underlined)	IBM Supplied Default	C	C++	IPA Link	More Information
<u>INLINE</u> (subopts) <u>NOINLINE</u> [(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 118
<u>INLRPT</u> [(filename)] <u>NOINLRPT</u> [(filename)]	NOINLR	✓	✓	✓	See 121
<u>IPA</u> (subopts) <u>NOIPA</u> (subopts)	NOIPA(NOLINK, OBJECT, OPT, NOLIST, NOGONUMBER, NOATTRIBUTE, NOXREF, LEVEL(1),NOMAP, DUP, ER, NONCAL, NOUPCASE, NOCONTROL, NOPDF1, NOPDF2, NOPDFNAME)	✓	✓	✓	See 122
<u>KEYWORD</u> (name) <u>NOKEYWORD</u> (name)	Recognizes all C++ keywords		✓	✓	See 130
<u>LANGLVL</u> (subopts)	LANG(EXTENDED)	✓	✓	✓	See 131
<u>LIBANSI</u> <u>NOLIBANSI</u>	NOLIB	✓	✓	✓	See 142
<u>LIST</u> [(filename)] <u>NOLIST</u> [(filename)]	NOLIS	✓	✓	✓	See 143
<u>LOCALE</u> [(name)] <u>NOLOCALE</u>	NOLOC	✓	✓	✓	See 145
<u>LONGNAME</u> <u>NOLONGNAME</u>	C:NOLO C++: LO	✓	✓	✓	See 147
<u>LP64</u> <u>ILP32</u>	ILP32	✓	✓	✓	See 148
<u>LSEARCH</u> (subopts) <u>NOLSEARCH</u>	NOLSE	✓	✓	✓	See 150
<u>MARGINS</u> <u>NOMARGINS</u>	NOMAR		✓		See 156
<u>MARGINS</u> (m,n) <u>NOMARGINS</u>	V-format: NOMAR F-format: MAR(1,72)	✓		✓	See 156
<u>MAXMEM</u> (size) <u>NOMAXMEM</u>	MAXM(2097152)	✓	✓	✓	See 157
<u>MEMORY</u> <u>NOMEMORY</u>	MEM	✓	✓	✓	See 158
<u>NAMEMANGLING</u> (subopt)	NAMEMANGLING(zOSV1R2)		✓	✓	See 159
<u>NESTINC</u> (num) <u>NONESTINC</u>	NEST(255)	✓	✓	✓	See 161
<u>OBJECT</u> [(filename)] <u>NOOBJECT</u> [(filename)]	OBJ	✓	✓	✓	See 162
<u>OBJECTMODEL</u> (subopt)	OBJECTMODEL(COMPAT)		✓	✓	See 163
<u>OE</u> [(filename)] <u>NOOE</u> [(filename)]	NOOE	✓	✓	✓	See 165
<u>OFFSET</u> <u>NOOFFSET</u>	NOOF	✓	✓	✓	See 166
<u>OPTFILE</u> [(filename)] <u>NOOPTFILE</u> [(filename)]	NOOPTF	✓	✓	✓	See 167
<u>OPTIMIZE</u> [(level)] <u>NOOPTIMIZE</u>	NOOPT	✓	✓	✓	See 169
<u>PHASEID</u> <u>NOPHASEID</u>	NOPHASEID	✓	✓	✓	See 172
<u>PLIST</u> (HOST OS)	PLIST(HOST)	✓	✓	✓	See 173
<u>PORT</u> (PPS <u>NOPPS</u>) <u>NOPORT</u> (PPS <u>NOPPS</u>)	NOPORT(NOPPS)		✓		See 174

Table 5. Compiler options, abbreviations, and IBM supplied defaults (continued)

Compiler Option (Abbreviated Names are underlined)	IBM Supplied Default	C	C++	IPA Link	More Information
<u>PPONLY</u> [(subopts)] <u>NOPPONLY</u> [(subopts)]	NOPP	✓	✓	✓	See 175
<u>REDIR</u> <u>NOREDIR</u>	RED	✓	✓	✓	See 178
<u>RENT</u> <u>NORENT</u>	NORENT	✓		✓	See 179
<u>ROCONST</u> <u>NOROCONST</u>	C: NOROC C++: ROC	✓	✓	✓	See 180
<u>ROSTRING</u> <u>NOROSTRING</u>	ROS	✓	✓	✓	See 181
<u>ROUND</u> (opt)	For IEE: ROUND(N) For HEX: ROUND(Z)	✓	✓	✓	See 182
<u>RTTIINORTTI</u>	NORTTI		✓	✓	See 183
<u>SEARCH</u> (opt1,opt2,...) <u>NOSEARCH</u>	For C++, SE(//CEE.SCEEH.+, //CBC.SCLBH.+') For C, SE(//CEE.SCEEH.+')	✓	✓	✓	See 184
<u>SEQUENCE</u> <u>NOSEQUENCE</u>	NOSEQ		✓		See 185
<u>SEQUENCE</u> (m,n) <u>NOSEQUENCE</u>	V-format: NOSEQ F-format: SEQ(73,80)	✓		✓	See 185
<u>SERVICE</u> (string) <u>NOSERVICE</u>	NOSERV	✓	✓	✓	See 186
<u>SHOWINC</u> <u>NOSHOWINC</u>	NOSHOW	✓	✓	✓	See 187
<u>SOURCE</u> [(filename)] <u>NOSOURCE</u> [(filename)]	NOSO	✓	✓	✓	See 188
<u>SPILL</u> (size) <u>NOSPILL</u> [(size)]	SP(128)	✓	✓	✓	See 189
<u>SQL</u> <u>NOSQL</u>	NOSQL	✓	✓	✓	See 191
<u>SSCOMM</u> <u>NOSSCOMM</u>	NOSS	✓		✓	See 192
<u>START</u> <u>NOSTART</u>	STA	✓	✓	✓	See 193
<u>STATICINLINE</u> <u>NOSTATICINLINE</u>	NOSTATICI		✓	✓	See 194
<u>STRICT</u> <u>NOSTRICT</u>	STRICT	✓	✓	✓	See 195
<u>STRICT_INDUCTION</u> <u>NOSTRICT_INDUCTION</u>	NOSTRICT_INDUC	✓	✓	✓	See 196
<u>SUPPRESS</u> (msg-no) <u>NOSUPPRESS</u>	NOSUPP	✓	✓	✓	See 197
<u>TARGET</u> (suboption)	TARG(LE, CURRENT)	✓	✓	✓	See 197
<u>TEMPINC</u> [(filename)] <u>NOTEMPINC</u> [(filename)]	PDS: TEMPINC(TEMPINC) HFS Directory: TEMPINC(/tempinc)		✓		See 203
<u>TEMPLATERECOMPILE</u> <u>NOTEMPLATERECOMPILE</u>	TEMPLATEREC		✓	✓	See 204
<u>TEMPLATEREGISTRY</u> <u>NOTEMPLATEREGISTRY</u>	NOTEMPL		✓	✓	See 205
<u>TERMINAL</u> <u>NOTERMINAL</u>	TERM	✓	✓	✓	See 206
<u>TEST</u> [(subopts)] <u>NOTEST</u> [(subopts)]	C: NOTEST (HOOK, SYM, BLOCK, LINE, PATH) C++: NOTEST(HOOK)	✓	✓	✓	See 206
<u>TMPLPARSE</u> (subopts)	TMPLPARSE(NO)		✓		See 211
<u>TUNE</u> (n)	TUN(5)	✓	✓	✓	See 212

Table 5. Compiler options, abbreviations, and IBM supplied defaults (continued)

Compiler Option (Abbreviated Names are underlined)	IBM Supplied Default	C	C++	IPA Link	More Information
<u>UNDEFINE</u> (name)	no default	✓	✓	✓	See 214
<u>UNROLL</u> (subopts)	UNROLL(AUTO)	✓	✓	✓	See “UNROLL” on page 215
<u>UPCONV</u> <u>NOUPCONV</u>	NOUPC	✓		✓	See 216
<u>WARN64</u> <u>NOWARN64</u>	NOWARN64	✓	✓	✓	See 216
<u>WSIZEOF</u> <u>NOWSIZEOF</u>	NOWSIZEOF		✓	✓	See 217
<u>XPLINK</u> [(subopts)] <u>NOXPLINK</u> [(subopts)]	NOXPL	✓	✓	✓	See 218
<u>XREF</u> <u>NOXREF</u>	NOXR	✓	✓	✓	See 222

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 6. Compiler options for file management

Option	Description	C Compile	C++ Compile	IPA Link	More Information
DBRMLIB	Enables the SQL option to be used in the UNIX System Services environment.	✓	✓	✓	See “DBRMLIB” on page 88
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 103
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 126
LSEARCH	Specifies the libraries or disks to be scanned for user include files.	✓	✓	✓	See 150
MEMORY	Improves compile-time performance by using a MEMORY file in place of a work file, if possible.	✓	✓	✓	See 158
OBJECT	Produces an object module, and stores it in the file that you specify, or in the data set associated with SYSLIN.	✓	✓	✓	See 162
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 165
OPTFILE	Directs the compiler to look for compiler options in the file specified.	✓	✓	✓	See 167

Table 6. Compiler options for file management (continued)

Option	Description	C Compile	C++ Compile	IPA Link	More Information
SEARCH	Specifies the libraries or disks to be scanned for system include files.	✓	✓	✓	See 184
TEMPINC	Places template instantiation files in the PDS or HFS directory specified.		✓		See 203
TEMPLATERECOMPILE	Helps to manage dependencies between compilation units that have been compiled using the TEMPLATEREGISTRY option.		✓	✓	See 204
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 205

Options that control the preprocessor

These options specify how the preprocessor runs.

Table 7. Summary of compiler options for preprocessor

Option	Description	C Compile	C++ Compile	IPA Link	More Information
CONVLIT	Turns on string literal codepage conversion.	✓	✓	✓	See 82
DEFINE	Defines preprocessor macro names.	✓	✓	✓	See 92
DIGRAPH	Allows you to use additional digraphs in both C and C++ applications.	✓	✓	✓	See 93
LOCALE	Specifies the locale to be used at compile time.	✓	✓	✓	See 145
PPONLY	Specifies that only the preprocessor is to be run and not the compiler.	✓	✓	✓	See 175
UNDEFINE	Removes any value its argument may have.	✓	✓	✓	See 214

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 8. 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 <i>n</i> or above.	✓	✓	✓	See 112
HALTONMSG	Instructs the C++ front-end to stop after the compilation phase when it encounters the specified <i>msg_number</i> .	✓	✓	✓	See 113
MARGINS	Identifies position of source to be scanned by the compiler.	✓	✓	✓	See 156
NESTINC	Specifies the number of nested include files to be allowed.	✓	✓	✓	See 161

Table 8. Summary of compiler options used for input source file processing control (continued)

Option	Description	C Compile	C++ Compile	IPA Link	More Information
SEQUENCE	Specifies the columns used for sequence numbers.	✓	✓	✓	See 185

Options that control the compiler listing

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

Table 9. 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 65
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 75
EXPMAC	Lists all expanded macros. You must use the SOURCE option with EXPMAC.	✓	✓	✓	See 102
INLINE(,REPORT,,)	Generates a report on the status of inlined functions.	✓	✓	✓	See 118
INLRPT	Generates a report on the status of inlined functions.	✓	✓	✓	See 121
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 122
LIST	Includes the object module in the compiler listing, in assembler-like code.	✓	✓	✓	See 143
OFFSET	Lists offset addresses relative to entry points of functions. The LIST option must be used with OFFSET.	✓	✓	✓	See 166
SHOWINC	Lists include files if SOURCE option is specified.	✓	✓	✓	See 187
SOURCE	Lists source file.	✓	✓	✓	See 188

Table 9. 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 222

Options that control the IPA object

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

Table 10. 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 123
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 123
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 123
IPA(OBJECT)	Indicates whether a conventional (non-IPA)/IPA object is to be produced during the IPA Compile step.	✓	✓	✓	See 123
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 123
IPA(XREF)	Saves information about symbol storage offsets in the IPA object file.	✓	✓	✓	See 123

Options that control the IPA Link step

These options control the IPA Link step.

Table 11. Compiler options for IPA Link control

Option	Description	C Compile	C++ Compile	IPA Link	More Information
IPA(LEVEL(0 1 2 PDF1 PDF2 PDFNAME))	Indicates the level of IPA optimization that the IPA Link step should perform after it links the object files into the call graph.	✓	✓	✓	See 126
IPA(LINK)	Instructs the compiler to perform IPA Link processing.	✓	✓	✓	See 126
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 126
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 126

Options for debugging and diagnosing errors

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

Table 12. 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 77
DEBUG	Instructs the compiler to generate debug information.	✓	✓		See “DEBUG NODEBUG” on page 88
EVENTS	Produces an events file that contains error information and source file statistics. The IPA Link step accepts but ignores this option.	✓	✓	✓	See 99
FLAG	Specifies the lowest severity level to be listed.	✓	✓	✓	See 104
GONUMBER	Generates line number tables for Debug Tool and error trace backs. The TEST option turns on GONUMBER.	✓	✓	✓	See 111
INFO	Generates informational messages.	✓	✓		See 115
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 122
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 122

Table 12. Compiler options for debugging and diagnostics (continued)

Option	Description	C Compile	C++ Compile	IPA Link	More Information
PHASEID	Causes each compiler module (phase) to issue an informational message which identifies the compiler phase module name, product identifier, and build level.	✓	✓	✓	See 172
SERVICE	Places a string in the object module, which is displayed in the traceback if the application fails abnormally.	✓	✓	✓	See 186
SUPPRESS	Prevents the batch compiler, or driver informational, or warning messages from being displayed or added to the listings.	✓	✓	✓	See 197
TERMINAL	Directs diagnostic messages to be displayed on the terminal.	✓	✓	✓	See 206
TEST	Generates information that Debug Tool needs to debug your program.	✓	✓	✓	See 206
WARN64	Generates diagnostic messages for situations where compiling the source code with ILP32 and LP64 may produce different behavior. This option is designed to help a program migrate to 64-bit mode.	✓	✓	✓	See 216
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 218
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 218

Options that control the programming language characteristics

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

Table 13. 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 bit fields.	✓	✓	✓	See 76
CHARS(UNSIGNED)	Instructs the compiler to treat all variables of type char as either signed or unsigned.	✓	✓	✓	See 76
KEYWORD	Controls whether the specified name is created as a keyword or an identifier whenever it appears in your C++ source.		✓	✓	See 130
LANGLVL	Specifies the language standard to be used.	✓	✓	✓	See 131

Table 13. Summary of compiler options used for programming language characteristics control (continued)

Option	Description	C Compile	C++ Compile	IPA Link	More Information
SSCOMM	Allows comments to be specified by two slashes (//). The IPA Link step accepts but ignores this option.	✓		✓	See 192
STATICINLINE	Treats an inline function as static instead of extern.		✓	✓	See 194
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 211
UPCONV	Preserves <i>unsignedness</i> 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 14. 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 64
ALIAS	Generates ALIAS binder control statements for each required entry point.	✓			See 66
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 67
ARCHITECTURE	Specifies the architecture for which the executable program instructions are to be generated.	✓	✓	✓	See 70
ASCII	Provides native ASCII/NLS support.	✓	✓	✓	See 74
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 79

Table 14. Summary of compiler options used for object code control (continued)

Option	Description	C Compile	C++ Compile	IPA Link	More Information
COMPRESS	Suppresses the generation of function names in the function control block, thereby reducing the size of your application's load module.	✓	✓	✓	See 81
CSECT	Instructs the compiler to generate csect names in the output object module.	✓	✓	✓	See 84
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 86
DLL	Generates object code for DLLs or DLL applications.	✓	✓	✓	See 95
ENUMSIZE	Specifies the amount of storage occupied by enumerations	✓	✓	✓	See 97
EXH	Controls the generation of C++ exception handling code.		✓		See 101
EXPORTALL	Exports all externally defined functions and variables.	✓	✓	✓	See 103
FLOAT	Switches floating-point representation between IEEE and hexadecimal.	✓	✓	✓	See 105
GOFF	Instructs the compiler to produce an object file in the Generalized Object File Format.	✓	✓	✓	See 110
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 114
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 116
INLINE	Inlines user functions into source and helps maximize optimization.	✓	✓	✓	See 118
IPA	Instructs the compiler to perform Interprocedural Analysis (IPA) processing.	✓	✓	✓	See 122
IPA(LEVEL)	Indicates the level of IPA optimization that the IPA Link step should perform.	✓	✓	✓	See 122
LIBANSI	Indicates whether or not functions with the name of an ANSI C library function are in fact ANSI C library functions.	✓	✓	✓	See 142
LONGNAME	Provides support for external names of mixed case and up to 1024 characters long.	✓	✓	✓	See 147
LP64	Instructs the compiler to generate code that runs on a 64-bit architecture machine.	✓	✓	✓	See 148
MAXMEM	Limits the amount of memory used for local tables of specific, memory intensive optimization.	✓	✓	✓	See 157

Table 14. Summary of compiler options used for object code control (continued)

Option	Description	C Compile	C++ Compile	IPA Link	More Information
NAMEMANGLING	Enables the encoding of variable names into unique names so that linkers can separate common names in the language.		✓	✓	See 159
OBJECT	Produces an object module, and stores it in the file that you specify, or in the data set associated with SYSLIN.	✓	✓	✓	See 162
OBJECTMODEL	Sets the type of object model.	✓	✓	✓	See 163
OPTIMIZE	Improves run-time performance by introducing optimizations during code generation.	✓	✓	✓	See 169
RENT	Generates reentrant code. The IPA Link step accepts but ignores this option.	✓		✓	See 179
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 180
ROSTRING	Informs the compiler that string literals are read-only.	✓	✓	✓	See 181
ROUND	Sets the rounding mode for binary floating point numbers.	✓	✓	✓	See 182
SPILL	Specifies the size of the spill area to be used for compilation.	✓	✓	✓	See 189
SQL	Enables the compiler to process embedded SQL statements.	✓	✓	✓	See 191
START	Generates a CEESTART whenever necessary.	✓	✓	✓	See 193
STRICT	Affects the precision of floating point calculations.	✓	✓	✓	See 195
STRICT_INDUCTION	Instructs the compiler to disable loop induction variable optimizations.	✓	✓	✓	See 196
TARGET	Generates an object module for the targeted operating system or run-time library.	✓	✓	✓	See 197
TUNE	Specifies the architecture for which the executable program will be optimized.	✓	✓	✓	See 212
UNROLL	Determines whether or not unrolling is allowed on any loops in a specified program.	✓	✓	✓	See "UNROLL" on page 215
WSIZEOF	Causes the size of operator to return the widened size for function return types.	✓	✓	✓	See 217
XPLINK	Instructs the compiler to generate extra performance linkage for function calls. XPLINK(CALLBACK) specifies that all calls via function pointers will be considered potentially incompatible, and fix-up code will be inserted by the compiler to assist the call.	✓	✓	✓	See 218

Options that control program execution

These options control the execution of your program

Table 15. 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 73
ASCII	Provides ASCII/NLS support.	✓	✓	✓	See 74
EXECOPS	Allows you to specify run-time options on the invocation line.	✓	✓	✓	See 100
PLIST	Specifies that the original operating system parameter list should be available.	✓	✓	✓	See 173
REDIR	Allows redirection of stderr, stdin, and stdout from the invocation line.	✓	✓	✓	See 178
RTTI	Generates run-time type identification (RTTI) information for the typeid operator and the dynamic_cast operator.		✓	✓	See 183
TARGET	Generates an object module for the targeted operating system or run-time library.	✓	✓	✓	See 197

Portability options

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

Table 16. 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 174

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.

Each compiler option description that follows includes an Option Scope table and an Option Default table. The following Option Scope table is an example that shows that, in this case, the compiler option is supported by both the C compiler and the C++ compiler. The table also shows that, in this case, the compiler option is accepted by the IPA Link step and also shows that it undergoes special IPA processing during the IPA Compile and IPA Link steps.

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

The Option Default table shows the possible defaults for the compiler option in various environments.

Option Default						
z/OS C/C++ Compiler (Batch and TSO Environments)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
A	B	C	D	E	F	G

The following list describes the possible table entries, which have been identified above with letters A through G:

- A** The default compiler option for the batch and TSO environment
- B, C, D** The compiler option settings for the corresponding z/OS UNIX System Services compiler utility (c89, cc, or c++)
- E, F, G** The compiler option settings for the corresponding z/OS UNIX System Services compiler utility (c89, cc, or c++) during the IPA Link step (if applicable)

Note: If the default compiler option in A is same as the option setting you get when you invoke the compiler under UNIX System Services, then it is not repeated in columns B, C, D, E, F, or G.

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

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.

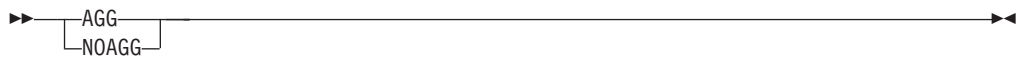
The value of the AGGRCOPY option for a partition is set to the value of the first subprogram that is placed in the partition. During IPA inlining, subprograms with different AGGRCOPY settings may be combined in the same partition. When this occurs, the resulting partition is always set to AGGRCOPY(OVERLAP).

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)	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 options 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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
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 options 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

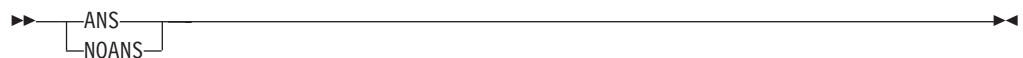
If you specify the ALIAS option on the IPA Link step, the IPA Link step generates an unrecoverable error.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
ANSIALIAS		NOANSIALIAS				

CATEGORY: Object Code Control



The ANSIALIAS option indicates to the compiler that the code strictly follows the type-based aliasing rules 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 *z/OS 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.

If you decide to use `ANSIALIAS` (the default), then you are making a promise to the compiler that your source code obeys the constraints in the ISO standard. On the basis of using this compiler option, the compiler front end passes aliasing information to the optimizer 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 promise and performs optimization accordingly. When it compares the instruction that stores into `x` and the instruction that loads out of `*(int *)`, it believes it is safe to put them in either order. Doing the load before the store will make the program run faster, so it interchanges them. The program becomes equivalent to:

```
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);
11 }
```

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

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 131 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 77 to see how to obtain more diagnostic information.

You can specify this option using the #pragma options 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
✓	✓	✓	✓	✓

z/OS C/C++ Compiler (Batch and TSO Environments)	Option Default					
	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
TARGET(zOSV1R6 and above): ARCH(5) TARGET(zOSV1R2 to zOSV1R5): ARCH(2)						

CATEGORY: Object Code Control

►► ARCH—(—n—)◄◄

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 212.

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 Produces code that is executable on all models.
- 1 Produces code that uses instructions available on the following system machine models:
 - 9021-520, 9021-640, 9021-660, 9021-740, 9021-820, 9021-860, and 9021-900

- 9021-xx1 and 9021-xx2
- 9672-Rx1, 9672-Rx2 (G1), 9672-Exx, and 9672-Pxx

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

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

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

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

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

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

Note that ARCH(3) is required for execution of a program that specifies the `FLOAT(IEEE)` compiler option. However, if the program is executed on a physical processor that does not actually provide these ARCH(3) facilities, any program check (operation or specification exception), resulting from an attempt to use features associated with IEEE floating point or the additional floating point registers, will be intercepted by the underlying OS/390 V2R6 or higher operating system, and simulated by software. There will be a significant performance degradation for the simulation.

4 Produces code that uses instructions available on model 2064-100 (z/900) in ESA/390 mode.

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

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

5 Is the default value. Produces code that uses instructions available on model 2064-100 (z/900) in z/Architecture mode.

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

6 Produces code that uses instructions available on the 2084-xxx models in z/Architecture mode.

Specifically, the compiler on these ARCH(6) machines and their follow-ons may exploit the long-displacement instruction set. The long-displacement facility provides a 20-bit signed displacement field in 69 previously existing instructions (by using a previously unused byte in the instructions) and 44

new instructions. A 20-bit signed displacement allows relative addressing of up to 524,287 bytes beyond the location designated by a base register or base and index register pair and up to 524,288 bytes before that location. The enhanced previously existing instructions generally are ones that handle 64-bit binary integers. The new instructions generally are new versions of instructions for 32-bit binary integers. The new instructions also include:

- A LOAD BYTE instruction that sign-extends a byte from storage to form a 32-bit or 64-bit result in a general register
- New floating-point LOAD and STORE instructions

The long-displacement facility provides register-constraint relief by reducing the need for base registers, code size reduction by allowing fewer instructions to be used, and additional improved performance through removal of possible address-generation interlocks.

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 options 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 212.

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.

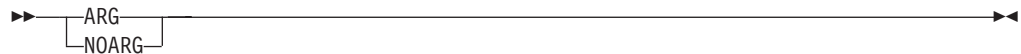
The Compiler Options Map section of the IPA Link step listing displays the option value that you specified for each IPA object file during the IPA Compile step.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
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.

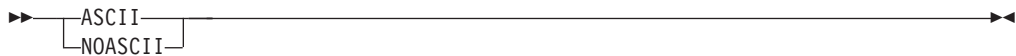
If you specified ARGPARSE on the IPA Compile step, you do not need to specify it again on the IPA Link step to affect that step. The IPA Link step uses the information generated for the compilation unit that contains the `main()` function. If it cannot find a compilation unit that contains `main()`, 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)	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 run-time functions will not be exposed under NOXPLINK. Specifying the NOXPLINK option explicitly will prevent you from using the ASCII run-time functions. ASCII 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.

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

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

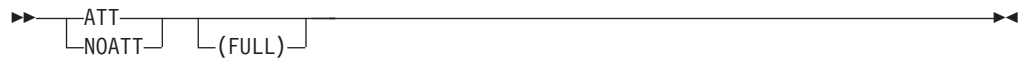
The IPA Link step accepts the ASCII option, but ignores it.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
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 options(XREF)`
- For C++, if you specify the **ATTR**, **XREF**, **IPA(ATTRIBUTE)**, or **IPA(XREF)** options

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

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.

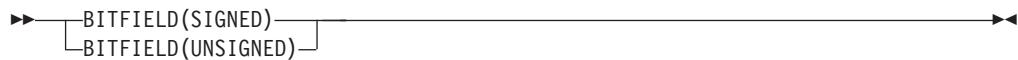
The IPA Link step creates a Storage Offset listing section if during the IPA Compile step you requested the additional symbol storage offset information for your IPA objects.

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)	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 instructs the compiler to specify all bit fields by default as either signed or unsigned.

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

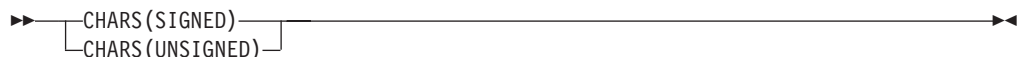
The IPA Link step accepts the BITFIELD option, but ignores it.

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)	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 *z/OS 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

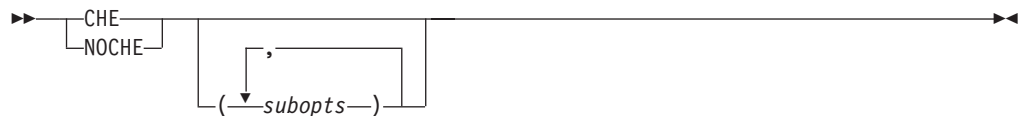
The IPA Link step accepts the CHARS option, but ignores it.

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)	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						

CATEGORY: Debug/Diagnostic



where:

subopts is one of the suboptions that are shown in Table 17 on page 78.

Note: As of z/OS V1R6, the INFO option is supported for both C and C++ so you may prefer to use INFO instead of CHECKOUT.

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 17. CHECKOUT suboptions, abbreviations, and descriptions

CHECKOUT Suboption	Abbreviated Name	Description
<u>ACCURACY</u> NO <u>ACCURACY</u>	AC NOAC	Assignments of long values to variables that are not long
<u>CAST</u> 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 67 for more information on ANSI type-based aliasing. Also see “DLL NODLL” on page 95 for DLL function pointer casting restrictions.
ENUM <u>NOENUM</u>	EN NOEN	Usage of enumerations
EXTERN <u>NOEXTERN</u>	EX NOEX	Unused variables that have external declarations
<u>GENERAL</u> NOGENERAL	GE NOGE	General checkout messages
<u>GOTO</u> NOGOTO	GO NOGO	Appearance and usage of goto statements
<u>INIT</u> NOINIT	I NOI	Variables that are not explicitly initialized
<u>PARAM</u> NOPARAM	PAR NOPAR	Function parameters that are not used
PORT <u>NOPORT</u>	POR NOPOR	Non-portable usage of the z/OS C language
<u>PPCHECK</u> NOPPCHECK	PPC NOPPC	All preprocessor directives
<u>PPTRACE</u> NOPTRACE	PPT NOPPT	Tracing of include files by the preprocessor
<u>TRUNC</u> 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 in 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 *z/OS C/C++ Language Reference* for more information regarding this pragma directive.

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

Note: See the “INFO | NOINFO” on page 115 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

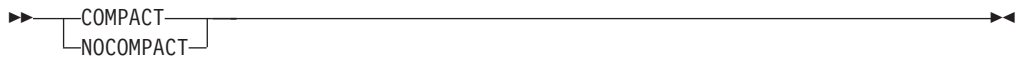
The IPA Link step accepts the CHECKOUT option, but ignores it.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOCOMPACT						

CATEGORY: Object Code Control



During optimizations performed as part of code generation, for both NOIPA and IPA, choices must be made between those optimizations which tend to result in faster but larger code and those which tend to result in smaller but slower code. The COMPACT option influences these choices. 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. Not all subprograms are inlined when COMPACT is specified.

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

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

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

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

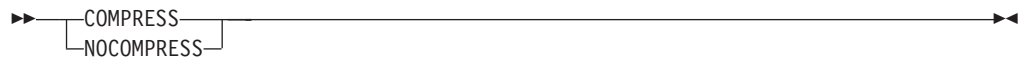
The Partition Map sections of the IPA Link step listing and the object module END information section display the value of the COMPACT option. The Partition Map also displays any subprogram-specific COMPACT values.

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)	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 Debug Tool and the 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.

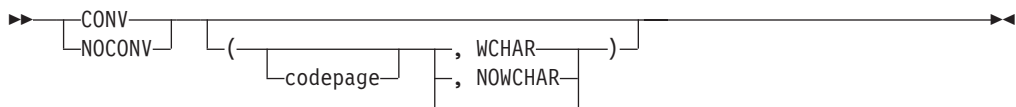
The Partition Map sections of the IPA Link step listing and the object module END information section display the value of the COMPRESS option.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
	NOCONVLIT (, NOWCHAR)					

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 and character constants within a compilation unit are converted to the specified codepage unless you use `#pragma convlit(suspend)` and `#pragma convlit(resume)` to exclude sections of code from conversion. See *z/OS 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 `NOWCHAR`. Only wide character constants and string literals made up of single byte character set (SBCS) characters are converted. If there are any shift-out (SO) and shift-in (SI) characters in the literal, the compilation will end with an error message.

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

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

Notes:

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

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

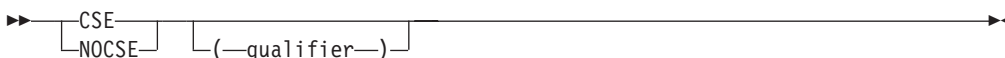
The IPA Link step accepts the `CONVLIT` option, but ignores it.

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)	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 *z/OS 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 right-most character is dropped and the compiler issues an informational message except in the case of GOFF. The test CSECT name is always truncated to 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.
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)	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.

Example: 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.

DBRMLIB

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
DBRMLIB (DD:DBRMLIB)						

CATEGORY: File Management

▶▶—DBRM—(—// partitioned data set (member)—)————▶▶

The DBRMLIB option specifies the data set for the database request module (DBRM), which is generated by the SQL option. The DBRM data set contains the embedded SQL statements and host variable information extracted from the source program, information that identifies the program, and ties the DBRM to the translated source statements. It becomes the input to the DB2 bind process. This option is only effective when the SQL option is specified. The partitioned data set must be either a relative data set name, or an absolute data set name enclosed in single quotes. In either case, it must also be prepended by //. When the option is specified in JCL, and a DBRMLIB DD statement is also specified, the option will take precedence over the DD statement. The compiler does not verify the DCB attributes of the data set; you must ensure the data set is created with the correct attributes, as expected by DB2 UDB. Refer to *DB2 Application Programming and SQL Guide* for details.

Effect on IPA Compile step

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

Effect on IPA Link step

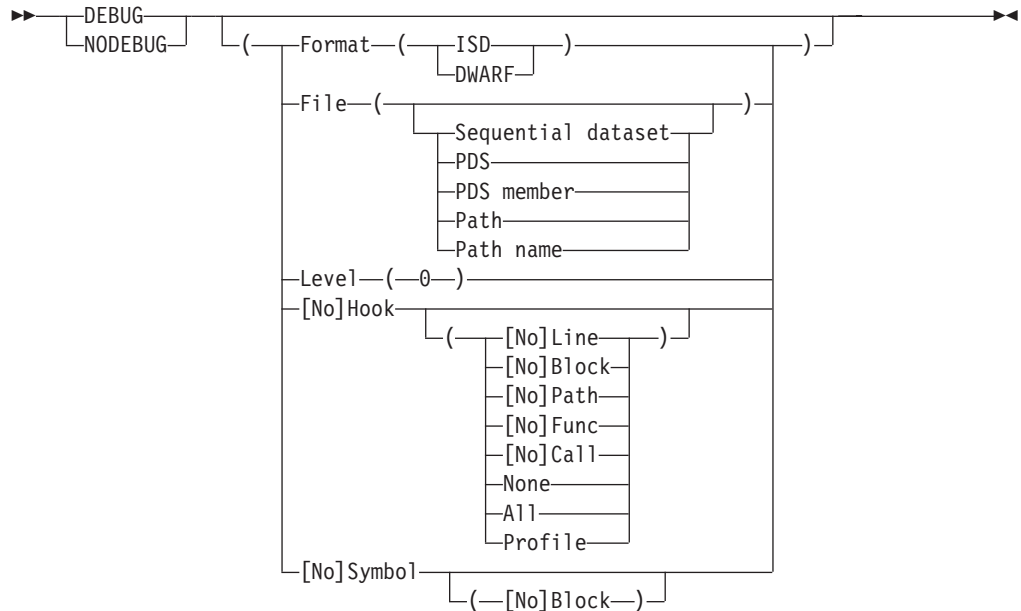
The IPA Link step accepts but ignores the DBRMLIB option.

DEBUG | NODEBUG

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NODEBUG						

CATEGORY: Debugging



The DEBUG option instructs the compiler to generate debug information based on the DWARF Version 3 debugging information format, which has been developed by the UNIX International Programming Languages Special Interest Group (SIG), and is an industry standard format.

Note: The Performance Analyzer only works with DEBUG(FORMAT(ISD)) in both z/OS V1R5 and V1R6. This also means that it will not work on 64-bit executables.

The TEST and GONUMBER options remain unchanged but will only work with ILP32. If you specify both TEST and DEBUG, the last valid specification is used. If you specify DEBUG and NODEBUG multiple times, the compiler uses the last specified option with the last specified suboption. For example, the following specifications have the same result:

```
cc -Wc,"NODEBUG(FORMAT(DWARF),HOOK(ALL))" -Wc,"DEBUG(NOSYMBOL)" hello.c
cc -WC,"DEBUG(FORMAT(DWARF),HOOK(ALL),NOSYMBOL)" hello.c
```

If you specify OPTIMIZE and DEBUG(FORMAT(DWARF)), no symbolic debug information is generated, but function entry, function exit, function call and function return hooks are generated. If you specify OPTIMIZE and DEBUG(FORMAT(ISD)), the behavior is the same as OPTIMIZE and TEST.

The following DEBUG suboptions control the debug format that is generated, as well as the level of debug information produced:

FORMAT

DEFAULT: FORMAT(DWARF)

Has the following suboptions: ISD and DWARF. ISD produces the same debug information as the TEST option. This suboption is available only with ILP32. If this format is used, the FILE suboption is ignored.

The DWARF suboption produces debug information in the DWARF Version 3 debugging information format, stored in the file specified by the FILE suboption. This is the only FORMAT supported with LP64. The compiler always generates a line number table in DWARF format when this suboption is specified; however, this DWARF line number table is incompatible with the Language Environment dump service routines (for example, ctrace()).

FILE

Specifies the name of the output file for FORMAT(DWARF). It can be a sequential data set, a PDS member, or an HFS file. If you do not specify a file name, the compiler uses the SYSCDBG DD statement, or its alternative, if you allocated it. Otherwise, the compiler constructs a file name as follows:

- If you are compiling a data set, the compiler uses the source file name to form the name of the output data set. The high-level qualifier is replaced with the userid under which the compiler is running, and .DBG is appended as the low-level qualifier.
- If you are compiling an HFS file, the compiler stores the debug information in a file that has the name of the source file with an .dbg extension.

For example, if TSY0U19 is compiling TSPERF.EON.SOURCE(EON) with the DEBUG option and does not specify a file name, the default output file name will be TSY0U19.EON.SOURCE.DBG(EON).

For a PDS or HFS directory compile, the FILE option specifies the PDS or HFS directory where the output files are generated.

The default for c89 is FILE(./filename.dbg).

The compiler resolves the full pathname for this file name, and places it in the generated object file. This information can be used by program analysis tools to locate the output file for FORMAT(DWARF). The user can examine this generated file name in the compiler listing file (see “LIST | NOLIST” on page 143 for instructions on how to create a compiler listing file), as shown in the following example:

```
PPA4: Compile Unit Debug Block
000140 0000001A          =F'26'          DWARF File Name
000144 ****          C'/hfs/fullpath/filename.dbg'
```

If the compiler cannot resolve the full pathname for the file name (for example, because the search permission was denied for a component of the file name), the compiler will issue a warning message, and the relative file name will be used instead.

LEVEL

DEFAULT: LEVEL(0)

Controls the amount of debug information produced. LEVEL(0) is the only level currently supported.

HOOK

DEFAULT: HOOK(ALL) for NOOPTIMIZE, HOOK(NONE,PROFILE) for OPTIMIZE

Controls the generation of LINE, BLOCK, PATH, CALL, and FUNC hook instructions. Hook instructions appear in the compiler Pseudo Assembly listing in the following form:

EX r0,HOOK.[type of hook]

The type of hook that each hook suboption controls is summarized in the list below:

- LINE
 - STMT - General statement
- BLOCK
 - BLOCK-ENTRY - Beginning of block
 - BLOCK-EXIT - End of block
 - MULTIEXIT - End of block and procedure
- PATH
 - LABEL - A label
 - DOBGN - Start of a loop
 - TRUEIF - True block for an if statement
 - FALSEIF - False block for an if statement
 - WHENBGN - Case block
 - OTHERW - Default case block
 - GOTO - Goto statement
 - POSTCOMPOUND - End of a PATH block
- CALL
 - CALLBGN - Start of a call sequence
 - CALLRET - End of a call sequence
- FUNC
 - PGM-ENTRY - Start of a function
 - PGM-EXIT - End of a function

There is also a set of shortcuts for specifying a group of hooks:

NONE It is the same as specifying NOLINE, NOBLOCK, NOPATH, NOCALL, and NOFUNC. It instructs the compiler to suppress all hook instructions.

ALL It is the same as specifying LINE, BLOCK, PATH, CALL, and FUNC. It instructs the compiler to generate all hook instructions. This is the ideal setting for debugging purposes.

PROFILE

It is the same as specifying CALL and FUNC. It is the ideal setting for tracing the program with the Performance Analyzer.

SYMBOL

DEFAULT: SYMBOL(BLOCK) for NOOPT, NOSYMBOL for OPT

Generates symbol information that gives you access to variable and other symbol information. The BLOCK suboption has no effect on the generated debug information, and is reserved for future use only.

If you specify the INLINE and DEBUG compiler options when NOOPTIMIZE is in effect, INLINE is ignored.

You can specify the `DEBUG` option and `TARGET` to a release prior to z/OS V1R5. However, if the debug format is DWARF, you must debug using `dbx` on a z/OS V1R5 (and above) system.

Effect on IPA Compile step

The `DEBUG` option is not supported by the IPA Compile step and is ignored.

Effect on IPA Link step

The `DEBUG` option is not supported by the IPA Link step and is ignored.

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)	z/OS UNIX System Services Utilities					
	Note you can override with appropriate <code>-D</code> or <code>-U</code> 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 ()))			

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 46 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, you can unset variables specified by -D, or automatically specified by c89, using -U when using the c89, cc, or c++ commands.

Note: c89 preprocesses -D and -U flags before passing them onto the compiler. xlc just passes -D and -U to the compiler, which interprets them as DEFINE and UNDEFINE. For more information, see Chapter 18, “c89 — Compiler invocation using host environment variables,” on page 471 or Chapter 19, “xlc — Compiler invocation using a customizable configuration file,” on page 513.

Effect on IPA Compile step

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

Effect on IPA Link step

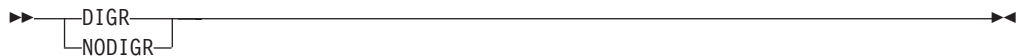
The IPA Link step accepts but ignores the DEFINE option.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
DIGRAPH						

CATEGORY: Preprocessor



The DIGRAPH option allows you to use additional digraphs when building 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 18 shows the digraphs that z/OS C/C++ supports:

Table 18. Digraphs

Key Combination	Character Produced
<%	{

Table 18. Digraphs (continued)

Key Combination	Character Produced
%>	}
<:	[
:>]
%:	#
%% ²	#
%:%:	##
%% ² %% ²	##

Table 19 shows additional keywords that z/OS C++ supports:

Table 19. 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 *z/OS 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.

2. The digraphs %% and %%² 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 %%² 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)	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 NOXPLINK NODLL code in another DLL, or non-XPLINK DLL code in another DLL, are passed to exported XPLINK functions, the compiler inserts code to check whether or not the function pointers received as actual arguments are valid (useable directly) XPLINK function pointers, and converts them if required. This provides results that are similar in many respects to the function pointer conversion provided when DLL(CALLBACKANY) is specified for non-XPLINK code. Other function pointers that have their origins in non-XPLINK code, including function pointer parameters passed to non-exported functions or otherwise acquired, are not converted automatically by XPLINK compiled code. Use of such function pointers will cause the application to fail. The abbreviation for CALLBACKANY is CBA.

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 *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 77 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)	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, 4, or 8 bytes of storage, depending on the range of the enum
 | constants.

INT

Specifies that enumerations occupy 4 bytes of storage and are represented by int.

INTLONG

Valid only when LP64 is specified and for C++ only. It specifies that enumerations occupy 8 bytes of storage and are represented by long if the range of the enum constants exceed the limit for int. Otherwise, the enumerations occupy 4 bytes of storage and are represented by int.

1

Specifies that enumerations occupy 1 byte of storage.

2

Specifies that enumerations occupy 2 bytes of storage

4

Specifies that enumerations occupy 4 bytes of storage.

8

Specifies that enumerations occupy 8 bytes of storage. This suboption is only valid with LP64.

The default is ENUM(SMALL). It allocates the amount of storage that is required by the smallest predefined 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 20. ENUM constants for C and C++

ENUM Constants	small	1	2	4	8 *	int	intlong * (C++ only)
0..127	unsigned char	signed char	short	int	long	int	int
-128..127	signed char	signed char	short	int	long	int	int
0..255	unsigned char	unsigned char	short	int	long	int	int
0..32767	unsigned short	ERROR	short	int	long	int	int
-32768..32767	short	ERROR	short	int	long	int	int
0..65535	unsigned short	ERROR	unsigned short	int	long	int	int
0..2147483647	unsigned int	ERROR	ERROR	int	long	int	int

Table 20. ENUM constants for C and C++ (continued)

ENUM Constants	small	1	2	4	8 *	int	intlong * (C++ only)
$-2^{31}..2^{31}-1$	int	ERROR	ERROR	int	long	int	int
0..4294967295	unsigned int	ERROR	ERROR	unsigned int	long	unsigned int (C++ only)	unsigned int
$0..(2^{63}-1) *$	unsigned long	ERROR	ERROR	ERROR	long	ERROR	long
$-2^{63}..(2^{63}-1) *$	long	ERROR	ERROR	ERROR	long	ERROR	long
$0..2^{64} *$	unsigned long	ERROR	ERROR	ERROR	unsigned long	ERROR	unsigned long

Note: The rows and columns marked with asterisks above (*) are only valid when the LP64 option is in effect.

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

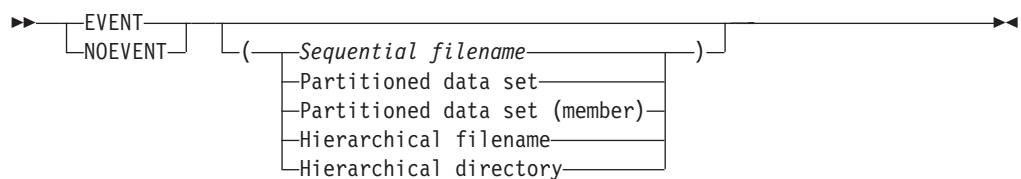
The IPA Link step accepts the ENUMSIZE option, but ignores it.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
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 F, “Layout of the Events file,” on page 617.

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

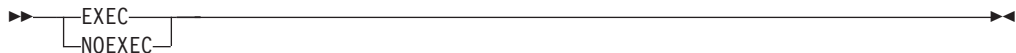
The IPA Link step accepts the EVENTS option, but ignores it.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
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)	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 NOEXH has been specified and the source file has try/catch blocks or throws objects, the program may not execute as expected.

The program makes a higher demand on heap memory if exception handling is used. The COMPRESS suboption allows the compiler to trade off heap memory

requirement with execution speed and load module size, if possible. This is only a suggestion to the compiler. Whether space saving can be achieved or not, depends on the actual code. The default is NOCOMPRESS.

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

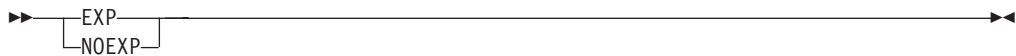
The IPA Link step issues a diagnostic message if you specify the EXH option for that step.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
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

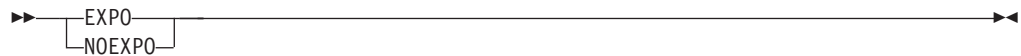
The IPA Link Step accepts the EXPMAC option, but ignores it.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
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 external functions and variables defined in the DLL. You may not export the `main()` function.

Notes:

1. If you only want to export some of the external functions and variables in the DLL, use `#pragma export`, or the `_Export` keyword for C++. For more information on `#pragma export`, see *z/OS 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.
3. Unused extern inline functions will not be exported when the EXPORTALL option is specified.

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.

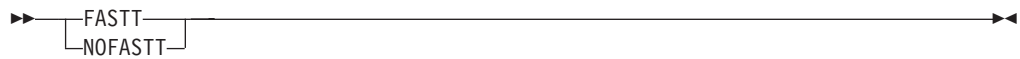
If you use the EXPORTALL option during the IPA Compile step, you severely limit IPA optimization. Refer to “DLL | NODLL” on page 95 for more information about the effects of this option on IPA processing.

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)	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)	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.
- 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

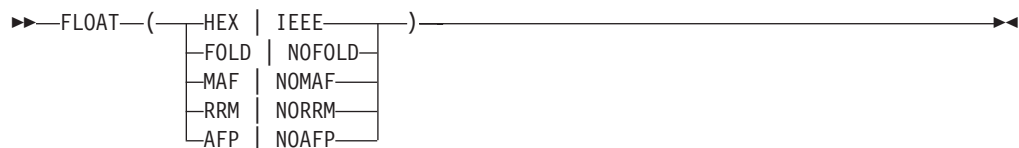
The IPA Link step uses the FLAG value that you specify for that step.

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)	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. For LP64, the default is IEEE.						

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.

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 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 zSeries 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 10 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 *z/OS 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)	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 NOCSECT option.

When you specify the GOFF option, you must use the binder to bind the output object. You cannot use the prelinker to process GOFF objects.

Note: When using GOFF and source files with duplicate file names, the linker may emit an error and discard one of the code sections. In this case, turn off the CSECT option by specifying NOCSECT.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOGONUMBER						

CATEGORY: Debug/Diagnostic



The GONUMBER option generates line number tables that correspond to the input source file. These tables are for use by Debug Tool and for error trace back information when an exception occurs. This option is available only with ILP32. If you specify LP64 and GONUMBER, the compiler issues a warning message and ignores the GONUMBER option.

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 options 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 45 and “LIST | NOLIST” on page 143 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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
HALT(16)						

CATEGORY: Input Source File Processing Control

▶▶—HALT—(num)—▶▶

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOHALTON						

CATEGORY: Input Source File Processing Control

▶▶—

HALTON
NOHALTON

—(n)—▶▶

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: The HALTONMSG option for C allows you to specify more than one message number by separating the message numbers with commas. The HALTONMSG option for C++ can only accept one message number.

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

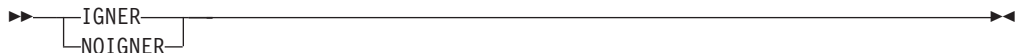
The IPA Link step accepts the HALTONMSG option but ignores it.

IGNERRNO | NOIGNERRNO

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

z/OS C/C++ Compiler (Batch and TSO Environments)	Option Default					
	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
For NOOPT and OPT(2), the default is NOIGNERRNO.						
For OPT(3), the default is IGNERRNO.						

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 options` 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.

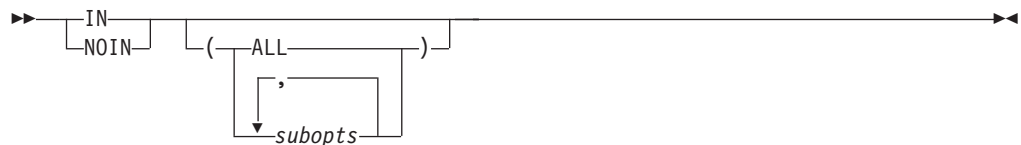
The value of the IGNERRNO option for a partition is set to the value of the first subprogram that is placed in the partition. The Partition Map sections of the IPA Link step listing and the object module display the value of the IGNERRNO option.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
For C++: INFO(LAN)	INFO(NOALL)	INFO(NOALL)	INFO(LAN)			
For C: NOINFO						

CATEGORY: Debug/Diagnostic



Note: The INFO option may not produce the same diagnostic messages as the previous releases.

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 (C++ only).
- CMP Emits conditional expression check messages.
- CND Emits messages on redundancies or problems in conditional expressions.
- CNV Emits messages about conversions.
- CNS Emits redundant operation on constants messages.
- CPY Emits warnings about copy constructors (C++ only).
- EFF Emits information about statements with no effect.
- ENU Emits information about ENUM checks.

- I EXT Emits warnings about unused variables that have external declarations (C
- I only).
- I GNR Emits information about the generation of temporary variables (C++ only).
- GEN Emits message if compiler generates temporaries.
- LAN Emits language level checks.
- PAR Emits warning messages on unused parameters.
- POR Emits warnings about non-portable constructs.
- PPC Emits messages on possible problems with using the preprocessor.
- PPT Emits trace of preprocessor actions.
- REA Emits warnings about unreached statements.
- RET Emits warnings about return statement consistency.
- TRD Emits warnings about possible truncation of data.
- I UND Emits warnings about undefined classes (C++ only).
- USE Emits information about usage of variables.
- I VFT Indicates where vftable is generated (C++ only).
- ALL Emits all of the above

no suboptions

Same result as INFO(ALL).

I In the z/OS UNIX System Services environment, this option is turned on by

I specifying -V. If you use the c++ command, the suboption is ALL. If you use the c89

I or cc commands, the suboptions are ALL, NOEXT, NOPPC, and NOPPT.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOINITAUTO						

CATEGORY: Object Code Control

The diagram shows the syntax for the INITAUTO option. It starts with a double arrow pointing right, followed by a bracketed choice between 'INITA' and 'NOINITA'. This is followed by a space, then a left parenthesis, then a bracketed choice between a hexadecimal value 'nnnnnnnn' and a comma followed by 'WORD'. This is followed by a right parenthesis, a space, and another double arrow pointing right.

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 *z/OS 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.

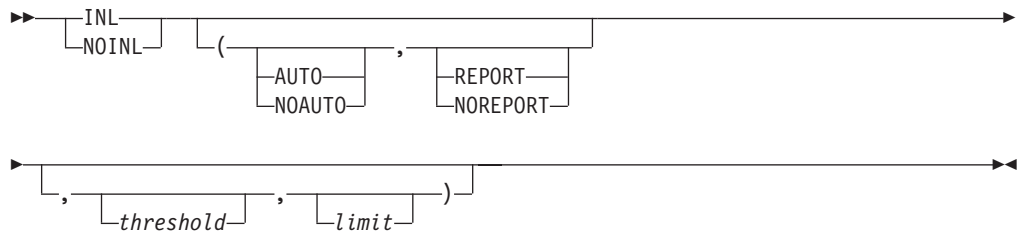
The Partition Map sections of the IPA Link step listing and the object module display the value of the INITAUTO option.

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)	z/OS UNIX System Services Utilities					
	Note that these values change if the -0 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 noline` directives allow you to determine which subprograms are to be inlined and which are not when the `INLINE` option is specified. These `#pragma` directives have no effect if you specify `NOINLINE`. See *z/OS 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 270, “Inline Report” on page 256, and “Inline Report for IPA Inliner” on page 281.

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 options directive for C.

In the z/OS UNIX System Services environment, specifying -V, when using the c89 or cc commands, will turn on the REPORT suboption of INLINE. The INLINE option itself is not touched (or changed) by -V.

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.

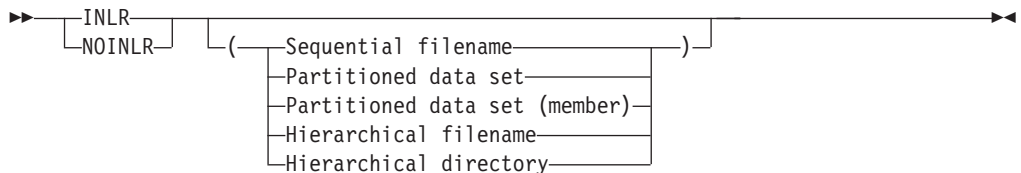
The IPA inliner has the inlining capabilities of the compilation unit inliner. In addition, the IPA inliner detects complex recursion, and may inline it. If you specify the `INLRPT` option, the IPA Link listing contains the IPA Inline Report section. This section is similar to the report that the compilation unit inliner generates. If you specify `NOINLRPT`, or `NOINLRPT`, 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)	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

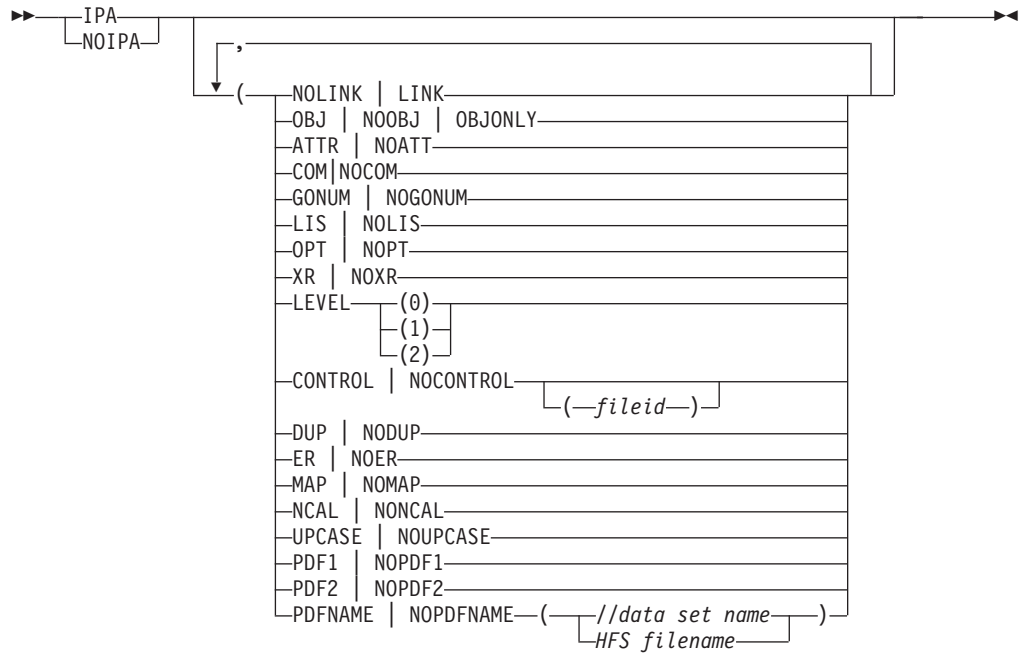
If you specify the INLRPT option on the IPA Link step, the IPA Link step listing contains an IPA Inline Report section. Refer to “INLINE | NOINLINE” on page 118 for more information about generating an IPA Inline Report section.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOIPA	NOIPA	NOIPA	NOIPA	NOIPA(NOCONTROL (ipa.ct1), DUP,NOER, NOMAP, NOUPCASE, NONCAL) IPA(LINK, LEVEL(1))	NOIPA(NOCONTROL (ipa.ct1), DUP,NOER, NOMAP, NOUPCASE, NONCAL) IPA(LINK, LEVEL(1))	NOIPA(NOCONTROL (ipa.ct1), 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.

GONUMBER NOGONUMBER	<p>The default is IPA(COMPRESS). The abbreviations are IPA(COM NOCOM).</p> <p>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.</p> <p>Refer to “GONUMBER NOGONUMBER” on page 111 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 45.</p> <p>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.</p>
LIST NOLIST	<p>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.</p> <p>Refer to “LIST NOLIST” on page 143 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 45.</p> <p>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.</p>
OBJECT NOOBJECT OBJONLY	<p>Controls the content of the object file.</p> <ul style="list-style-type: none"> • OBJECT <p>The options IPA(NOLINK,OBJECT) result in an IPA Compile step.</p> <p>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 <i>z/OS C/C++ Programming Guide</i> for a list of optimizations.</p> <p>The object file may be used by an IPA Link step, a prelink/link, or a bind.</p> • NOOBJECT <p>The options IPA(NOLINK,NOOBJECT) result in an IPA compile step.</p> <p>The compiler performs IPA compile-time optimizations and generates IPA object</p>

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 222 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).

DUP | NODUP

Indicates whether the IPA Link step writes a message and a list of duplicate symbols to the console.

The default is IPA(DUP).

ER | NOER

Indicates whether the IPA Link step writes a message and a list of unresolved symbols to the console.

The default is IPA(NOER).

LEVEL(0|1|2)

Indicates the level of IPA optimization that the IPA Link step should perform after it links the object files into the call graph.

If you specify LEVEL(0), IPA performs subprogram pruning and program partitioning only.

If you specify LEVEL(1), IPA performs all of the optimizations that it does at LEVEL(0), as well as subprogram inlining and global variable coalescing. IPA performs more precise alias analysis for pointer dereferences and subprogram calls.

Under IPA Level 1, many optimizations such as constant propagation and pointer analysis are performed at the intraprocedural (subprogram) level. If you specify LEVEL(2), IPA performs specific optimizations across the entire program, which can lead to significant improvement in the generated code.

The compiler option OPTIMIZE that you specify on the IPA Link step controls subsequent optimization for each partition during code generation. Regardless of the optimization level you specified during the IPA Compile step, you can request IPA optimization, regular code generation optimization, both, or neither, on the IPA Link step.

The default is IPA(LEVEL(1)).

MAP | NOMAP

Specifies that the IPA Link step will produce a listing. The listing contains a Prolog and the following sections:

- Object File Map
- Compiler Options Map
- Global Symbols Map (which may or may not appear, depending on how much global coalescence was done during optimization)
- Partition Map for each partition
- Source File Map

The default is IPA(NOMAP).

See “Using the IPA Link Step Listing” on page 272 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).

IPA(PDF) suboptions

The following section describes the IPA(PDF) suboptions.

PDF1 | NOPDF1, PDF2 | NOPDF2, PDFNAME | NOPDFNAME

The default is IPA(NOPDF1, NOPDF2, NOPDFNAME).

PDF (Profile-Directed Feedback) is a suboption of IPA that enables you to use the results from sample program execution to improve optimization near conditional branches and in frequently executed code sections. PDF allows the user to gather information about the critical paths and the usage of various parts of the application. PDF passes this information to the compiler so that the optimizer can work to make these critical paths faster. This is a three stage process that involves:

1. Performing a full IPA build with the PDF1 and PDFNAME suboptions
2. Running the trial application with representative data
3. Performing another full IPA build with the PDF2 suboption (the file indicated by PDFNAME holds the profile generated when the code was run in step 2)

Note: The trial application built from the IPA(PDF1) compiler option can only be run on the current system.

The following list describes each of the IPA(PDF) suboptions:

PDF1 An IPA suboption specified during IPA Compile and Link steps. It tells IPA to prepare the application to collect profile information.

PDF2 An IPA suboption specified during IPA Compile and Link steps. This option tells IPA to use the profile information that is provided when optimizing the application.

PDFNAME

This IPA suboption should be used with PDF1 and PDF2 to provide the name of the file that will be used for the profile information.

PDFDIR

This environment variable can be used when using IPA(PDF) in the z/OS shell. It is used to specify the directory for the profile file.

Before you begin: IPA(PDF) requires that you compile the entire application twice and is intended to be used after other debugging and tuning is finished. IPA(PDF) compiles should be performed during one of the last steps before putting the application into production. The following is a list of restrictions that applies to the procedures that follow:

- You must compile the main program with PDF for profiling information to be collected at runtime.
- Do not compile or run two different applications that use the same PDFDIR directory at the same time, unless you have used the PDFNAME(filename) option to distinguish the sets of profiling information.
- You must use the same set of compiler options at all compilation steps for a particular program otherwise PDF cannot optimize your program correctly and may even slow it down.
- You must ensure the profiling information that is provided to the compiler during the PDF2 step is for the application you are tuning.
 - If a non-qualified data set name is provided, the same userid that runs the application to collect the profiling information must perform the PDF2 step.
 - If PDFDIR is set for the PDF2 step, it must name a directory where the actual profiling information can be found.
- The profiling information is placed in the file specified by the PDFNAME(filename) suboption, where filename can be an HFS file name or a z/OS data set name (physical sequential data set or a member of a partitioned data set). For a data set name, it can be fully qualified such as PDFNAME="//HLQ.PDF" or non-qualified such as PDFNAME=//PDF, in which case the actual data set name will be 'userid.PDF', where the userid identifies the user who is executing the application built with PDF1 and building the application with PDF2. The key DCB attributes are RECFM=U and LRECL=0. In the USS environment, the profile is placed in the current working directory or in the directory that the PDFDIR environment variable names, if that variable is set. If PDFNAME(filename) is not specified, the default file name is PDFNAME(@@PDF). This file is referred to as the PDF file.
- If PDFNAME is not provided, then you need to ensure that the same environment (USS, batch/TSO, POSIX mode) is used to collect the profiling information and to perform the PDF1/PDF2 steps. This is

because PDFNAME will default to @@PDF and thus the actual location of the file is based on the environment. For example, when the POSIX(ON) run-time option is used, the PDF file will be ./@@PDF and when the POSIX(OFF) run-time option is used, the PDF file will be in data set userid.@@PDF. In order to eliminate unnecessary confusion, it is recommended that an explicit PDF file name always be provided.

- The compiler makes an attempt to delete the PDF file during PDF1 IPA(LINK) processing.
- If PDFNAME names a data set, it is strongly recommended that the data set physically exist and be allocated with sufficient space before step 2 in the process described below. Since the actual space required is based on the complexity of the application and the amount of the test data, you may run into a situation where the pre-allocated space is insufficient and you need to re-allocate the data set with larger space. The recommended attributes for the PDF data set are: RECFM=U LRECL=0.
- If you do compile a program with PDF1, it will generate profiling information when it runs, which involves some performance overhead. This overhead goes away when you recompile with PDF2 or with no PDF (NOPDF1 and NOPDF2).
- The CCNXP1B, CCNP1B, and CCNQPD1B PROCs have been created to help the batch user link with the libraries required for IPA(PDF1). Unlike our default link PROCs, these PROCs will statically bind the libraries to ensure correct operation of the information capture runs.
- Applications built with IPA(PDF1) should not be put into production because the application will be slower due to the instrumented code. The application will lose its natural reentrancy due to the sharing of the global data between the application and the statically bound PDF run-time code.

Perform the following steps to use IPA(PDF):

1. Compile some or all of the source files in a program with the IPA(PDF1) suboption on both the IPA Compile and IPA Link steps. You need to specify the OPTIMIZE(2) option, or preferably the OPTIMIZE(3) option, and the IPA(LEVEL(1|2)) option. Pay special attention to the compiler options that you use to compile the files, because you will need to use the same options later. In a large application, concentrate on those areas of the code that can benefit most from optimization, which are the paths that are executed most in a typical run of the program. Use sample data for profiling that reflects the typical runs that end users make and then the generated profiling information will show the most used paths, and the optimizer will be able to make these paths as fast as possible. You do not need to compile all of the application's code with the PDF1 suboption but you need to compile the main function with the PDF1 suboption. Link the program using CCNXP1B, CCNP1B, or CCNQPD1B in batch, or the -W1,PDF option in the z/OS shell.
2. Run the program built from step 1 all the way through using a typical data set. The program records profiling information when it finishes. You can run the program multiple times with different input data sets, and the profiling information is accumulated to provide an accurate count of how often branches are taken and blocks of code are executed.

Note: Use data that is representative of the data that will be used during a normal run of your finished program.

3. Re-build your program using the identical set of source files with the identical compiler options that you used in step 1, but change PDF1 to

PDF2. This must be done with the same compiler you use in step 1. In this second stage, the accumulated profiling information is used to fine-tune the optimizations. The resulting program does not contain profiling overhead and runs at full speed.

PDF1 at IPA Compile step causes IPA to place an indicator in the IPA object so the functions in the compilation unit are instrumented during the IPA Link step. PDF2 at IPA Compile step causes IPA to place an indicator in the IPA object so the functions in the compilation unit are optimized based on the profiling information.

PDF1 at IPA Link step causes IPA to insert instrumentation in the application code. PDF2 at IPA Link step causes IPA to optimize the application based on the profiling information collected in the file specified by PDFNAME.

Refer to "Using the IPA option" in *z/OS C/C++ Programming Guide* for an overview, examples, 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)	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)	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:

COMMONC

It indicates language constructs that are defined by XPG, many of which LANGLVL(EXTENDED) already supports. LANGLVL(ANSI) and LANGLVL(EXTENDED) do not support the following, but LANGLVL(COMMONC) does:

- Unsignedness is preserved for standard integral promotions (that is, unsigned char is promoted to unsigned int)
- Trigraphs within literals are not processed
- `sizeof` operator is permitted on bit fields
- Bit fields other than `int` are tolerated, and a warning message is issued
- Macro parameters within quotation marks are expanded
- 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 67 for more information.

- SAA** Indicates language constructs that are defined by SAA. See *z/OS C/C++ Language Reference* for more information on these language constructs.
- SAAL2** Indicates language constructs that are defined by SAA Level 2. See *z/OS C/C++ Language Reference* for more information on these language constructs.

The following suboptions are available under `z/OS C/C++`:

EXTENDED

It indicates all language constructs available with `z/OS C/C++`. It enables extensions to the ISO C standard. The macro `__EXTENDED__` is defined as 1.

- 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/C++`. If you want to make your code purely ANSI, you should redefine `_Packed` in a header file as follows:

```
#ifdef __ANSI__
#define _Packed
#endif
```

The compiler will now see the `_Packed` attribute as a blank when `LANGLVL(ANSI)` is specified at compile time, and the language level of the code will be ANSI.

LIBEXT|NOLIBEXT

For C, specifying this option affects the C/C++ run-time provided headers, which in turn control the availability of general ISO run-time extensions. In addition, it also defines the following macro and sets its value to 1:

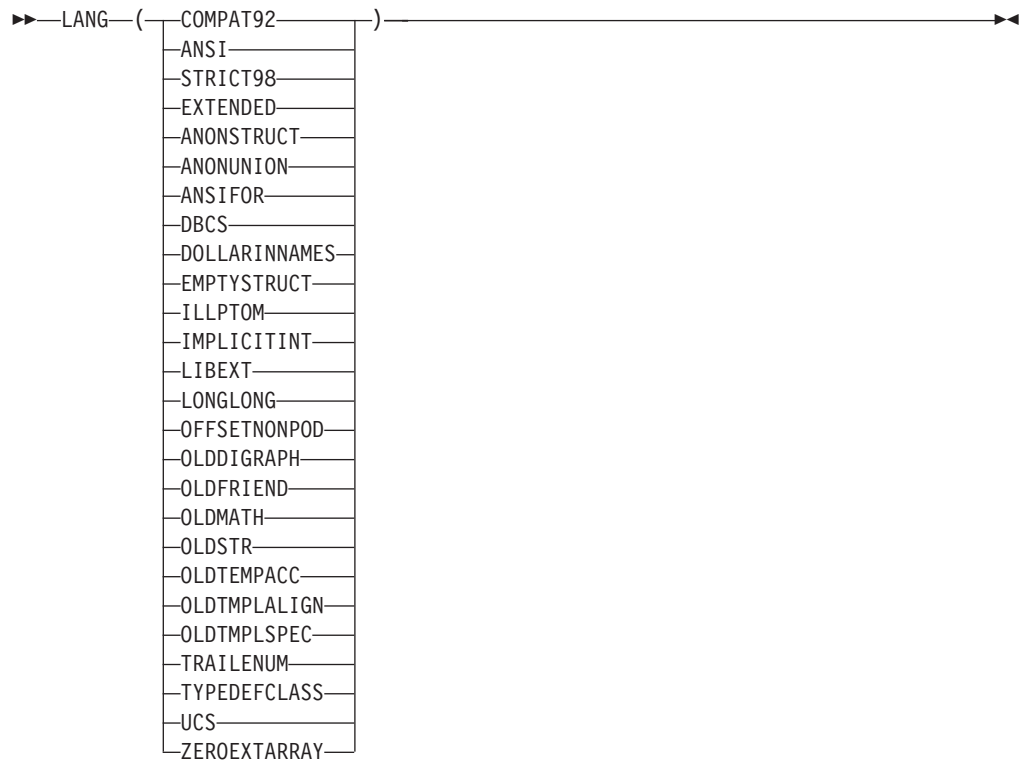
- `_MI_BUILTIN` (this macro controls the availability of machine built-in instructions. Refer to the section on using built-in functions in *z/OS C/C++ Programming Guide*)

For C++, 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)`.

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.

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 TEMPLPARSE, 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 21. Compatibility options for z/OS C/C++ V1R5 compiler

Options	Group names		
	compat92	strict98/ ansi	extended
KEYWORD(bool) NOKEYWORD(bool)	Off	On	On

Table 21. Compatibility options for z/OS C/C++ V1R5 compiler (continued)

Options	Group names		
	compat92	strict98/ ansi	extended
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(EMPTYSTRUCT NOEMPTYSTRUCT)	On	Off	On
LANGLVL(ILLPTOM NOILLPTOM)	On	Off	On
LANGLVL(IMPLICITINT NOIMPLICITINT)	On	Off	On
LANGLVL(LIBEXT NOLIBEXT)	On	Off	On
LANGLVL(LONGLONG NOLONGLONG)	On	Off	On
LANGLVL(OFFSETNONPOD NOOFFSETNONPOD)	On	Off	On
LANGLVL(OLDDIGRAPH NOOLDDIGRAPH)	Off	On	Off
LANGLVL(OLDFRIEND NOOLDFRIEND)	On	Off	On
LANGLVL(OLDMATH NOOLDMATH)	On	Off	Off
LANGLVL(OLDSTR NOOLDSTR)	On	Off	Off
LANGLVL(OLDTEMPACC NOOLDTEMPACC)	On	Off	On
LANGLVL(OLDTMPLALIGN NOOLDTMPLALIGN)	On	Off	Off
LANGLVL(OLDTMPLSPEC NOOLDTMPLSPEC)	On	Off	On

Table 21. Compatibility options for z/OS C/C++ V1R5 compiler (continued)

Options	Group names		
	compat92	strict98/ ansi	extended
LANGLVL(TRAIENUM NOTRAIENUM)	On	Off	On
LANGLVL(TYPDEFCLASS TYPDEFCLASS)	On	Off	On
LANGLVL(ZEROEXTARRAY NOZEROEXTARRAY)	Off	Off	On
RTTI NORTTI	Off	On	On
TMPLPARSE(NO ERROR WARN)	NO	WARN	NO

You can control individual language features in the z/OS V1R2 C++ compiler by using the LANGLVL and KEYWORD suboptions listed in Table 21 on page 133. 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.

Example: Anonymous structs typically are used in unions, as in the following code example:

```
union U {
    struct {
        int i:16;
        int j:16;
    };
    int k;
} u;
// ...
u.j=3;
```

When LANG(ANONSTRUCT) is in effect, you receive a warning if your code declares an anonymous struct. You can suppress the warning with SUPPRESS(CCN5017). When you build with LANG(NOANONSTRUCT) an anonymous struct is flagged as an error. Specify LANG(NOANONSTRUCT) for compliance with ISO standard C++. The default is LANG(ANONSTRUCT).

ANONUNION | NOANONUNION

This option controls what members are allowed in anonymous unions. When LANG(ANONUNION) is in effect, anonymous unions can have members of all types that ISO standard C++ allows in non-anonymous unions. For example, non-data members, such as structs, typedefs, and enumerations are allowed. Member functions, virtual functions, or objects of classes that have non-trivial default constructors, copy constructors, or destructors

cannot be members of a union, regardless of the setting of this option. When LANG(ANONUNION) is in effect, z/OS C++ allows non-data members in anonymous unions. This is an extension to ISO standard C++. When LANG(ANONUNION) is in effect, you receive a warning if your code uses the extension, unless you suppress the message with SUPPRESS(CCN6608). Specify LANG(NOANONUNION) for compliance with ISO standard C++. The default is LANG(ANONUNION).

ANSIFOR|NOANSIFOR

This option controls whether scope rules defined in the C++ standard apply to names declared in for-init statements. By default, ISO standard C++ rules are used.

Example: The following code causes a name lookup error:

```
{
  //...
  for (int i=1; i<5; i++) {
    cout << i * 2 << endl;
  }
  i = 10; // error
}
```

The reason for the error is that `i`, or any name declared within a for-init-statement, is visible only within the for statement. To correct the error, either declare `i` outside the loop or specify LANG(NOANSIFOR). Specify LANG(NOANSIFOR) to allow old language behavior. The default is LANG(ANSIFOR).

DBCS|NODBCS

This option controls whether multi-byte characters are accepted in string literals and in comments. The default 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 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.

Example: When LANG(NOEMPTYSTRUCT) is in effect, the following example will be rejected by the compiler:

```
struct S {
    ; // this line is ill-formed
};
```

The default is LANG(NOEMPTYSTRUCT).

ILLPTOM|NOILLPTOM

This controls what expressions can be used to form pointers to members. The compiler accepts some forms that are in common use, but do not conform to the C++ standard. When LANG(ILLPTOM) is in effect, the compiler allows these forms. For example, the following code defines the pointer to a function member, `p`, and initializes the address of `C::foo`, in the old style:

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(int);
};

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

```

Specify LANG(NOILLPTOM) for compliance with the C++ standard.

Example: The example must be modified to use the & operator:

```

struct C {
void foo(int);
};

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

```

The default is LANG(ILLPTOM).

IMPLICITINT|NOIMPLICITINT

This option controls whether z/OS C++ will accept missing or partially specified types as implicitly specifying int. This is no longer accepted in the standard but may exist in legacy code. When LANG(NOIMPLICITINT) is specified, all types must be fully specified. Also, when LANG(IMPLICITINT) is specified, a function declaration at namespace scope or in a member list will implicitly be declared to return int. Also, any declaration specifier sequence that does not completely specify a type will implicitly specify an integer type. Note that the effect is as if the int specifier were present. This means that the specifier const, by itself, would specify a constant integer. The following specifiers do not completely specify a type:

- auto
- const
- extern
- extern "<literal>"
- inline
- mutable
- friend
- register
- static
- typedef
- virtual
- volatile
- platform specific types (for example, _cdecl, __cdeclspec)

Note that any situation where a type is specified is affected by this option. This includes, for example, template and parameter types, exception specifications, types in expressions (eg, casts, dynamic_cast, new), and types for conversion functions. By default, LANG(EXTENDED) sets LANG(IMPLICITINT). This is an extension to the C++ standard.

Example: The return type of function MyFunction is int because it was omitted in the following code:

```

MyFunction()
{
    return 0;
}

```

Specify `LANG(NOIMPLICITINT)` for compliance with ISO standard C++.

Example: The function declaration above must be modified to:

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

The default is `LANG(IMPLICITINT)`.

`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 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 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;
```

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

Specifying this option will cause an error condition and message to be generated for the example declaration above.

```
friend class IFont;
```

The default for batch and TSO is LANG(OLDFRIEND).

OLDMATH|NOOLDMATH

This option controls which math function declarations are introduced by the `math.h` header file. For conformance with the C++ standard, the `math.h` header file declares several new functions that were not declared by `math.h` in previous releases. These new function declarations may cause an existing program to become invalid and, therefore, to fail to compile. This occurs because the new function declarations introduce the possibility of ambiguities in function overload resolution. The OLDMATH option specifies that these new function declarations are not to be introduced by the `math.h` header file, thereby eliminating the possibility of ambiguous overload resolution. The default is LANG(NOOLDMATH).

OLDSTR|NOOLDSTR

This option provides backward compatibility with previous versions of z/OS C++ and predecessor products, by controlling which string function declarations are introduced by the `string.h` and `wchar.h` header files. For conformance with the current C++ standard, `string.h` and the `wchar.h` header files declare several C++ string functions differently for C++ source files than they were declared in previous releases. These new function declarations may cause an existing C++ program to become invalid and therefore fail to compile. The LANG(OLDSTR) option specifies that the new C++ string function declarations are not to be introduced by the `string.h` and `wchar.h` header files, thereby causing only the C versions of these functions to be declared, as in previous releases. Note that when a C source file is compiled, these declarations remain unchanged from previous releases.

A number of the string function signatures that are defined in the 1989 C International Standard and the C Amendment are not const-safe.

Example: Consider the following standard C signature:

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

The behavior of this function is specified as follows:

- The `strchr` function locates the first occurrence of `c` (converted to a `char`) in the string pointed to by `s`. The terminating null character is considered to be part of the string.
- The `strchr` function returns a pointer to the located character, or a null pointer if the character does not occur in the string.

Since the parameter `s` is of type `const char *`, the string being searched by the `strchr` function is potentially composed of characters whose type is `const char`. The `strchr` function returns a pointer to one of the constituent characters of the string, but this pointer is of type `char *` even though the character that it points to is potentially of type `const char`. For this reason, `strchr` can be used to implicitly (and unintentionally) defeat the const-qualification of the string referenced by the pointer `s`.

Example: To correct this problem, the C++ standard replaces the above signature from the C standard with the following overloaded signatures:

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

Both of these overloaded functions have the same behavior as the original C version of `strchr`.

In a similar manner, the signatures of several other standard C library routines are replaced in the C++ standard. The affected routines are:

- `strchr`
- `strupbrk`
- `strrchr`
- `strstr`
- `memchr`
- `wcschr`
- `wcspbrk`
- `wcsrchr`
- `wcsstr`
- `wmemchr`

Example: Because of the changes mandated by the C++ standard, the following unsafe example will not compile in C++:

```
#include <string.h>

const char s[] = "foobar";

int main(void) {
    char * c = strchr(s, 'b');
}
```

To preserve backward compatibility with previous releases (and thus enable the code example above), specify `LANGLVL(OLDSTR)`.

OLDTEMPACC|NOOLDTEMPACC

This option controls whether access to a copy constructor to create a temporary object is always checked, even if creation of the temporary object is avoided. When `LANG(NOOLDTEMPACC)` is in effect, z/OS C++ suppresses the access checking. This is an extension to the C++ standard. When `LANG(OLDTEMPACC)` is in effect, you receive a warning if your code uses the extension, unless you disable the message. Disable the message by building with `SUPPRESS(CCN5306)` when the copy constructor is a private member, and `SUPPRESS(CCN5307)` when the copy constructor is a protected member. Specify `LANG(NOOLDTEMPACC)` for compliance with ISO standard C++.

Example: The `throw` statement in the following code causes an error because the copy constructor is a protected member of class `C`:

```
class C {
public:
    C(char *);
protected:
    C(const C&);
};

C foo() {return C("test");} // returns a copy of a C object

void f()
{
// catch and throw both make implicit copies of the thrown object
    throw C("error"); // throws a copy of a C object
    const C& r = foo(); // uses the copy of a C object created by foo()
}
```

The example code above contains three ill formed uses of the copy constructor `C(const C&)`. The default is `LANG(OLDTEMPACC)`.

`OLDTMPLALIGN|NOOLDTMPLALIGN`

This option specifies the alignment rules implemented by the compiler for nested templates. Previous versions of the compiler ignored alignment rules specified for nested templates. By default, `LANG(EXTENDED)` sets `LANG(NOOLDTMPLALIGN)` so the alignment rules are not ignored. The default for is `LANG(NOOLDTMPLALIGN)`.

`OLDTMPLSPEC|NOOLDTMPLSPEC`

This option controls whether template specializations that do not conform to the C++ standard are allowed. When `LANG(NOOLDTMPLSPEC)` is in effect, z/OS C++ allows these old specializations. This is an extension to ISO standard C++. When `LANGLVL(OLDTMPLSPEC)` is set, you receive a warning if your code uses the extension, unless you suppress the message with `SUPPRESS(CCN5080)`.

Example: You can explicitly specialize the template class `ribbon` for type `char` with the following lines:

```
template<class T> 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 is `LANG(OLDTMPLSPEC)`.

`TRAIENUM|NOTRAIENUM`

This option controls whether trailing commas are allowed in enum declarations. When `LANG(TRAIENUM)` 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(NOTRAIENUM)` for compliance with the ISO C and C++ standards. The default is `LANG(TRAIENUM)`.

`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 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 \Uhhhhhhh for 32-bit characters, where h represents a hexadecimal digit. Short identifiers of characters are specified by ISO/IEC 10646. The default is LANG(NUCS).

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.

Example: 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(ZEROEXTARRAY) is set, you receive warnings about zero-extent arrays in your code, unless you suppress the message with SUPPRESS(CCN6607). The default 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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
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 behavior 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 options 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

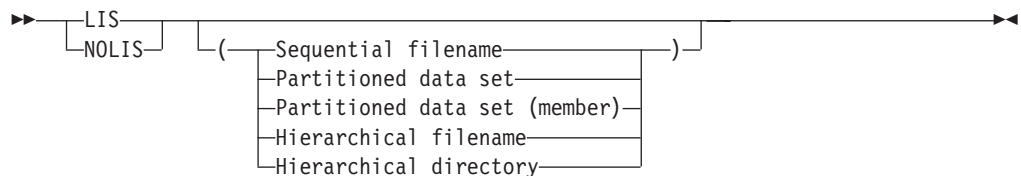
The LIBANSI option will be in effect for the IPA Link step unless the NOLIBANSI option is specified. The value of the LIBANSI option from the IPA Compile step is ignored, but is shown in the IPA Link listing Compile Option Map for reference.

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)	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. -V produces all reports for the compiler, and binder, or prelinker, and directs them to stdout. To produce only the listing (and no other reports), and write the listing to a user-specified file, use the following command:

```
-Wc,"LIST(filename)"
```

Notes:

1. Usage of information such as registers, pointers, data areas, and control blocks that are shown in the object listing are not programming interface information.
2. If you use the following form of the command in a JES3 batch environment where xxx is an unallocated data set, you may get undefined results.
LIST(xxx)
3. Statement line numbers exceeding 99999 will wrap back to 00000 for the generated assembly listing for the C/C++ source file. This may occur when the compiler LIST option is used.

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 45 and “GONUMBER | NOGONUMBER” on page 111 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.

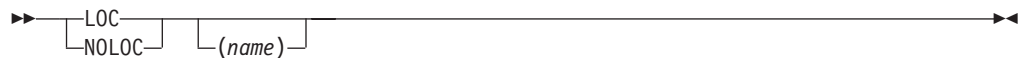
Refer to “Interactions between compiler options and IPA suboptions” on page 45 and “GONUMBER | NOGONUMBER” on page 111 for more information.

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)	z/OS UNIX System Services Utilities					
	(These utilities pick up the locale value of the environment using <code>setLocale(LC_ALL, NULL)</code>). Because the compiler runs with the <code>POSIX(0FF)</code> option, categories that are set to <code>C</code> are changed to <code>POSIX</code> .)					
	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 can 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 *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:

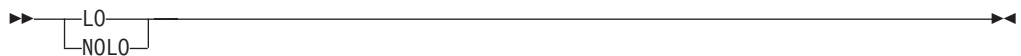
- The Prolog section of the listing displays the LOCALE or NOLOCALE option. If you specified the LOCALE option, the Prolog displays the locale and code set that are in effect.
- The Compiler Options Map listing section displays the LOCALE option active on the IPA Compile step for each IPA object. If 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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
C: NOLONGNAME	LONGNAME	LONGNAME	LONGNAME			
C++: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

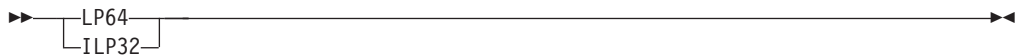
The IPA Link step ignores this option if you specify it, and uses the LONGNAME option for all partitions it generates.

LP64 | ILP32

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
ILP32						

CATEGORY: Object Code Control



The LP64 option instructs the compiler to generate AMODE 64 code utilizing the z/Architecture 64-bit instructions. ILP32 instructs the compiler to generate AMODE 31 code. This is the default and is the same mode as in previous releases of the compiler. LP64 and ILP32 are mutually exclusive. If they are specified multiple times, the compiler will take the last one.

Note: AMODE is the addressing mode of the program code generated by the compiler. In AMODE 64 and AMODE 31, 64 and 31 refer to the range of addresses that can be accessed (in other words 64-bits and 31-bits are used to form the address respectively). When there is no ambiguity, we will refer to these as 64-bit mode and 31-bit mode. Refer to the information that follows for further information on the data model.

If LP64 is specified with TARGET(z0SV1R5), no object will be generated.

LP64 and ILP32 refer to the data model used by the language. "I" is an abbreviation that represents int type, "L" represents long type, and "P" represents the pointer type. 64 and 32 refer to the size of the data types. When the ILP32 option is used, int, long and pointers are 32-bit in size. When LP64 is used, long and pointer are 64-bit in size; int remains 32-bit. As explained in a note above, the addressing mode used by LP64 is AMODE 64, and by ILP32 is AMODE 31. In the latter case, only 31 bits within the pointer are taken to form the address. For the sake of conciseness, the terms 31-bit mode and ILP32, will be used interchangeably in this document when there is no ambiguity. The same applies to 64-bit mode and LP64.

The LP64 option requires the XPLINK and GOFF compiler options. It also requires architecture level 5 or above (ARCH(5) or higher). ARCH(5), XPLINK, and GOFF are the default settings for LP64 if you don't explicitly override them. If you explicitly specify NOXPLINK, or NOGOFF, or specify an architecture level lower than 5, the compiler will issue a warning message, ignore NOXPLINK or NOGOFF, and raise the architecture level to 5.

Notes:

1. The maximum size of a GOFF object is 1 gigabyte.
2. ARCH(5) specifies the 2064 hardware models.

Since the runtime only supports IEEE functions, the default for LP64 is FLOAT(IEEE). However, the compiler supports FLOAT(HEX) as well if explicitly requested.

The LP64 option cannot be used with the TARGET option to target OS releases prior to z/OS V1R5. LP64 is ignored if it is used with the TARGET suboption prior to z/OS V1R5. Also, TARGET(IMS) is not supported; a warning message will be issued.

The prelinker cannot be used with 64-bit object modules.

Note: The z/OS platform does not support mixing 64-bit and 31-bit object files. If one compilation unit is compiled with LP64, all CUs within the program must be compiled with LP64. The binder will issue a message if it encounters mixed addressing modes during external name resolution.

In 31-bit mode, the size of long and pointers is 4 bytes and the size of wchar_t is 2 bytes. Under LP64, the size of long and pointer is 8 bytes and the size of wchar_t is 4 bytes. The size of other intrinsic datatypes remain the same between 31-bit mode and LP64. Under LP64, the type definition for size_t changes to long, and the type definition for ptrdiff_t changes to unsigned long. The following tables give the size of the intrinsic types:

Table 22. Size of intrinsic types in 64-bit mode

Type	Size (in bits)
char, unsigned char, signed char	8
short, short int, unsigned short, unsigned short int, signed short, signed short int	16
int, unsigned int, signed int	32
long, long int, unsigned long, unsigned long int, signed long, signed long int	64
long long, long long int, unsigned long long, unsigned long long int, signed long long, signed long long int	64
pointer	64

Table 23. Size of intrinsic types in 31-bit mode

Type	Size (in bits)
char, unsigned char, signed char	8
short, short int, unsigned short, unsigned short int, signed short, signed short int	16
int, unsigned int, signed int	32
long, long int, unsigned long, unsigned long int, signed long, signed long int	32
long long, long long int, unsigned long long, unsigned long long int, signed long long, signed long long int	64
pointer	32

The `__ptr32` pointer qualifier is intended to make the process of porting applications from ILP32 to LP64 easier. Use this qualifier in structure members to minimize the changes in the overall size of structures. Note that these pointers cannot refer to objects above the 31-bit address line (also known as "the bar"). In general, the program has no control over the address of a variable; the address is assigned by the implementation. It is up to the programmer to make sure that the use of `__ptr32` is appropriate within the context of the program's logic. For more information on the `__ptr32` pointer qualifier, refer to *z/OS C/C++ Language Reference*.

Notes:

1. The `long` and `wchar_t` data types also change in size.
2. LP64 only supports `OBJECTMODEL(IBM)`.

Effect on IPA Compile step

The IPA Compile step generates information for the IPA Link step for ILP32. The LP64 option affects the regular object module if you request one by specifying the `IPA(OBJECT)` option, in which case, the object module generated will be in 64-bit.

Effect on IPA Link step

The IPA Link step accepts the LP64 option, but ignores it. The DLL side deck generated by the binder has been enhanced. The side deck contains attribute flags to mark symbols exported from 64-bit DLLs; the flags are `CODE64` and `DATA64` for code and data respectively. IPA recognizes these flags.

The IPA Link step will check that all objects have a consistent data model, either ILP32 or LP64. It checks both IPA object modules and non-IPA object modules. If the IPA Link step finds a mixture of addressing modes among the object files, the compiler issues a diagnostic message and ends the compilation.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
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 152 for details.

The `#include "filename"` format of the `#include` C/C++ preprocessor directive indicates user include files. See “Using include files” on page 316 for a description of the `#include` preprocessor directive.

For further information on library search sequences, see “Search sequences for include files” on page 324.

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 `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:

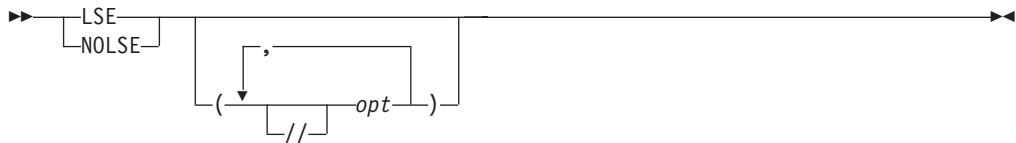
./sub/fred.h

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 321 for more details on absolute #include *filename*.

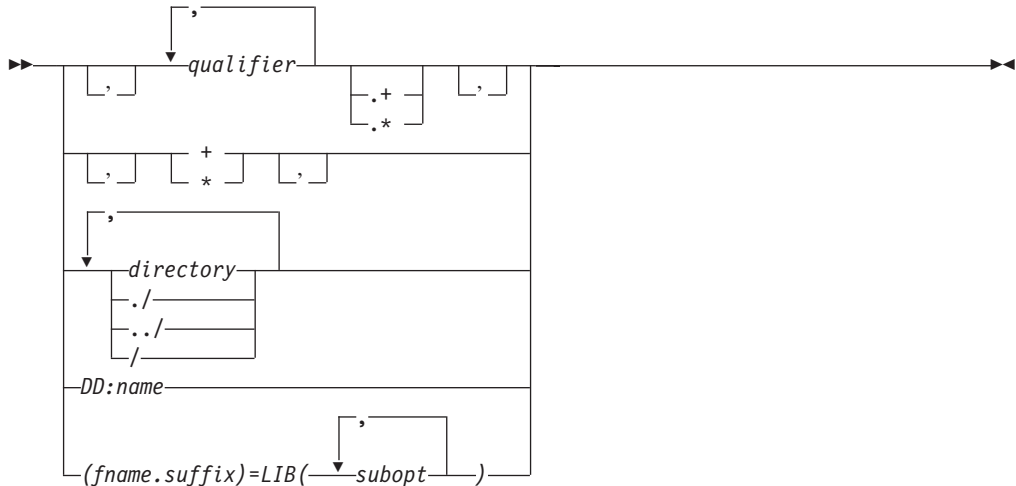
Additional syntax



You must use the double slashes (//) to specify data set library searches when you specify the 0E compiler option. (You may use them regardless of the 0E 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 pathname 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 $(fname.suffix) = (subopt, subopt, \dots)$ where:
 - `fname` is the name of the include file, or `*`
 - `suffix` is the suffix of the include file, or `*`
 - `subopt` indicates a subpath to be used in the search for the include files that match the pattern of `fname.suffix`. There should be at least one `subopt`. The possible values are:
 - LIB(`[pds, ...]`) where each `pds` is a partitioned data set name. They are searched in the same order as they are specified.
There is no effect on the search path if no `pds` is specified, but a warning is issued.
 - LIBs are cumulative; for example, LIB(A), LIB(B) is equivalent to LIB(A, B).
 - NOLIB specifies that all LIB(...) previously specified for this pattern should be ignored at this point.

When the `#include filename` matches the pattern of `fname.suffix`, the search continues according to the subopts in the order specified. An asterisk (`*`) in `fname` or `suffix` matches anything. If the compiler does not find the file, it attempts other searches according to the remaining options in LSEARCH.

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 323 for additional information on HFS files.

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

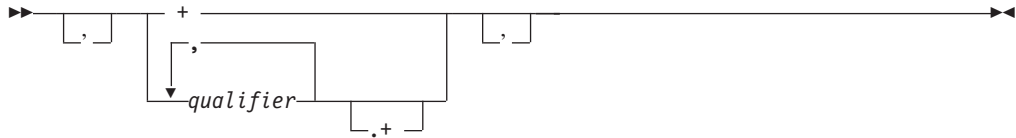
```
LSEARCH(/u/mike/myfiles)
#include "new/headers.h"
```

The resulting 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 *paths* and slashes (slashes are replaced by periods)
 2. All the periods and *qualifiers* after the left-most *qualifier*
- For the PDS member name, the left-most *qualifier* is used as the member name

See the first example in Table 24.

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 24.

See "Forming data set names with LSEARCH | SEARCH options" on page 318 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 24.

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

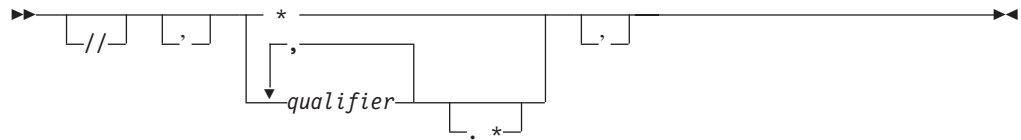
Table 24. Partitioned data set examples

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)'

Table 24. Partitioned data set examples (continued)

include Directive	LSEARCH option	Result
#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 25. 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

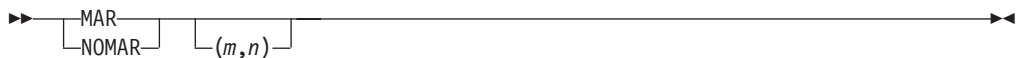
The IPA Link step accepts the LSEARCH option, but ignores it.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
C++ and C(V-format): NOMARGINS	NOMARGINS	NOMARGINS	NOMARGINS			
C(F-format): MARGINS(1,72)						

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.

The MARGINS option has the following suboptions:

- m* specifies the first column of the source input that contains valid z/OS C/C++ code. The value of *m* must be greater than 0 and less than 32761.
- n* specifies the last column of the source input that contains valid z/OS C/C++ code. The value of *n* must be greater than *m* and less than 32761. An asterisk (*) can be assigned to *n* to indicate the last column of the input record. If you specify MARGINS (9,*), the compiler scans from column 9 to the end of the record for input source statements.

If the MARGINS option is specified along with the SOURCE option in a C or C++ program, only the range specified on the MARGINS option is shown in the compiler source listing.

Notes:

1. The MARGINS option does not reformat listings.
2. If your program uses the #include preprocessor directive to include z/OS 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

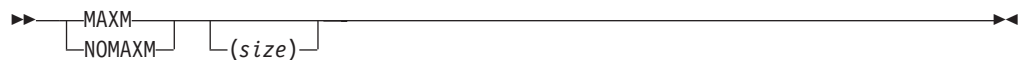
The IPA Link step accepts the MARGINS option, but ignores it.

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)	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.
5. At OPT(3), the default for MAXMEM is set to (*).

You can specify this option using the #pragma options 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.

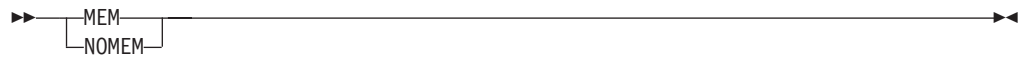
If you do not specify the option on the IPA Link step, the value that it uses for a partition is the maximum MAXMEM value you specified for the IPA Compile step for any compilation unit that provided code for that partition. The IPA Link Step Prolog listing section does not display the value of the MAXMEM 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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
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

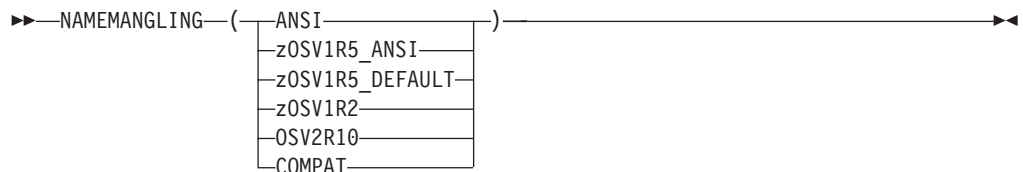
The MEMORY option has the same effect on the IPA Link step as it does on a regular compilation. If the IPA Link step fails due to an out-of-memory condition, provide additional virtual storage. If additional storage is unavailable, specify the NOMEMORY option.

NAMEMANGLING

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

z/OS C/C++ Compiler (Batch and TSO Environments)	Option Default					
	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NAMEMANGLING(zOSV1R2)						

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:

ANSI This scheme complies with the C++ standard and is the default when the LP64 compiler option is specified.

zOSV1R5_ANSI

This scheme is compatible with z/OS V1R5 link modules that were created with NAMEMANGLING(ANSI).

zOSV1R5_DEFAULT

This scheme is compatible with z/OS V1R5 link modules that were created with the default name mangling in V1R5, which is the same name mangling scheme that is compatible with z/OS V1R2 link modules that were created with NAMEMANGLING(ANSI). This name mangling scheme is the default when the ILP32 compiler option is specified.

zOSV1R2

This scheme is compatible with z/OS V1R2 link modules that were created with NAMEMANGLING(ANSI).

OSV2R10

This scheme is compatible with OS/390 V2R10 and versions prior to OS/390 V2R10.

COMPAT This scheme is equivalent to the OSV2R10 scheme.

Notes:

1. If the NAMEMANGLING compiler option is not specified, LANGLVL(EXTENDED) and LANGLVL(ANSI) set NAMEMANGLING to ZOSV1R2. LANGLVL(COMPAT92) sets NAMEMANGLING to COMPAT.
2. For information on the #pragma namemangling and #pragma namemanglingrule directives, see *z/OS C/C++ Language Reference*.

The NAMEMANGLING compiler option takes precedence over the LP64 compiler option. The LP64 compiler option takes precedence over the LANGLVL compiler option. However, when the NAMEMANGLING and LANGLVL compiler options, or the NAMEMANGLING, LANGLVL, and LP64 are specified, the last one specified between NAMEMANGLING and LANGLVL takes effect. This is to preserve the V1R2 behavior so that existing customer code is not broken.

The following table shows some examples of the NAMEMANGLING options that are in effect when certain compiler options are specified:

Table 26. Examples of NAMEMANGLING in effect

Compiler option(s) specified	NAMEMANGLING in effect
NAMEMANGLING(zOSV1R2)	zOSV1R2
LANGLVL(COMPAT92)	COMPAT
LP64	ANSI
NAMEMANGLING(zOSV1R2) LANGLVL(COMPAT92)	COMPAT
LANGLVL(COMPAT92) NAMEMANGLING(zOSV1R2)	zOSV1R2
NAMEMANGLING(zOSV1R2) LP64	zOSV1R2
LP64 NAMEMANGLING(zOSV1R2)	zOSV1R2
LANGLVL(COMPAT92) LP64	ANSI
LP64 LANGLVL(COMPAT92)	ANSI

Table 26. Examples of NAMEMANGLING in effect (continued)

Compiler option(s) specified	NAMEMANGLING in effect
NAMEMANGLING(zOSV1R2) LANGLVL(COMPAT92) LP64	COMPAT
NAMEMANGLING(zOSV1R2) LP64 LANGLVL(COMPAT92)	COMPAT
LP64 NAMEMANGLING(zOSV1R2) LANGLVL(COMPAT92)	COMPAT
LP64 LANGLVL(COMPAT92) NAMEMANGLING(zOSV1R2)	zOSV1R2
LANGLVL(COMPAT92) LP64 NAMEMANGLING(zOSV1R2)	zOSV1R2
LANGLVL(COMPAT92) NAMEMANGLING(zOSV1R2) LP64	zOSV1R2

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

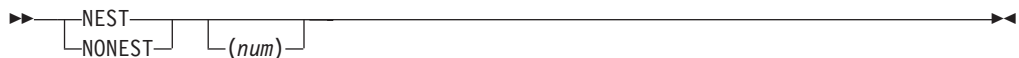
The IPA Link step accepts the NAMEMANGLING option, but ignores it.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
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

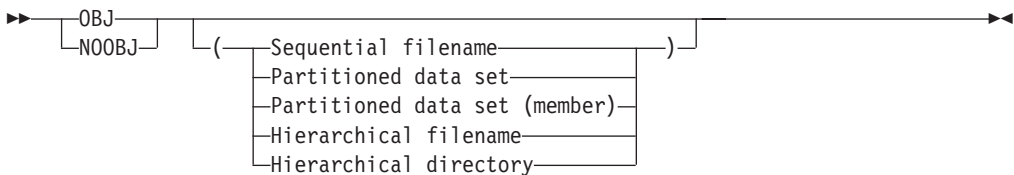
The IPA Link step accepts the NESTINC option, but ignores it.

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)	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 (<i>file_name.o</i>)	OBJECT (<i>file_name.o</i>)	OBJECT (<i>file_name.o</i>)	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 options 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 122 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 122 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)	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 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.

Note: In order to use the OBJECTMODEL(IBM) option, the XPLINK option must be specified. If XPLINK is not specified, the compiler will issue a warning and use the default OBJECTMODEL(COMPAT) setting.

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.

Example: 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

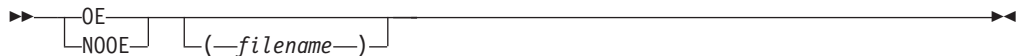
The IPA Link step does not accept the OBJECTMODEL option. The compiler issues a warning message if you specify this option in the IPA Link step.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
	NOOE	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 when searching for files specified with `#include` directives. These rules state that the path of the file currently being processed is the path used as the starting point for searches of include files contained in that file.

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

Example: The following specifications have the same result:

```
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.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
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.

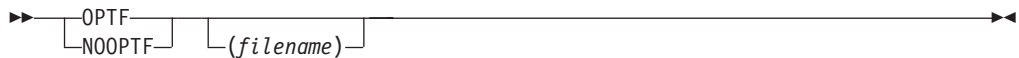
The OFFSET option that you specified on the IPA Compile step has no effect on the IPA Link step.

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)	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.

Example: The following specifications have the same result:

```

CXX HELLO (NOOPTF(/hello.opt) OPTF
CXX HELLO (OPTF(/hello.opt)
  
```

The options are specified in a free format with the same syntax as they would have on the command line or in JCL. The code points for the special characters \f, \v, and \t are whitespace characters. Everything that is specified in the file is taken to be part of a compiler option (except for the continuation character), and unrecognized entries are flagged. Nothing on a line is ignored.

If the record format of the options file is fixed and the record length is greater than 72, columns 73 to the end-of-line are treated as sequence numbers and are ignored.

Notes:

1. Comments are supported in an option file used in the OPTFILE option. When a line begins with the # character, the entire line is ignored, including any continuation character. The option files are encoded in the IBM-1047 codepage.
2. You cannot nest the OPTFILE option. If the OPTFILE option is also used in the file that is specified by another OPTFILE option, it is ignored.
3. If you specify NOOPTFILE after a valid OPTFILE, it does not undo the effect of the previous OPTFILE. This is because the compiler has already processed the options in the options file that you specified with OPTFILE. The only reason to use NOOPTFILE is to specify an option file name that a later specification of OPTFILE can use.
4. If the file cannot be opened or cannot be read, a warning message is issued and the OPTFILE option is ignored.
5. The options file can be an empty file.
6. **Example:** You can use an option file only once in a compilation. If you use the following options:

```
OPTFILE(DD:OF) OPTFILE
```

the compiler processes the option OPTFILE(DD:OF), but the second option OPTFILE is not processed. A diagnostic message is produced, because the second specification of OPTFILE uses the same option file as the first.

Example: You can specify OPTFILE more than once in a compilation, if you use a different options file with each specification:

```
OPTFILE(DD:OF) OPTFILE(DD:OF1)
```

Examples

1. Suppose that you use the following JCL:

```
// CPARAM='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:

```
// CPARAM='SO OBJECT LONGNAME EXPORTALL'
```

2. Suppose that you include the following in the JCL:

```
// CPARAM='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:

```
// CPARAM='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 ',
//      CPARAM='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

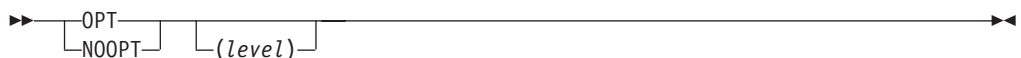
The OPTFILE option has the same effect on the IPA Link step as it does on a regular compilation.

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)	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	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)
IPA Link: OPTIMIZE(2)						

CATEGORY: Object Code Control



Note: When the compiler is invoked using the c89, cc, c++, x1c or x1C commands under z/OS UNIX System Services, optimization level is specified by the compiler flags -0 (the letter) or -0 (the number), -2, or -3. The OPTIMIZE option has no effect for those commands.

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.
- 3** Performs additional optimizations to those performed with OPTIMIZE(2). OPTIMIZE(3) is recommended when the desire for runtime improvement outweighs the concern for minimizing compilation resources. Increasing the level of optimization may or may not result in additional performance improvements, depending on whether additional analysis detects further opportunities for optimization. Compilation may require more time and machine resources.

Use the STRICT option with OPTIMIZE(3) to turn off the aggressive optimizations that might change the semantics of a program. STRICT combined with OPTIMIZE(3) invokes all the optimizations performed at OPTIMIZE(2) as well as further loop optimizations. The STRICT compiler option must appear after the OPTIMIZE(3) option, otherwise it is ignored.

The aggressive optimizations performed when you specify OPTIMIZE(3) are:

- Aggressive code motion, and scheduling on computations that have the potential to raise an exception, are allowed.
- Conformance to IEEE rules are relaxed. With OPTIMIZE(2), certain optimizations are not performed because they may produce an incorrect sign in cases with a zero result, and because they remove an arithmetic operation that may cause some type of floating-point exception. For example, $X + 0.0$ is not folded to X because, under IEEE rules, $-0.0 + 0.0 = 0.0$, which is $-X$. In some other cases, some optimizations may perform optimizations that yield a zero result with the wrong sign. For example, $X - Y * Z$ may result in a -0.0 where the original computation would produce 0.0 . In most cases, the difference in the results is not important to an application and OPTIMIZE(3) allows these optimizations.
- Floating-point expressions may be rewritten. Computations such as $a*b*c$ may be rewritten as $a*c*b$ if, for example, an opportunity exists to get a common subexpressions by such rearrangement. Replacing a divide with a multiply by the reciprocal is another example of reassociating floating-point computations.

no level

OPTIMIZE specified with no level defaults, depending on the compilation environment and IPA mode. See the Option Default table above for details.

You can specify this option using the #pragma options 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 118 for more information about the `INLINE` option and the optimization information in *z/OS C/C++ Programming Guide*.

If you specify `OPTIMIZE` with `TEST` or `DEBUG`, you can only set breakpoints at function call, function entry, function exit, and function return points.

The option `INLINE` is automatically turned on when you specify `OPTIMIZE`, unless you have explicitly specified the `NOINLINE` option.

A comment that notes the level of optimization will be generated in your object module to aid you in diagnosing your program.

Effect of `ANSIALIAS`: When the `ANSIALIAS` option is specified, the optimizer assumes that pointers can point only to objects of the same type, and performs more aggressive optimization. However, if this assumption is not true and `ANSIALIAS` is specified, wrong program code could be generated. If you are not sure, use `NOANSIALIAS`. For more information, see “`ANSIALIAS | NOANSIALIAS`” on page 67.

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 122 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 122 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)	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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
PLIST(HOST)						

CATEGORY: Program Execution

▶▶ PLIST—(HOST)————▶▶
 └── OS ─┘

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 *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)	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.

Example: 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

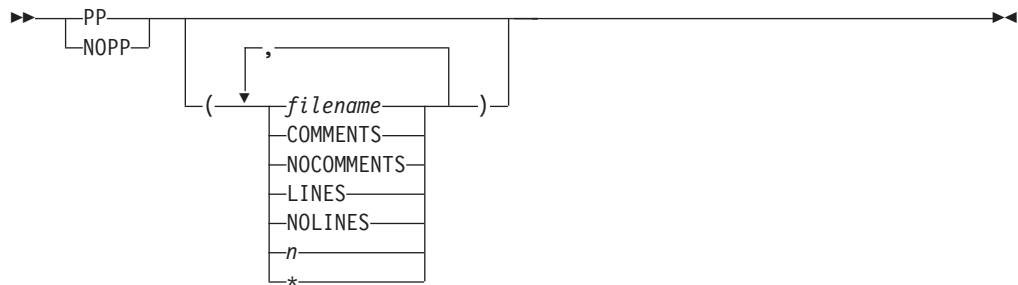
The IPA Link step issues a diagnostic message if you specify the PORT option for that step.

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)	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 PPOONLY 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. PPOONLY also removes conditional compilation constructs like #if, and #ifdef.

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 PPOONLY 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 .iextension.

Note: If you are using the x1c utility and you do not specify the file name, the preprocessed output goes to stdout. If you also specify the -E or -P flag option, the output file is determined by the flag option specified. If both -E and -P are specified, the output file is determined by the -E option. -E flag option maps to PP(stdout). -P maps to PP(default_name). default_name is constructed using the source file name as the base and the suffix is replaced with the appropriate suffix, as defined by the isuffix, isuffix_host, ixsuffix, and ixsuffix_host configuration file attributes.

|
|
|
|

See Chapter 19, “xlc — Compiler invocation using a customizable configuration file,” on page 513 for further information on the xlc utility.

<i>n</i>	If a parameter <i>n</i> , which is an integer between 2 and 32760 inclusive, is specified, all lines are folded at column <i>n</i> .
*	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 PPOONLY suboptions are cumulative. If you specify suboptions in multiple instances of PPOONLY and NOPPOONLY, all the suboptions are combined and used for the last occurrence of the option.

Example: The following three specifications have the same result:

```
CXX HELLO (NOPPOONLY(/aa.exp) PPOONLY(LINES) PPOONLY(NOLINES)
CXX HELLO (PPOONLY(/aa.exp,LINES,NOLINES)
CXX HELLO (PPOONLY(/aa.exp,NOLINES)
```

All #line and #pragma preprocessor directives (except for margins and sequence directives) remain. When you specify PPOONLY(*), #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 PPOONLY option, the compiler turns on the TERMINAL option. If you specify the SHOWINC, XREF, AGGREGATE, or EXPMAC options with the PPOONLY option, the compiler issues a warning, and ignores the options.

If you specify the PPOONLY 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 *z/OS C/C++ Programming Guide*.

The NOPPOONLY option specifies that both the preprocessor and the compiler are to be run against the source file.

If you specify both PPOONLY and NOPPOONLY, 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 PPOONLY, they must use -E and -C. The {_ELINES} envvar is also relevant (for further information on {_ELINES}, refer to “Environment variables” on page 486). The output always goes to stdout.

Effect on IPA Compile step

The PPOONLY 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 PPOONLY 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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
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 does not apply to programs invoked 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()`.

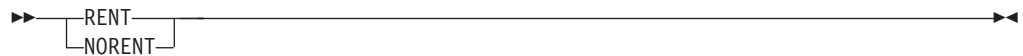
If you specify this option on both the IPA Compile and the IPA Link steps, the setting on the IPA Link step overrides the setting on the IPA Compile step. This situation occurs whether you use REDIR and NOREDIR as compiler options or specify them using the `#pragma runopts` directive (on the IPA Compile 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)	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 *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 9, “Binding z/OS C/C++ programs,” on page 355, or the prelinker, which is described in Appendix A, “Prelinking and linking z/OS C/C++ programs,” on page 535.

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 options 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.

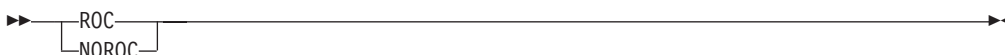
If you generate an IPA Link listing by using the LIST or MAP compiler option, the IPA Link step generates a Partition Map listing section for each partition. If any symbols within a partition are reentrant, the options section of the Partition Map displays the RENT compiler option.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
For C: NOROCONST						
For C++: 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.

When the ROCONST option is specified, `const` qualified variables are not placed into the Writeable Static Area (WSA), even if the RENT option is in effect. 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 *z/OS 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 export` 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 is maintained during IPA optimization and code generation.

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

The ROCONST setting for a partition is set to the specification of the first subprogram that is placed in the partition. 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)	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 *z/OS 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)	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

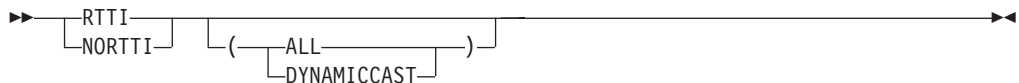
The IPA Link step merges and optimizes the application code, and then divides it into sections for code generation. Each of these section is a partition. The IPA Link step uses information from the IPA Compile step to ensure that an object is included in a compatible partition. Refer to the “FLOAT” on page 105 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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
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

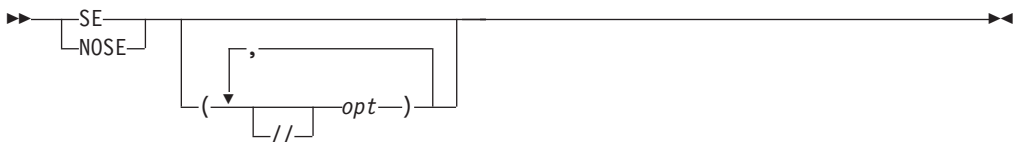
The IPA Link step does not accept the RTTI option. The compiler issues a warning message if you specify this option in the IPA Link step.

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)	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: <ul style="list-style-type: none"> • Additional include search directories identified by the c89 -I options. Refer to Appendix F for more information. • z/OS UNIX System Services 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 486 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 316 for a description of the #include preprocessor directive.

For further information on library search sequences, see “Search sequences for include files” on page 324.

The suboptions for the SEARCH option are identical to those for the LSEARCH option, as described on page “LSEARCH | NOLSEARCH” on page 150.

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 321 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

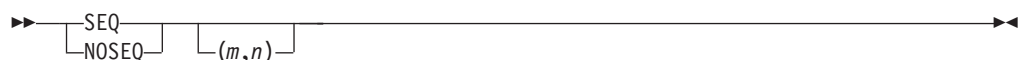
The IPA Link step accepts the SEARCH option, but ignores it.

SEQUENCE | NOSEQUENCE

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

z/OS C/C++ Compiler (Batch and TSO Environments)	Option Default					
	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
C++ and C(V-format and HFS): NOSEQUENCE	NOSEQUENCE	NOSEQUENCE	NOSEQUENCE			
C(F-format): SEQUENCE (73,80)						

CATEGORY: Input Source File Processing Control



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.

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.

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.

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)	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 options 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

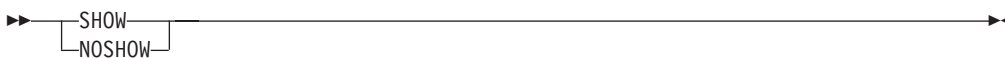
If you specify the SERVICE option on the IPA Compile step, or specify #pragma options(SERVICE) in your code, it has no effect on the IPA Link step. Only the SERVICE option you specify on the IPA Link step affects the generation of the service string for that step.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOSHOWINC						

CATEGORY: Listing



The SHOWINC option instructs the compiler to show, in both the compiler listing and the Pseudo-Assembly 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

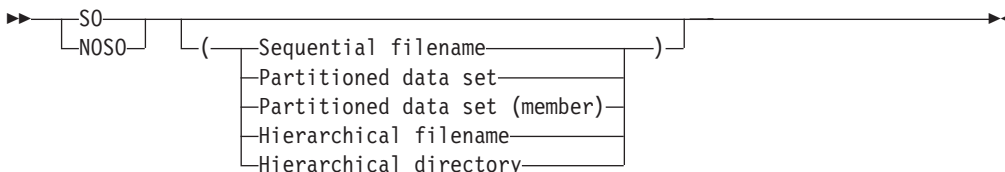
The IPA Link step accepts the SHOWINC option, but ignores it.

SOURCE | NOSOURCE

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

z/OS C/C++ Compiler (Batch and TSO Environments)	Option Default					
	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 SYSPRT 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.

Example: The following specifications have the same result:

```
CXX HELLO (NOSO(/hello.lst) S0
CXX HELLO (S0(/hello.lst)
```

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.lst) S0(/n1.lst) NOSO(/test.lst) S0
CXX HELLO (S0(/test.lst)
```

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
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 saves the contents of some registers in 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. 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 options 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.

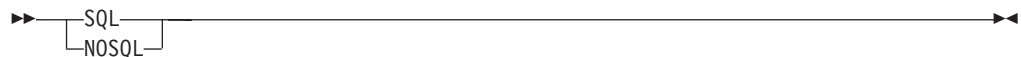
The Partition Map sections of the IPA Link step listing and the object module END information section display the value of the SPILL option. The Partition Map also displays any subprogram-specific SPILL values.

SQL | NOSQL

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOSQL						

CATEGORY: Object Code Control



The SQL compiler option enables the C/C++ compiler to process embedded SQL statements. Use the SQL compiler option to enable the SQL statement coprocessor capability and to specify SQL statement coprocessor options. The SQL coprocessor options are only passed to the SQL statement coprocessor; the C/C++ compiler does not act on any of the options. Refer to *DB2 Application Programming and SQL Guide* for further information.

You may use this option to compile C and C++ programs containing embedded SQL statements, that have not been pre-compiled by the DB2 Precompiler. When you specify this option, the compiler writes the database request module (DBRM Bind file) to the ddname DBRMLIB.

Note: To use this option, the C/C++ compiler requires access to DB2 Version 7 or later. Ensure you specify the DB2 load module data set in your compile step STEPLIB.

To use this option with the supplied proc, specify the following items in your JCL:

```
//SQLCOMP EXEC EDCC,
// CPARM='SQL',
// INFILE=PAYROLL.SOURCE(MASTER) '
//STEPLIB DD
// DD
// DD
// DD DSN=DSN710.SDNSLOAD,DISP=SHR
//DBRMLIB DD DSN=PAYROLL.DBRMLIB.DATA(MASTER),DISP=SHR
```

An SQL INCLUDE statement is treated identically to an #include directive. The following two lines are processed the same way by the compiler:

```
EXEC SQL INCLUDE name;
#include "name"
```

The library search order for SQL INCLUDE statements is the same as specified in the LSEARCH option or the USERLIB ddname. Nested SQL INCLUDE statements, that are not supported with the DB2 Precompiler, are supported by the SQL compiler option.

For C++, host variable names do not need to be unique, as they are previously required to be by the DB2 Precompiler. You may declare host variables, using the SQL BEGIN DECLARE SECTION and SQL END DECLARE SECTION statements, of the same name but in different lexical scopes.

Example: The same lexical scoping rules for C/C++ variables apply when they are used as host variables in SQL statements:

```
EXEC SQL BEGIN DECLARE SECTION;
int salary;
EXEC SQL END DECLARE SECTION;

main() {
  EXEC SQL BEGIN DECLARE SECTION; /* (1) */
  int salary;
  EXEC SQL END DECLARE SECTION; /* (2) */

  /* The local variable salary will be used here */
  EXEC SQL SELECT SALARY INTO :salary FROM ctab WHERE EMPNO = 12345;
}
```

If the local variable has not been declared as host variable, that is, the SQL BEGIN DECLARE SECTION statement (1) and SQL END DECLARE SECTION statement (2) are missing, you will get a compiler error.

Effect on IPA Compile step

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

Effect on IPA Link step

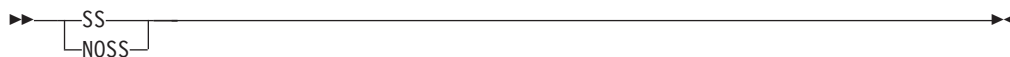
The IPA Link step accepts but ignores this option.

SSSCOMM | 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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
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.

Example: 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:

```
//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

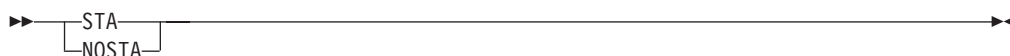
The IPA Link step accepts the SSCOMM option, but ignores it.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
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 options` 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

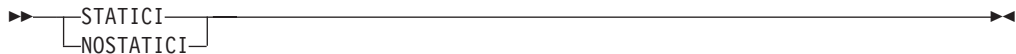
The IPA Link step uses the value of the `START` option that you specify for that step. It does not use the value that you specify for the IPA Compile step.

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)	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. (As of z/OS V1R2, the 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 previous versions of the C++ 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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
For NOOPT and OPT(2), the default is STRICT. For OPT(3), the default is NOSTRICT.						

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` sets `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

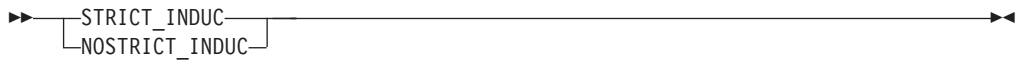
The IPA Link step merges and optimizes the application code, and then divides it into sections for code generation. Each of these sections is a partition. The IPA Link step uses information from the IPA Compile step to ensure that an object is included in a compatible partition. See “`FLOAT`” on page 105 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)	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.

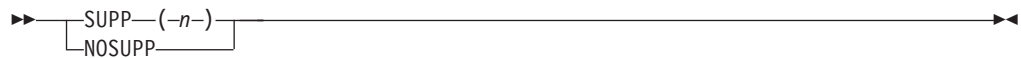
You can override the setting of STRICT_INDUCTION by specifying the option on the IPA Link step. If you do so, all partitions will contain that value, and the prolog section of the IPA Link step listing will display the value.

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)	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 to the listing file and to the terminal. For C, the message ID range that is affected is CCN3000 through CCN3999. The message ID range that is affected for C++ is CCN5001 to CCN6999, CCN7500 to CCN7999 and CCN8000 to CCN8999. Note that this option has no effect on linker or operating system messages. Compiler messages that cause compilation to stop, such as (S) and (U) level messages cannot be suppressed.

Effect on IPA Compile step

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

Effect on IPA Link step

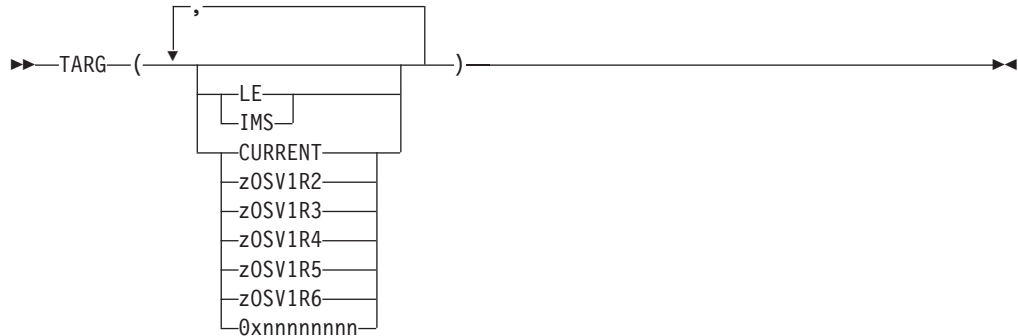
The IPA Link step accepts the SUPPRESS option, but ignores it.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
TARGET (LE, CURRENT)						

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 targeted release. With the TARGET option, you can compile and link an application on a higher level system, and run the application on a lower level system.

To use the TARGET option, select a run-time environment of either LE or IMS. Then select the desired release, for example, z0SV1R2. 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. TARGET(IMS) is not supported with LP64.

For more information about these suboptions refer to "TARGET run-time environment suboptions (LE,IMS)" on page 202.

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 is included. As the compiler is included with z/OS V1R6, TARGET(CURRENT) is the same as TARGET(z0SV1R6). This is the default.³

3. Note that for some releases of z/OS, z/OS C/C++ might not include a new version of the compiler. The same version of the compiler is then included with more than one z/OS release. The compiler is designed to run on all these z/OS releases. In this case, the compiler sets CURRENT to the z/OS release on which it is running. (It does so by querying the Language Environment Library version of the system.) You can specify an z0SVxRy suboption that corresponds to a release that is earlier or the same as CURRENT. You cannot specify an z0SVxRy suboption that corresponds to a release later than CURRENT.

Note: The binder has an option called COMPAT that must be used when generating code to be run on an earlier version of the operating system. The C/C++ cataloged procedures and the USS utilities c89/c++/cc set the COMPAT option to CURR. As of z/OS V1R3, the binder output defaults to PM4. A PM4 object module will fail to load on operating systems prior to z/OS V1R3. If you are building on z/OS V1R3, and targeting back to earlier releases, you will need to pass COMPAT=PM3 or COMPAT=PM2 to the binder. For more information on the COMPAT binder option, see *z/OS MVS Program Management: User's Guide and Reference*.

TARGET(zOSV1R2)	Generates object code to run under z/OS Version 1 Release 2 and subsequent releases.
TARGET(zOSV1R3)	Generates object code to run under z/OS Version 1 Release 3 and subsequent releases.
TARGET(zOSV1R4)	Generates object code to run under z/OS Version 1 Release 4 and subsequent releases.
TARGET(zOSV1R5)	Generates object code to run under z/OS Version 1 Release 5 and subsequent releases.
TARGET(zOSV1R6)	Generates object code to run under z/OS Version 1 Release 6 and subsequent releases.
TARGET(0xnnnnnnnn)	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 <code>__TARGET_LIB__</code> macro. For more information on using this literal, please see “Using the hexadecimal string literal suboption” on page 200.

For more information about these suboptions refer to “TARGET release suboptions” on page 200.

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,0x220a0000, IMS, zOSV1R2, LE) resolves to TARGET(LE, zOSV1R2). TARGET(LE, 0x220a0000, IMS, zOSV1R2) resolves to TARGET(IMS, zOSV1R2). TARGET(LE, 0x220a0000, IMS) resolves to TARGET(IMS, 0x220a0000).

The default value of the ARCHITECTURE compiler option depends on the value of the TARGET release suboption. For TARGET(zOSV1R6), the default is ARCH(5). For TARGET(zOSV1R2) to TARGET(zOSV1R5), the default is ARCH(2).

TARGET release suboptions

The TARGET release suboptions (CURRENT, zOSV1R2, zOSV1R3, zOSV1R4, zOSV1R5 and zOSV1R6) help you to generate code that can be executed on a particular release of a z/OS system, and on subsequent releases.

In order to use these suboptions, you must:

- Use the z/OS V1R6 class library header files (found in the CBC.SCLBH.* data sets) during compilation
- If you are targeting 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 side-decks are in CBC.SCLBSID.

For example, to generate code that will execute on an OS/390 V2R10 system, using a z/OS V1R6 application development system:

- Use the z/OS V1R6 Language Environment data sets (CEE.SCEE*) during the assembly, compilation, pre-link, link-edit, and bind phases.
- Use the OS/390 V2R10 class library data sets (SCLBCPP, SCLB0BC, SCLB0XL, SCLBSID, SCLBXL) during pre-link, link-edit, and bind. Use the z/OS V1R6 class library header data sets (CBC.SCLBH.*) during compilation.
- Specify the compiler option TARGET(OSV2R10) on the C/C++ compilers. Note: The programmer is responsible for ensuring that they are not exploiting any Language Environment functions that are unavailable on OS/390 V2R10.

See Appendix A, “Prelinking and linking z/OS C/C++ programs,” on page 535 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(OSVxRy) release suboptions, the compiler sets the `__TARGET_LIB__` macro. See *z/OS C/C++ Language Reference* for more information about this macro.

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 targeted release during pre-link, link-edit, and bind.
- Ensure that any other libraries incorporated in the application, are compatible with the target release.

While there are no restrictions on the use of ARCH and TUNE with TARGET, ensure that the level specified is consistent with the target hardware.

TARGET Release Suboption	Restrictions
<ul style="list-style-type: none"> • CURRENT • zOSV1R6 	TARGET(IMS) is not supported for LP64. The compiler will issue a message and ignore the option.
<ul style="list-style-type: none"> • zOSV1R5 	The compiler will issue a message and ignore the option. The trial application built from the IPA(PDF1) compiler option can only be run on the current system. LP64 is not supported. ARCH(2) is the default.

TARGET Release Suboption	Restrictions
<ul style="list-style-type: none"> • zOSV1R4 • zOSV1R3 • zOSV1R2 	LP64 and WARN64 are not supported.
<ul style="list-style-type: none"> • zOSV1R1 • OSV2R10 	Standard C++ features that are not supported in the targeted run time are disabled. The RTTI compiler option is not supported. ARCH(2) 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(zOSVxRy) is not permitted in a #pragma target() directive.

If you specify TARGET(zOSVxRy) 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 zOSV1R2. 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)	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 103 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

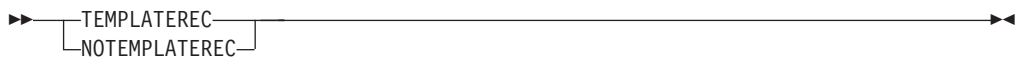
The IPA Link step issues a diagnostic message if you specify the TEMPINC option for that step.

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)	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 does not accept the `TEMPLATERECOMPILE` option. The compiler issues a warning message if you specify this option in the IPA Link step.

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)	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`. If a file currently exists with the name of the file name used for `TEMPLATEREGISTRY`, then that file will be overwritten. For more information on using the `TEMPLATEREGISTRY` option, see *z/OS 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.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
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 Compile step

The `TERMINAL` compiler option has the same effect on the IPA Compile step as it does on a regular compile step.

Effect on IPA Link step

The `TERMINAL` compiler option has the same effect on the IPA Link step as it does on a regular compile step.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
Default for C++ compile: NOTEST (HOOK)	NOTEST-g sets TEST	NOTEST-g sets TEST	NOTEST-g sets TEST	NOTEST	NOTEST	NOTEST
Default for C compile: NOTEST (HOOK, SYM, BLOCK, LINE, PATH)	-s sets NOTEST	-s sets NOTEST	-s sets 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	<ul style="list-style-type: none"> 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. 	<ul style="list-style-type: none"> Generates Function Entry, Function Exit, Function Call and Function Return hooks. Does not generate symbol information.
NOHOOK	<ul style="list-style-type: none"> 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. 	<ul style="list-style-type: none"> Does not generate any hooks. Does not generate symbol information.

The TEST suboptions generate symbol tables and program hooks. Debug Tool uses these tables and hooks to debug your program. The Performance Analyzer uses these hooks to trace your program. The choices you make when compiling your program affect the amount of Debug Tool function available during your debugging session. These choices also impact the ability of the Performance Analyzer to trace your program.

To look at the flow of your code with Debug Tool, or to trace the flow of your code with the Performance Analyzer, use the H00K 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 N0OPT is in effect, and you use the H00K suboption, the debugger runs slower, but all Debug Tool commands such as AT ENTRY * are available. You must specify the H00K suboption in order to trace your program with the Performance Analyzer.

If you specify the N0TEST option, debugging information is not generated and you cannot trace your program with the Performance Analyzer.

In order for the debugger to access the source lines, the primary source file of a compilation unit should come from one file (or sequential data set or PDS member), and not be the result of concatenated DD statements. This is because only the name of the first data set is known to the compiler reading the concatenation; the debug information generated in this case would contain only the first data set name. All the source files, including header files, should not be temporary files, and should be available to the debugger under the same name as used during compilation.

You can use the CSECT option with the TEST option to place your debug information in a named CSECT. This enables the compiler and linker to collect the debug information in your module together, which may improve the run-time performance of your program.

If you specify the INLINE and TEST compiler options when N0OPTIMIZE is in effect, INLINE is ignored.

If you specify the TEST option, the compiler turns on GONUMBER.

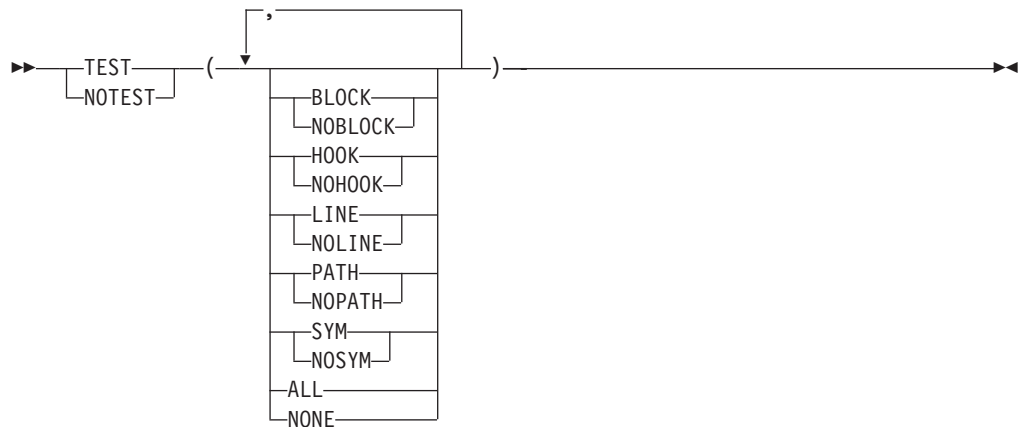
You can specify this option using the #pragma options directive for C.

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.

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 Debug Tool where to gain control of your program.

The symbol table suboption SYM regulates the inclusion of symbol tables into the object output of the compiler. Debug Tool uses the symbol tables to obtain information about the variables in the program.

SYM Generates symbol tables in the object output of the program that give you access to variables and other symbol information.

- You can reference all program variables by name, allowing you to examine them or use them in expressions.
- You can use the Debug Tool command GOTO to branch to a label (paragraph or section name).
- The Performance Analyzer does not use symbol information. Specify NOSYM if you want to trace the program with the Performance Analyzer.

BLOCK Inserts only block entry and exit hooks into the object output of the program. A block is any number of data definitions, declarations, or statements that are enclosed within a single set of braces. BLOCK also creates entry hooks and exit hooks for nested blocks. If SYM is enabled, symbol tables are generated for variables local to these nested blocks.

- You can only gain control at entry and exit of blocks.
- Issuing a command such as STEP causes your program to run, until it reaches the exit point.
- The Performance Analyzer does not use block entry and exit hooks. Specify NOBLOCK if you want to trace the program with the Performance Analyzer.

LINE Generates hooks at most executable statements. Hooks are not generated for the following:

- Lines that identify blocks (lines that contain braces)
- Null statements
- Labels
- Statements that begin in an #include file
- The Performance Analyzer does not use statement hooks. Specify NOLINE if you want to trace the program with the Performance Analyzer.

- PATH** Generates hooks at all path points; for example, hooks are inserted at if-then-else points.
- This option does not influence the generation of entry and exit hooks for nested blocks. You must specify the **BLOCK** suboption if you desire such hooks.
 - Debug Tool can gain control only at path points and block entry and exit points. If you attempt to **STEP** through your program, Debug Tool gains control only at statements that coincide with path points, giving the appearance that not all statements are executed.
 - The Debug Tool command **GOTO** is valid only for statements and labels that coincide with path points.
 - The Performance Analyzer uses function call and function return hooks. Specify **PATH** if you want to trace the program with the Performance Analyzer.
- ALL** Inserts block and line hooks, and generates symbol table. Hooks are generated at all statements, all path points (if-then-else, calls, and so on), and all function entry and exit points.
- ALL** is equivalent to **TEST(HOOK, BLOCK, LINE, PATH, SYM)**.
- NONE** Generates all compiled-in hooks only at function entry and exit points. Block hooks and line hooks are not inserted, and the symbol tables are suppressed.
- TEST(NONE)** is equivalent to **TEST(HOOK, NOBLOCK, NOLINE, NOPATH, NOSYM)**.

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 **H00K** suboption.

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 **N00BJECT** 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(H00K)**, **TEST(N0H00K)**, and **NOTEST** options. If you specify **TEST(H00K)** or **TEST**, the **IPA Link** step generates function call, entry, exit, and return hooks. It does not generate symbol table information. If you

specify TEST(NOH00K), 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.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
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 template 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 TEMPLPARSE option for that step.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
TUNE(5) If the TUNE level is lower than the specified ARCH level, the compiler forces TUNE to match the ARCH level or uses the default TUNE level, whichever is greater.						

CATEGORY: Object Code Control

►►—TUN(n)—◄◄

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 70.

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 generates code that is executable on all models but that is optimized for the following and follow-on models: 9672-Rx5 (G4), 9672-xx6 (G5), and 9672-xx7 (G6).
- 4 This option generates code that is executable on all models but that is optimized for the model 2064-100 (z/900).
- 5 This option is the default. This option generates code that is executable on all models but that is optimized for the model 2064-100 (z/900) in z/Architecture mode.
- 6 This option generates code that is executable on all models, but is optimized for the 2084-xxx models.

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 options` 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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
no action						

CATEGORY: Preprocessor

►► UNDEF (*name*) ►►

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.

Note: c89 preprocesses -D and -U flags before passing them to the compiler. x1c just passes -D and -U to the compiler, which interprets them as DEFINE and UNDEFINE. For more information, see Chapter 18, “c89 — Compiler invocation using host environment variables,” on page 471 or Chapter 19, “xlc — Compiler invocation using a customizable configuration file,” on page 513.

Effect on IPA Compile step

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

Effect on IPA Link step

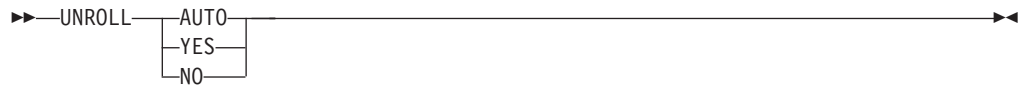
The IPA Link step accepts the UNDEFINE option, but ignores it.

UNROLL

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
UNROLL(AUTO)						

CATEGORY: Object Code Control



The UNROLL compiler option instructs the compiler to perform loop unrolling, which is an optimization that replicates a loop body multiple times, and adjusts the loop control code accordingly. Loop unrolling exposes instruction level parallelism for instruction scheduling and software pipelining and thus can improve a program's performance. It also increases code size in the new loop body, which may increase pressure on register allocation, cause register spilling, and therefore cause a loss in performance. Before applying unrolling to a loop, you must evaluate these tradeoffs. In order to check if the unroll option improves performance of a particular application, you should compile your program with the usual options, run it with a representative workload, recompile it with the UNROLL option and/or unroll pragmas, and rerun it under the same conditions to see if the UNROLL option leads to a performance improvement.

The UNROLL compiler option provides the following suboptions:

- YES** Allows the compiler to do unrolling, but compiler is not required to do so.
- NO** Means that the compiler is not permitted to unroll loops in the compilation unit, unless `unroll` or `unroll(n)` pragmas are specified for particular loops.
- AUTO** This option is the default. It tells the compiler that unrolling is allowed on any loop in the specified program, unless a loop is marked with `#pragma unroll`.

Effect on IPA Compile step

The UNROLL 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 UNROLL 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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOUPCONV	UPCONV					

CATEGORY: Programming Language Characteristics Control



The UPCONV option causes the z/OS C compiler to follow unsignedness preserving rules when doing z/OS C type conversions; that is, when widening all integral types (char, short, int, long). Use this option when compiling older z/OS 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.

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

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.

Effect on IPA Link step

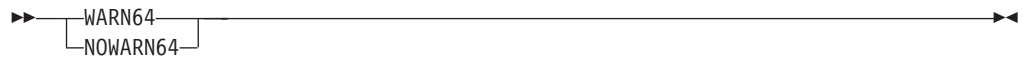
The IPA Link step accepts UPCONV option, but ignores it.

WARN64 | NOWARN64

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
NOWARN64						

CATEGORY: Debug/Diagnostic



The WARN64 option generates informational messages for situations where compiling the source code with ILP32 and LP64 may produce different behavior. This option is designed to help migrate a program to 64-bit mode. Use the FLAG(I) option to display all informational messages.

WARN64 warns you about any code fragments that have the following types of portability errors:

- A constant that selected an unsigned long int data type in 31-bit mode may fit within a long int data type in 64-bit mode
- A constant larger than UINT_MAX, but smaller than ULONGLONG_MAX will overflow in 31-bit mode, but will be acceptable in an unsigned long or signed long in 64-bit mode

It also warns you about the following types of possible portability errors:

- Loss of digits when you assign a long type to an int type
- Change in the result when you assign an int to a long type
- Loss of high-order bytes of a pointer when a pointer type is assigned to an int type
- Incorrect pointer when an int type is assigned to a pointer type
- Change of a constant value when the constant is assigned to a long type

Effect on IPA Compile step

The WARN64 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 WARN64 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)	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 WSIZE0F 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 WSIZE0F 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 WSIZE0F option to return the size of widened type for function return types.

The WSIZE0F 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 *z/OS C/C++ Language Reference*.

Effect on IPA Compile step

The WSIZE0F option has the same effect on the IPA Compile step that it does on a regular compilation.

Effect on IPA Link step

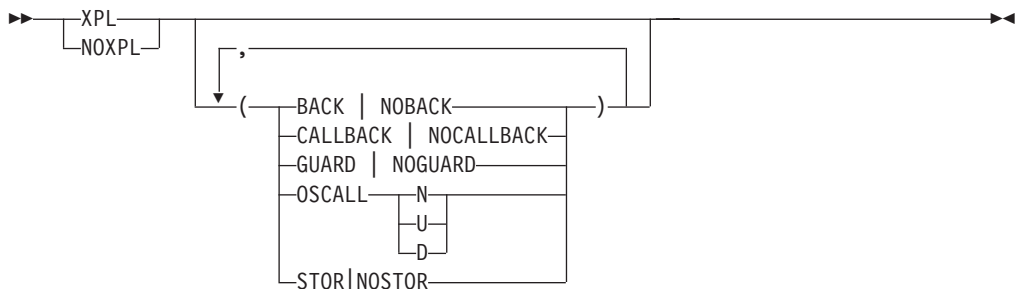
The IPA Link step accepts the WSIZE0F option, but ignores it.

XPLINK | NOXPLINK

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

z/OS C/C++ Compiler (Batch and TSO Environments)	Option Default					
	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 a warning message and promotes the option to GOFF.

In addition, the XPLINK option requires that the value of ARCH must be 2 or greater. The compiler issues a message if you specify ARCH(0) or ARCH(1) with XPLINK and forces ARCH to 2.

Note: When using XPLINK and source files with duplicate file names, the linker may emit an error and discard one of the code sections. In this case, turn off the CSECT option by specifying NOCSECT.

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.

CALLBACK | NOCALLBACK

DEFAULT: NOCALLBACK

XPLINK(CALLBACK) is primarily intended to enable function pointer calls across XPLINK DLLs and non-XPLINK programs. With XPLINK, function calls are supported across a DLL boundary with certain restrictions. In particular, if a function pointer is created by a non-XPLINK caller pointing to an XPLINK

function, it can be passed as an argument via an exported function into the DLL, which can then use it as callback. This is because the compiler knows about the function pointer argument and is able to insert code to fix-up the function pointer. However, non-XPLINK function pointers passed into the DLL by other means are not supported. If you specify CALLBACK, all calls via function pointers will be considered potentially incompatible, and fix-up code will be inserted by the compiler at the locations of the incompatible DLL callbacks through function pointers to assist the call. The application will be impacted by a performance penalty. In an XPLINK(NOCALLBACK) compilation, if a function pointer is declared using the `__callback` qualifier keyword, the compiler will insert fix-up code to assist the call. For more information on this keyword, see *z/OS C/C++ Language Reference*.

Note: In LP64 mode, the only linkage supported is XPLINK. Do not use XPLINK(CALLBACK) in LP64 mode.

GUARD | NOGUARD

DEFAULT: GUARD

If you specify NOGUARD, the compiler generates an explicit check for stack overflow, which enables the storage run-time option. Using NOGUARD causes a performance degradation at run time, even if you do not use the Language Environment run-time STORAGE option.

OSCALL(NOSTACK | UPSTACK | DOWNSTACK)

DEFAULT: NOSTACK

This suboption directs the compiler to use the linkage (OS_NOSTACK, OS_UPSTACK, or OS_DOWNSTACK) as specified in this suboption for any `#pragma linkage(identifier, OS)` calls in your application.

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.

You can build a non-XPLINK application by ensuring that there are no references to the C++ Standard Library header files and by using the Standard Template Library (STL) which is freely available on the web at <http://www.stlport.org/product.html>. The Standard C++ library header files do not have suffixes, so to find the references to the C++ Standard Library header files look for include statements that reference header file names without a suffix. For iostream classes, you can either statically link the USL iostream Class Library objects from the CBC.SCLBCPP or use the side-deck from the CBC.SCLBSID(IOSTREAM) for the DLL version. For all other classes, you can use the STLport package. The following URL has useful information about the STLport on z/OS:

<http://www.ibm.com/software/awdtools/c390/cmvsstlp.htm>. Once you make the necessary source code changes, you need to recompile the code with the NOXPLINK compiler option. To ensure that you are not referencing Standard C++ Library header files, you should make sure that the CBC.SCLBH.* header file data sets are concatenated in front of the following Language Environment system header file data sets: CEE.SCEEH.H, CEE.SCEEH.SYS.H, CEE.SCEEH.NET.H, CEE.SCEEH.NETINET.H, and CEE.SCEEH.ARPA.H, which should be the only Language Environment header file data sets used. To get the proper data set allocation at prelink/link time, the following c++ or cxx environment variables, if exported, should include the concatenations listed below. If they are unset, these variables take the default values which already include the required concatenations.

For static binding with USL iostream objects:

- `_CXX_PSYSLIB="{_CXX_PLIB_PREFIX}.SCEEOBJ:{_CXX_PLIB_PREFIX}.SCEECPP:{_CXX_CLIB_PREFIX}.SCLBCPP"`
- `_CXX_LSYSLIB="{_CXX_PLIB_PREFIX}.SCEELKEX:{_CXX_PLIB_PREFIX}.SCEELKED"`

For USL iostream DLL:

- `_CXX_PSYSIX="{_CXX_PLIB_PREFIX}.SCLBSID(IOSTREAM)"`
- `_CXX_PSYSLIB="{_CXX_PLIB_PREFIX}.SCEEOBJ:{_CXX_PLIB_PREFIX}.SCEECPP"`
- `_CXX_LSYSLIB="{_CXX_PLIB_PREFIX}.SCEELKEX:{_CXX_PLIB_PREFIX}.SCEELKED"`

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.

This option affects the IPA optimized object module that is generated by specifying the IPA(OBJONLY) option. The object format used to encode this object depends on the GOFF option.

The XPLINK option also affects the regular object module if you request one by specifying the IPA(OBJECT) option. The object format used to encode the regular object depends on the GOFF option.

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.

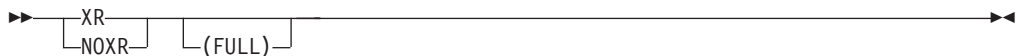
Partitions with the XPLINK option are always generated with the G0FF option.

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)	z/OS UNIX System Services Utilities					
	Regular Compile			IPA Link		
	c89	cc	c++	c89	cc	c++
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. XREF and XREF(FULL) will generate the same output.

You can specify this option using the #pragma options 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 options(XREF)
- For C++, if you specify the ATTR, XREF, IPA(ATTRIBUTE), or IPA(XREF) options

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

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 272 for information about IPA Link step listings.

Example of a C Compiler Listing

Figure 12 on page 224 shows an example of a C compiler listing.

* * * * * P R O L O G * * * * *

```

Compile Time Library . . . . . : 41060000
Command options:
Program name. . . . . : 'TSCTEST.ZOSV1R6.SCCNSAM(CCNUAAM) '
Compiler options. . . . . : *NOGONUMBER *NOALIAS *NORENT *TERMINAL *NOUPCONV *SOURCE *LIST
                          : *XREF *AGGR *NOPPONLY *NOEXPMAC *NOSHOWINC *NOOFFSET *MEMORY *NOSSCOMM
                          : *NOLONGNAME *START *EXECOPS *ARGPARSE *NOEXPORTALL *NODLL (NOCALLBACKANY)
                          : *NOLIBANSI *NOWSIZEOF *REDIR *ANSIALIAS *DIGRAPH *NOROCONST *ROSTRING
                          : *TUNE (5) *ARCH (5) *SPILL (128) *MAXMEM (2097152) *NOCOMPACT
                          : *TARGET (LE, CURRENT) *FLAG (I) *NOTEST (SYM, BLOCK, LINE, PATH, HOOK) *NOOPTIMIZE
                          : *INLINE (AUTO, REPORT, 100, 1000) *NESTINC (255) *BITFIELD (UNSIGNED)
                          : *NOCHECKOUT (NOPPTRACE, PPCHECK, GOTO, ACCURACY, PARM, NOENUM,
                          : NOEXTERN, TRUNC, INIT, NOPORT, GENERAL, CAST)
                          : *FLOAT (HEX, FOLD, AFP) *STRICT *NOIGNERRNO *NOINITAUTO
                          : *NOCOMPRESS *NOSTRICT_INDUCTION *AGGRCOPY (NOOVERLAP) *CHARS (UNSIGNED)
                          : *NOCSECT
                          : *NOEVENTS
                          : *OBJECT
                          : *NOOPTFILE
                          : *NOSERVICE
                          : *NOOE
                          : *NOIPA
                          : *SEARCH (// 'TSCTEST.CEEZ160.SCEEH.+ ' )
                          : *NOLSEARCH
                          : *NOLOCALE *HALT (16) *PLIST (HOST)
                          : *NOCONVLIT
                          : *NOASCII
                          : *NOGOFF *ILP32 *NOWARN64
                          : *NOXPLINK (NOBACKCHAIN, NOSTOREARGS, NOCALLBACK, GUARD, OSCALL (NOSTACK))
                          : *ENUMSIZE (SMALL)
                          : *NOHALTONMSG
                          : *NOSUPPRESS
                          : *NODEBUG
                          : *NOSQL
                          : *UNROLL (AUTO)
Version Macros. . . . . : __COMPILER_VER__=0x41060000 __LIBREL__=0x41060000 __TARGET_LIB__=0x41060000
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 12. Example of a C listing (Part 1 of 31)

```

          * * * * * S O U R C E * * * * *

LINE  STMT
1      | #include <stdio.h>
2      |
3      | #include "ccnuaan.h"
4      |
5      | void convert(double);
6      |
7      | int main(int argc, char **argv)
8      | {
9      |     double c_temp;
10     |
11     | 1     if (argc == 1) { /* get Celsius value from stdin */
12     |     int ch;
13     |
14     | 2     printf("Enter Celsius temperature: \n");
15     |
16     | 3     if (scanf("%f", &c_temp) != 1) {
17     | 4     printf("You must enter a valid temperature\n");
18     |     }
19     |     else {
20     | 5     convert(c_temp);
21     |     }
22     | }
23     | else { /* convert the command-line arguments to Fahrenheit */
24     |     int i;
25     |
26     | 6     for (i = 1; i < argc; ++i) {
27     | 7     if (sscanf(argv[i], "%f", &c_temp) != 1)
28     | 8     printf("%s is not a valid temperature\n",argv[i]);
29     |     else
30     | 9     convert(c_temp);
31     |     }
32     | }
33     | 10    return 0;
34     | }
35     |
36     | void convert(double c_temp) {
37     | 11    double f_temp = (c_temp * CONV + OFFSET);
38     | 12    printf("%5.2f Celsius is %5.2f Fahrenheit\n",c_temp, f_temp);
39     | }

          * * * * * E N D   O F   S O U R C E * * * * *

```

Figure 12. Example of a C listing (Part 2 of 31)

***** INCLUDES *****

INCLUDE FILES --- FILE# NAME

1 TSCTEST.CEEZ160.SCEEH.H(STDIO)
 2 TSCTEST.CEEZ160.SCEEH.H(FEATURES)
 3 TSCTEST.CEEZ160.SCEEH.SYS.H(TYPES)
 4 TSCTEST.ZOSV1R6.SCCNSAM(CCNAAAM)

***** END OF INCLUDES *****

***** CROSS REFERENCE LISTING *****

IDENTIFIER	DEFINITION	ATTRIBUTES <SEQNBR>-<FILE NO>:<FILE LINE NO>
__valist	1-1:133	Class = typedef, Length = 8 Type = array[2] of pointer to unsigned char 1-1:136, 1-1:443, 1-1:444, 1-1:445
__abend	1-1:797	Type = struct with no tag in union at offset 0
__alloc	1-1:807	Type = struct with no tag in union at offset 0
__amrc_ptr	1-1:835	Class = typedef, Length = 4 Type = pointer to struct __amrc_type
__amrc_type	1-1:831	Class = typedef, Length = 224 Type = struct __amrc_type 1-1:835
__amrc_type	1-1:779	Class = struct tag
__amrc2_ptr	1-1:849	Class = typedef, Length = 4 Type = pointer to struct __amrc2_type
__amrc2_type	1-1:845	Class = typedef, Length = 32 Type = struct __amrc2_type 1-1:849
__amrc2_type	1-1:840	Class = struct tag
__blksize	1-1:677	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	1-1:808	Type = union with no tag in struct __amrc_type 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	1-1:192	Class = typedef, Length = 4 Type = pointer to const unsigned short
__device	1-1:676	Type = enum with no tag in struct __fileData at offset 4
__device_t	1-1:622	Class = typedef, Length = 1 Type = enum with no tag 1-1:676
__disk	1-1:607	Class = enumeration constant: 0, Length = 4 Type = int
__dsname	1-1:682	Type = pointer to unsigned char in struct __fileData at offset 28

Figure 12. Example of a C listing (Part 3 of 31)

```

***** CROSS REFERENCE LISTING *****

IDENTIFIER      DEFINITION      ATTRIBUTES
__dsorgConcat   1-1:653         <SEQNBR>-<FILE NO>-<FILE LINE NO>
                  Type = unsigned int:1 in struct __fileData at offset 1(3)
__dsorgHiper    1-1:655         Type = unsigned int:1 in struct __fileData at offset 1(5)
__dsorgHFS      1-1:660         Type = unsigned int:1 in struct __fileData at offset 2(0)
__dsorgMem      1-1:654         Type = unsigned int:1 in struct __fileData at offset 1(4)
__dsorgPDSdir   1-1:651         Type = unsigned int:1 in struct __fileData at offset 1(1)
__dsorgPDSmem   1-1:650         Type = unsigned int:1 in struct __fileData at offset 1(0)
__dsorgPDSE     1-1:667         Type = unsigned int:1 in struct __fileData at offset 2(7)
__dsorgPO       1-1:649         Type = unsigned int:1 in struct __fileData at offset 0(7)
__dsorgPS       1-1:652         Type = unsigned int:1 in struct __fileData at offset 1(2)
__dsorgTemp     1-1:656         Type = unsigned int:1 in struct __fileData at offset 1(6)
__dsorgVSAM     1-1:657         Type = unsigned int:1 in struct __fileData at offset 1(7)
__dummy        1-1:612         Class = enumeration constant: 6, Length = 4
                  Type = int
__error         1-1:791         Type = int in union at offset 0
__error2        1-1:841         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          1-1:802         Type = unsigned char in struct at offset 3
__fdbk_fill     1-1:799         Type = unsigned char in struct at offset 0
__feedback      1-1:803         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       1-1:843         Type = pointer to struct __ffile in struct __amrc2type at offset 4
__fileData      1-1:641         Class = struct tag
                  1-1:686
__fill          1-1:766         Type = unsigned int in struct at offset 0

```

Figure 12. Example of a C listing (Part 4 of 31)

***** CROSS REFERENCE LISTING *****

IDENTIFIER	DEFINITION	ATTRIBUTES
		<SEQNBR>-<FILE NO>-<FILE LINE NO>
__filler1	1-1:594	Type = unsigned char in struct __S99emparms at offset 3
__fill2	1-1:822	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	1-1:801	Type = unsigned char in struct at offset 2
__hfs	1-1:619	Class = enumeration constant: 9, Length = 4 Type = int
__hiperspace	1-1:620	Class = enumeration constant: 10, Length = 4 Type = int
__last_op	1-1:815	Type = unsigned int in struct __amrctype at offset 8
__len	1-1:818	Type = unsigned int in struct at offset 4
__len_fill	1-1:817	Type = unsigned int in struct at offset 0
__maxreclen	1-1:678	Type = unsigned long in struct __fileData at offset 12
__memory	1-1:618	Class = enumeration constant: 8, Length = 4 Type = int
__modeflag	1-1:666	Type = unsigned int:4 in struct __fileData at offset 2(3)
__msg	1-1:825	Type = struct with no tag in struct __amrctype at offset 12
__msgfile	1-1:615	Class = enumeration constant: 7, Length = 4 Type = int
__openmode	1-1:665	Type = unsigned int:2 in struct __fileData at offset 2(1)
__other	1-1:621	Class = enumeration constant: 255, Length = 4 Type = int
__parmr0	1-1:820	Type = unsigned int in struct at offset 128
__parmr1	1-1:821	Type = unsigned int in struct at offset 132
__printer	1-1:609	Class = enumeration constant: 2, Length = 4 Type = int
__rc	1-1:796	Type = unsigned short in struct at offset 2
__rc	1-1:800	Type = unsigned char in struct at offset 1

Figure 12. Example of a C listing (Part 5 of 31)

```

***** CROSS REFERENCE LISTING *****
IDENTIFIER      DEFINITION      ATTRIBUTES
<SEQNBR>-<FILE NO>:<FILE LINE NO>
__recfmASA      1-1:647        Type = unsigned int:1 in struct __fileData at offset 0(5)
__recfmBlk     1-1:646        Type = unsigned int:1 in struct __fileData at offset 0(4)
__recfmF       1-1:642        Type = unsigned int:1 in struct __fileData at offset 0(0)
__recfmM       1-1:648        Type = unsigned int:1 in struct __fileData at offset 0(6)
__recfmS       1-1:645        Type = unsigned int:1 in struct __fileData at offset 0(3)
__recfmU       1-1:644        Type = unsigned int:1 in struct __fileData at offset 0(2)
__recfmV       1-1:643        Type = unsigned int:1 in struct __fileData at offset 0(1)
__recnum       1-1:767        Type = unsigned int in struct at offset 4
__reserved     1-1:577        Type = unsigned char in struct __S99rbx at offset 16
__reserved     1-1:844        Type = array[6] of int in struct __amrc2type at offset 8
__reserve2     1-1:671        Type = unsigned int:5 in struct __fileData at offset 3(3)
__reserve4     1-1:683        Type = unsigned int in struct __fileData at offset 32
__reserv1      1-1:599        Type = int in struct __S99emparms at offset 20
__reserv2      1-1:585        Type = int in struct __S99rbx at offset 32
__reserv2      1-1:600        Type = int in struct __S99emparms at offset 24
__rp1fdbwd    1-1:828        Type = array[4] of unsigned char in struct __amrctype at offset 220
__rrds_key_type 1-1:772        Class = typedef, Length = 8
                Type = struct with no tag
__str          1-1:819        Type = array[120] of unsigned char in struct at offset 8
__str2         1-1:823        Type = array[64] of unsigned char in struct at offset 144
__svc99_error  1-1:806        Type = unsigned short in struct at offset 2
__svc99_info   1-1:805        Type = unsigned short in struct at offset 0
__syscode     1-1:795        Type = unsigned short in struct at offset 0
__tape        1-1:610        Class = enumeration constant: 3, Length = 4
                Type = int
__tdq         1-1:611        Class = enumeration constant: 5, Length = 4
                Type = int
__terminal     1-1:608        Class = enumeration constant: 1, Length = 4

```

Figure 12. 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>:<FILE LINE NO> Type = int
__vsamkeylen	1-1:680	Type = unsigned long in struct __fileData at offset 20
__vsamtype	1-1:679	Type = unsigned short in struct __fileData at offset 16
__vsamRKP	1-1:681	Type = unsigned long in struct __fileData at offset 24
__vsamRLS	1-1:670	Type = unsigned int:3 in struct __fileData at offset 3(0)
__EMBUFP	1-1:598	Type = pointer to void in struct __S99emparms at offset 16
__EMCPPLP	1-1:597	Type = pointer to void in struct __S99emparms at offset 12
__EMFUNCT	1-1:591	Type = unsigned char in struct __S99emparms at offset 0
__EMIDNUM	1-1:592	Type = unsigned char in struct __S99emparms at offset 1
__EMNMSGBK	1-1:593	Type = unsigned char in struct __S99emparms at offset 2
__EMRETCOD	1-1:596	Type = int in struct __S99emparms at offset 8
__EMS99RBP	1-1:595	Type = pointer to void in struct __S99emparms at offset 4
__FILEP	1-1:84	Class = typedef, Length = 4 Type = pointer to struct __ffile
__RBA	1-1:809	Type = unsigned int in struct __amrctype at offset 4
__S99emparms	1-1:590	Class = struct tag 1-1:603
__S99emparms_t	1-1:603	Class = typedef, Length = 28 Type = struct __S99emparms
__S99parms	1-1:563	Class = typedef, Length = 20 Type = struct __S99struc 1-1:735
__S99rbx	1-1:567	Class = struct tag 1-1:588
__S99rbx_t	1-1:588	Class = typedef, Length = 36 Type = struct __S99rbx
__S99struc	1-1:546	Class = struct tag 1-1:563
__S99ECPLP	1-1:576	Type = pointer to void in struct __S99rbx at offset 12
__S99EERR	1-1:583	Type = unsigned short in struct __S99rbx at offset 28
__S99EID	1-1:569	Type = array[6] of unsigned char in struct __S99rbx at offset 0

Figure 12. 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>:<FILE LINE NO>
__S99EINFO      1-1:584          Type = unsigned short in struct __S99rbx at offset 30
__S99EKEY       1-1:573          Type = unsigned char in struct __S99rbx at offset 9
__S99EMGSV      1-1:574          Type = unsigned char in struct __S99rbx at offset 10
__S99EMSGP      1-1:582          Type = pointer to void in struct __S99rbx at offset 24
__S99ENMSG      1-1:575          Type = unsigned char in struct __S99rbx at offset 11
__S99EOPTS      1-1:571          Type = unsigned char in struct __S99rbx at offset 7
__S99ERCF       1-1:580          Type = unsigned char in struct __S99rbx at offset 19
__S99ERCO       1-1:579          Type = unsigned char in struct __S99rbx at offset 18
__S99ERES       1-1:578          Type = unsigned char in struct __S99rbx at offset 17
__S99ERROR      1-1:552          Type = unsigned short in struct __S99struc at offset 4
__S99ESUBP      1-1:572          Type = unsigned char in struct __S99rbx at offset 8
__S99EVER       1-1:570          Type = unsigned char in struct __S99rbx at offset 6
__S99EWRC       1-1:581          Type = int in struct __S99rbx at offset 20
__S99FLAG1      1-1:550          Type = unsigned short in struct __S99struc at offset 2
__S99FLAG2      1-1:558          Type = unsigned int in struct __S99struc at offset 16
__S99INFO       1-1:553          Type = unsigned short in struct __S99struc at offset 6
__S99RBLN       1-1:548          Type = unsigned char in struct __S99struc at offset 0
__S99S99X       1-1:556          Type = pointer to void in struct __S99struc at offset 12
__S99XTTPP      1-1:554          Type = pointer to void in struct __S99struc at offset 8
__S99VERB       1-1:549          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 12. 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
_PUTCFUNC	1-1:67	<SEQNBR>-<FILE NO>-<FILE LINE NO> 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	36-0:36	Class = parameter, Length = 8 Type = double in function convert 37-0:37, 38-0:38
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 1-1:397
clrmemf		Class = extern Type = function returning int 1-1:739
convert	36-0:36	Class = extern Type = function returning void 5-0:5, 20-0:20, 30-0:30
f_temp	37-0:37	Class = auto, Length = 8 Type = double in function convert 37-0:37, 38-0:38
fclose		Class = extern Type = function returning int 1-1:398
fdelrec		Class = extern Type = function returning int 1-1:737
feof		Class = extern Type = function returning int 1-1:399
ferror		Class = extern Type = function returning int

Figure 12. 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>:<FILE LINE NO>
1-1:400

fflush          Class = extern
                Type = function returning int
                1-1:401

fgetc           Class = extern
                Type = function returning int
                1-1:402

fgetpos         Class = extern
                Type = function returning int
                1-1:403

fgets           Class = extern
                Type = function returning pointer to unsigned char
                1-1:404

fldata          Class = extern
                Type = function returning int
                1-1:740

fldata_t        1-1:686      Class = typedef, Length = 36
                Type = struct __fileData
                1-1:740

flocate         Class = extern
                Type = function returning int
                1-1:736

fopen           Class = extern
                Type = function returning pointer to struct __ffile
                1-1:405

fpos_t          1-1:98      Class = typedef, Length = 32
                Type = struct __fpos_t
                1-1:403, 1-1:416

fprintf         Class = extern
                Type = function returning int
                1-1:407

fputc           Class = extern
                Type = function returning int
                1-1:408

fputs           Class = extern
                Type = function returning int
                1-1:409

fread           Class = extern
                Type = function returning unsigned int
                1-1:410

```

Figure 12. 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>:<FILE LINE NO>

freopen          Class = extern
                 Type = function returning pointer to struct __ffile
                 1-1:412

fscanf          Class = extern
                 Type = function returning int
                 1-1:414

fseek           Class = extern
                 Type = function returning int
                 1-1:415

fsetpos         Class = extern
                 Type = function returning int
                 1-1:416

ftell           Class = extern
                 Type = function returning long
                 1-1:417

fupdate         Class = extern
                 Type = function returning unsigned int
                 1-1:738

fwrite          Class = extern
                 Type = function returning unsigned int
                 1-1:418

getc            Class = extern
                 Type = function returning int
                 1-1:420

getchar         Class = extern
                 Type = function returning int
                 1-1:421

gets            Class = extern
                 Type = function returning pointer to unsigned char
                 1-1:422

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
                 1-1:423

printf          Class = extern

```

Figure 12. Example of a C listing (Part 11 of 31)

```

***** CROSS REFERENCE LISTING *****

IDENTIFIER      DEFINITION      ATTRIBUTES
<SEQNBR>-<FILE NO>:<FILE LINE NO>
Type = function returning int
1-1:424, 14-0:14, 17-0:17, 28-0:28, 38-0:38

putc           Class = extern
Type = function returning int
1-1:425

putchar        Class = extern
Type = function returning int
1-1:426

puts           Class = extern
Type = function returning int
1-1:427

remove         Class = extern
Type = function returning int
1-1:428

rename         Class = extern
Type = function returning int
1-1:429

rewind         Class = extern
Type = function returning void
1-1:430

scanf          Class = extern
Type = function returning int
1-1:431, 16-0:16

setbuf         Class = extern
Type = function returning void
1-1:432

setvbuf        Class = extern
Type = function returning int
1-1:433

size_t         1-1:50      Class = typedef, Length = 4
Type = unsigned int
1-1:410, 1-1:410, 1-1:411, 1-1:418, 1-1:418, 1-1:418, 1-1:434, 1-1:736, 1-1:738, 1-1:738

sprintf        Class = extern
Type = function returning int
1-1:436

sscanf         Class = extern
Type = function returning int
1-1:438, 27-0:27

ssize_t        1-1:59      Class = typedef, Length = 4
Type = int

```

Figure 12. Example of a C listing (Part 12 of 31)

```

***** CROSS REFERENCE LISTING *****

IDENTIFIER      DEFINITION      ATTRIBUTES
<SEQNBR>-<FILE NO>-<FILE LINE NO>

svc99           Class = extern
                Type = function returning int
                1-1:735

tmpfile        Class = extern
                Type = function returning pointer to struct __ffile
                1-1:440

tmpnam         Class = extern
                Type = function returning pointer to unsigned char
                1-1:441

ungetc         Class = extern
                Type = function returning int
                1-1:442

va_list        1-1:136  Class = typedef, Length = 8
                Type = array[2] of pointer to unsigned char

vfprintf       Class = extern
                Type = function returning int
                1-1:443

vprintf        Class = extern
                Type = function returning int
                1-1:444

vsprintf       Class = extern
                Type = function returning int
                1-1:445

FILE           1-1:90   Class = typedef, Length = 4
                Type = struct __ffile
                1-1:397, 1-1:398, 1-1:399, 1-1:400, 1-1:401, 1-1:402, 1-1:403, 1-1:404, 1-1:405, 1-1:407,
                1-1:408, 1-1:409, 1-1:411, 1-1:412, 1-1:413, 1-1:414, 1-1:415, 1-1:416, 1-1:417, 1-1:419,
                1-1:420, 1-1:425, 1-1:430, 1-1:432, 1-1:433, 1-1:440, 1-1:442, 1-1:443, 1-1:736, 1-1:737,
                1-1:738, 1-1:740, 1-1:843

***** END OF CROSS REFERENCE LISTING *****

```

Figure 12. Example of a C listing (Part 13 of 31)

***** STRUCTURE MAPS *****

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 12. Example of a C listing (Part 14 of 31)

```

***** STRUCTURE MAPS *****
| 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 __amrc_type Total size: 224 bytes
.....
__amrc_type
*_amrc_ptr
=====
Offset Length Member Name
Bytes(Bits) Bytes(Bits)
-----
0 4 __code
=====

```

Figure 12. Example of a C listing (Part 15 of 31)


```

***** STRUCTURE MAPS *****

```

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 Bytes(Bits)	Length Bytes(Bits)	Member Name
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 12. Example of a C listing (Part 16 of 31)

```

***** STRUCTURE MAPS *****
| 220 | 4 | __rp1fdbwd[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 12. Example of a C listing (Part 17 of 31)

***** STRUCTURE MAPS *****

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
Offset Bytes(Bits)	Length Bytes(Bits)	Member Name	
fldata_t			
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 12. 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 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
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 12. Example of a C listing (Part 19 of 31)

```

***** STRUCTURE MAPS *****
| 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 12. Example of a C listing (Part 20 of 31)

```

***** STRUCTURE MAPS *****

```

3	1	__filler1
4	4	__EMS99RBP
8	4	__EMRETCOD
12	4	__EMCPPLP
16	4	__EMBUF
20	4	__reserv1
24	4	__reserv2

```

Aggregate map for: struct __S99rbx Total size: 36 bytes
__S99rbx_t

```

Offset Bytes(Bits)	Length Bytes(Bits)	Member Name
0	6	__S99EID[6]
6	1	__S99EVER
7	1	__S99EOPTS
8	1	__S99ESUBP
9	1	__S99EKEY
10	1	__S99EMGSV
11	1	__S99ENMSG
12	4	__S99ECPLP
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 Bytes(Bits)	Length Bytes(Bits)	Member Name
0	6	__S99EID[6]
6	1	__S99EVER
7	1	__S99EOPTS
8	1	__S99ESUBP
9	1	__S99EKEY
10	1	__S99EMGSV
11	1	__S99ENMSG
12	4	__S99ECPLP
16	1	__reserved
17	1	__S99ERES
18	1	__S99ERCO
19	1	__S99ERCF

Figure 12. Example of a C listing (Part 21 of 31)

```

***** STRUCTURE MAPS *****
| 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      | |__S99XTTP
| 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      | |__S99XTTP
| 12      | 4      | |__S99S99X
| 16      | 4      | |__S99FLAG2
=====
***** END OF STRUCTURE MAPS *****

```

Figure 12. Example of a C listing (Part 22 of 31)

***** MESSAGE SUMMARY *****

Total Informational(00) Warning(10) Error(30) Severe Error(40)

0 0 0 0 0

***** END OF MESSAGE SUMMARY *****

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	15	0/0	2/2	convert
A	T,N	E	110 (72)	2/2	0/0	main

Mode = AUTO Inlining Threshold = 100 Expansion Limit = 1000

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

```

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 F0F3          =C'2003'          Compiled Year
000004 F1F1 F2F1          =C'1121'          Compiled Date MMDD
000008 F1F8 F0F2 F2F3      =C'180223'        Compiled Time HHMMSS
00000E F0F1 F0F6 F0F0      =C'010600'        Compiler Version
Timestamp and Version End
    
```

Figure 12. Example of a C listing (Part 23 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
000001 | * #include <stdio.h>
000002 | *
000003 | * #include "ccnuaan.h"
000004 | *
000005 | * void convert(double);
000006 | *
000007 | * int main(int argc, char **argv)
000007 | main      DS      0D
000007 |          B      34(,r15)
000018 |          CEE eyecatcher
00001C |          DSA size
000020 |          =A(PPA1-main)
000024 |          B      1(,r15)
000028 |          L      r15,796(,r12)
00002C |          LR     r4,r14
000030 |          BALR  r14,r15
000032 |          =F'0'
000034 |          BR     r3
000038 |          STM   r14,r5,12(r13)
00003A |          L      r14,76(,r13)
00003E |          LA    r0,288(,r14)
000042 |          CL   r0,788(,r12)
000046 |          LA    r3,58(,r15)
00004A |          BH   20(,r15)
00004E |          L      r15,640(,r12)
000052 |          STM   r15,r0,72(r14)
000056 |          MVI  0(r14),16
00005A |          ST   r13,4(,r14)
000062 |          LR   r13,r14
000064 |          End of Prolog

000064 |          LARL  r5,F'286'
00006A |          ST   r1,#SR_PARAM_1(,r13,208)
000008 | * {
000009 | *   double c_temp;
000010 | *
000011 | *   if (argc == 1) { /* get Celsius value from stdin */
000011 |     L      r1,#SR_PARAM_1(,r13,208)
000011 |     L      r0,argc(,r1,0)
000011 |     CHI   r0,H'1'
000011 |     BNE  @1L1
000012 | *     int ch;
000013 | *
000014 | *     printf("Enter Celsius temperature: \n");
000014 |     L      r15,=V(PRINTF)(,r3,406)
000014 |     LR     r0,r5
000014 |     LA    r1,#MX_TEMP1(,r13,152)
000014 |     ST   r0,#MX_TEMP1(,r13,152)
000014 |     BALR  r14,r15
000015 | *
000016 | *     if (scanf("%f", &c_temp) != 1) {
000016 |     LA    r0,c_temp(,r13,176)
000016 |     L      r15,=V(SCANF)(,r3,410)
000016 |     LA    r2,+CONSTANT_AREA(,r5,29)
000016 |     LA    r1,#MX_TEMP1(,r13,152)
000016 |     ST   r2,#MX_TEMP1(,r13,152)

```

Figure 12. Example of a C listing (Part 24 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
0000A2 5000 D09C      000016      ST  r0,#MX_TEMP1(,r13,156)
0000A6 05EF          000016      BALR r14,r15
0000A8 180F          000016      LR   r0,r15
0000AA A70E 0001      000016      CHI  r0,H'1'
0000AE 4780 3076      000016      BE  @1L2
000017      *      printf("You must enter a valid temperature\n");
0000B2 58F0 3196      000017      L   r15,=V(PRINTF)(,r3,406)
0000B6 4100 5020      000017      LA  r0,+CONSTANT_AREA(,r5,32)
0000BA 4110 D098      000017      LA  r1,#MX_TEMP1(,r13,152)
0000BE 5000 D098      000017      ST  r0,#MX_TEMP1(,r13,152)
0000C2 05EF          000017      BALR r14,r15
0000C4 47F0 30AC      000017      B   @1L3
0000C8          000017      @1L2 DS  0H
000018      *      }
000019      *      else {
000020      *      convert(c_temp);
000020      LD  f0,c_temp(,r13,176)
0000CC 6000 D0C0      000020      STD f0,c_temp:convert(,r13,192)
0000D0 6820 5048      000037      +   LD  f2,+CONSTANT_AREA(,r5,72)
0000D4 2C02          000037      +   MDR f0,f2
0000D6 6820 5050      000037      +   LD  f2,+CONSTANT_AREA(,r5,80)
0000DA 2A02          000037      +   ADR f0,f2
0000DC 6000 D0C8      000037      +   STD f0,f_temp:convert(,r13,200)
0000E0 6820 D0C0      000038      +   LD  f2,c_temp:convert(,r13,192)
0000E4 6020 D09C      000038      +   STD f2,#MX_TEMP1(,r13,156)
0000E8 6000 D0A4      000038      +   STD f0,#MX_TEMP1(,r13,164)
0000EC 58F0 3196      000038      +   L   r15,=V(PRINTF)(,r3,406)
0000F0 4100 5058      000038      +   LA  r0,+CONSTANT_AREA(,r5,88)
0000F4 4110 D098      000038      +   LA  r1,#MX_TEMP1(,r13,152)
0000F8 5000 D098      000038      +   ST  r0,#MX_TEMP1(,r13,152)
0000FC 05EF          000038      +   BALR r14,r15
0000FE          000039      +@1L10 DS  0H
0000FE          000039      +@1L3  DS  0H
0000FE 47F0 3180      000039      +   B   @1L4
000102          000039      +@1L1  DS  0H
000021      *      }
000022      *      }
000023      *      else { /* convert the command-line arguments to Fahrenheit */
000024      *      int i;
000025      *
000026      *      for (i = 1; i < argc; ++i) {
000102 4100 0001      000026      LA  r0,1
000106 5000 D0B8      000026      ST  r0,i(,r13,184)
00010A 5810 D0D0      000026      L   r1,#SR_PARM_1(,r13,208)
00010E 5810 1000      000026      L   r1,argc(,r1,0)
000112 1901          000026      CR  r0,r1
000114 47B0 3180      000026      BNL @1L6
000118          000026      @1L5 DS  0H
000027      *      if (sscanf(argv[i], "%f", &c_temp) != 1)
000118 5810 D0D0      000027      L   r1,#SR_PARM_1(,r13,208)
00011C 5810 1004      000027      L   r1,argv(,r1,4)
000120 5820 D0B8      000027      L   r2,i(,r13,184)
000124 8920 0002      000027      SLL r2,2
000128 5822 1000      000027      L   r2,(*)uchar*(r2,r1,0)
00012C 4100 D0B0      000027      LA  r0,c_temp(,r13,176)
000130 58F0 319E      000027      L   r15,=V(SSCANF)(,r3,414)

```

Figure 12. 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
000134 4110 D098      000027      LA  r1,#MX_TEMP1(,r13,152)
000138 5020 D098      000027      ST  r2,#MX_TEMP1(,r13,152)
00013C 4120 501D      000027      LA  r2,+CONSTANT_AREA(,r5,29)
000140 5020 D09C      000027      ST  r2,#MX_TEMP1(,r13,156)
000144 5000 D0A0      000027      ST  r0,#MX_TEMP1(,r13,160)
000148 05EF          000027      BALR r14,r15
00014A 180F          000027      LR  r0,r15
00014C A70E 0001      000027      CHI r0,H'1'
000150 4780 3130      000027      BE  @1L7
                                000028      *
                                000028      printf("%s is not a valid temperature\n",argv[i]);
000154 5810 D0D0      000028      L   r1,#SR_PARM_1(,r13,208)
000158 5810 1004      000028      L   r1,argv(,r1,4)
00015C 5820 D0B8      000028      L   r2,i(,r13,184)
000160 8920 0002      000028      SLL r2,2
000164 5802 1000      000028      L   r0,(*)uchar*(r2,r1,0)
000168 58F0 3196      000028      L   r15,=V(PRINTF)(,r3,406)
00016C 4120 507B      000028      LA  r2,+CONSTANT_AREA(,r5,123)
000170 4110 D098      000028      LA  r1,#MX_TEMP1(,r13,152)
000174 5020 D098      000028      ST  r2,#MX_TEMP1(,r13,152)
000178 5000 D09C      000028      ST  r0,#MX_TEMP1(,r13,156)
00017C 05EF          000028      BALR r14,r15
00017E 47F0 3166      000028      B   @1L8
000182          000028      @1L7 DS  0H
                                000029      *
                                000029      else
000182 6800 D0B0      000030      *
                                000030      convert(c_temp);
000186 6000 D0C0      000030      LD  f0,c_temp(,r13,176)
00018A 6820 5048      000037      +   STD f0,c_temp:convert(,r13,192)
00018E 2C02          000037      +   LD  f2,+CONSTANT_AREA(,r5,72)
000190 6820 5050      000037      +   MDR f0,f2
000194 2A02          000037      +   LD  f2,+CONSTANT_AREA(,r5,80)
000196 6000 D0C8      000037      +   ADR f0,f2
00019A 6820 D0C0      000037      +   STD f0,f_temp:convert(,r13,200)
00019E 6020 D09C      000038      +   LD  f2,c_temp:convert(,r13,192)
0001A2 6000 D0A4      000038      +   STD f2,#MX_TEMP1(,r13,156)
0001A6 58F0 3196      000038      +   STD f0,#MX_TEMP1(,r13,164)
0001AA 4100 5058      000038      +   L   r15,=V(PRINTF)(,r3,406)
0001AE 4110 D098      000038      +   LA  r0,+CONSTANT_AREA(,r5,88)
0001B2 5000 D098      000038      +   LA  r1,#MX_TEMP1(,r13,152)
0001B6 05EF          000038      +   ST  r0,#MX_TEMP1(,r13,152)
0001B8          000039      +   BALR r14,r15
0001B8          000039      +@1L11 DS  0H
0001B8          000039      +@1L8 DS  0H
0001B8 5800 D0B8      000039      +   L   r0,i(,r13,184)
0001BC A70A 0001      000039      +   AHI r0,H'1'
0001C0 5000 D0B8      000039      +   ST  r0,i(,r13,184)
0001C4 5810 D0D0      000039      +   L   r1,#SR_PARM_1(,r13,208)
0001C8 5810 1000      000039      +   L   r1,argc(,r1,0)
0001CC 1901          000039      +   CR  r0,r1
0001CE 4740 30C6      000039      +   BL  @1L5
0001D2          000039      +@1L9 DS  0H
0001D2          000039      +@1L6 DS  0H
0001D2          000039      +@1L4 DS  0H
                                000031      *
                                000031      }
                                000032      *
                                000032      }
                                000033      *
                                000033      return 0;
0001D2 41F0 0000      000033      LA  r15,0

```

Figure 12. Example of a C listing (Part 26 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
0001D6                000034 |      *  }
0001D6                000034 |      @1L13  DS   0H

0001D6                Start of Epilog
0001D6 180D            000034 |      LR   r0,r13
0001D8 58D0 D004      000034 |      L    r13,4(,r13)
0001DC 58E0 D00C      000034 |      L    r14,12(,r13)
0001E0 9825 D01C      000034 |      LM   r2,r5,28(r13)
0001E4 051E            000034 |      BALR r1,r14
0001E6 0707            000034 |      NOPR 7

0001E8                Start of Literals
0001E8 00000000        =V(PRINTF)
0001EC 00000000        =V(SCANF)
0001F0 00000000        =V(SSCANF)
0001F4                End of Literals

*** General purpose registers used: 1111110000001111
*** Floating point registers used: 1111111000000000
*** Size of register spill area: 128(max) 0(used)
*** Size of dynamic storage: 288
*** Size of executable code: 464

```

Figure 12. 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
		000001		* #include <stdio.h>		
		000002		*		
		000003		* #include "ccnuaan.h"		
		000004		*		
		000005		* void convert(double);		
		000006		*		
		000007		* int main(int argc, char **argv)		
		000008		* {		
		000009		* double c_temp;		
		000010		*		
		000011		* if (argc == 1) { /* get Celsius value from stdin */		
		000012		* int ch;		
		000013		*		
		000014		* printf("Enter Celsius temperature: \n");		
		000015		*		
		000016		* if (scanf("%f", &c_temp) != 1) {		
		000017		* printf("You must enter a valid temperature\n");		
		000018		* }		
		000019		* else {		
		000020		* convert(c_temp);		
		000021		* }		
		000022		* }		
		000023		* else { /* convert the command-line arguments to Fahrenheit */		
		000024		* int i;		
		000025		*		
		000026		* for (i = 1; i < argc; ++i) {		
		000027		* if (sscanf(argv[i], "%f", &c_temp) != 1)		
		000028		* printf("%s is not a valid temperature\n",argv[i]);		
		000029		* else		
		000030		* convert(c_temp);		
		000031		* }		
		000032		* }		
		000033		* return 0;		
		000034		* }		
		000035		*		
		000036		* void convert(double c_temp) {		
0001F8		000036		convert	DS	0D
0001F8	47F0 F022	000036			B	34(,r15)
0001FC	01C3C5C5					CEE eyecatcher
000200	00000100					DSA size
000204	00000188					=A(PPA1-convert)
000208	47F0 F001	000036			B	1(,r15)
00020C	58F0 C31C	000036			L	r15,796(,r12)
000210	184E	000036			LR	r4,r14
000212	05EF	000036			BALR	r14,r15
000214	00000000					=F'0'
000218	07F3	000036			BR	r3
00021A	90E5 D00C	000036			STM	r14,r5,12(r13)
00021E	58E0 D04C	000036			L	r14,76(,r13)
000222	4100 E100	000036			LA	r0,256(,r14)
000226	5500 C314	000036			CL	r0,788(,r12)
00022A	4130 F03A	000036			LA	r3,58(,r15)
00022E	4720 F014	000036			BH	20(,r15)
000232	58F0 C280	000036			L	r15,640(,r12)
000236	90F0 E048	000036			STM	r15,r0,72(r14)
00023A	9210 E000	000036			MVI	0(r14),16

Figure 12. Example of a C listing (Part 28 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
00023E 50D0 E004        000036 |          ST  r13,4(,r14)
000242 18DE          000036 |          LR  r13,r14
000244          End of Prolog

000244 C050 0000 002E 000000 |          LARL r5,F'46'
00024A 5010 D0B8        000036 |          ST  r1,#SR_PARM_2(,r13,184)
000037 *          double f_temp = (c_temp * CONV + OFFSET);
00024E 5810 D0B8        000037 |          L   r1,#SR_PARM_2(,r13,184)
000252 6800 1000        000037 |          LD  f0,c_temp(,r1,0)
000256 6820 5048        000037 |          LD  f2,+CONSTANT_AREA(,r5,72)
00025A 2C02          000037 |          MDR f0,f2
00025C 6820 5050        000037 |          LD  f2,+CONSTANT_AREA(,r5,80)
000260 2A02          000037 |          ADR f0,f2
000262 6000 D0B0        000037 |          STD f0,f_temp(,r13,176)
000038 *          printf("%5.2f Celsius is %5.2f Fahrenheit\n",c_temp, f_temp);
000266 5810 D0B8        000038 |          L   r1,#SR_PARM_2(,r13,184)
00026A 6820 1000        000038 |          LD  f2,c_temp(,r1,0)
00026E 6020 D09C        000038 |          STD f2,#MX_TEMP2(,r13,156)
000272 6000 D0A4        000038 |          STD f0,#MX_TEMP2(,r13,164)
000276 58F0 3066        000038 |          L   r15,=V(PRINTF)(,r3,102)
00027A 4100 5058        000038 |          LA  r0,+CONSTANT_AREA(,r5,88)
00027E 4110 D098        000038 |          LA  r1,#MX_TEMP2(,r13,152)
000282 5000 D098        000038 |          ST  r0,#MX_TEMP2(,r13,152)
000286 05EF          000038 |          BALR r14,r15
000039 * }
000288          000039 |          @2L14 DS  0H

000288          Start of Epilog
000288 58D0 D004        000039 |          L   r13,4(,r13)
00028C 58E0 D00C        000039 |          L   r14,12(,r13)
000290 9825 D01C        000039 |          LM  r2,r5,28(r13)
000294 051E          000039 |          BALR r1,r14
000296 0707          000039 |          NOPR 7

000298          Start of Literals
000298 00000000          =V(PRINTF)
00029C          End of Literals

*** General purpose registers used: 1101110000001111
*** Floating point registers used: 1111111100000000
*** Size of register spill area: 128(max) 0(used)
*** Size of dynamic storage: 256
*** Size of executable code: 160

00029C 0000 0000

Constant Area
0002A0 C595A385 9940C385 93A289A4 A240A385 |Enter Celsius te
0002B0 94978599 81A3A499 857A4015 006C8600 |mperature: ..%f.
0002C0 E896A440 94A4A2A3 408595A3 85994081 |You must enter a
0002D0 40A58193 898440A3 85949785 9981A3A4 |valid temperatu
0002E0 99851500 C9C2D440 411CCCCC CCCCCCCC |re..IBM .....
0002F0 42200000 00000000 6CF54BF2 8640C385 |.....%5.2f Ce
000300 93A289A4 A24089A2 406CF54B F28640C6 |lsius is %5.2f F
000310 81889985 95888589 A315006C A24089A2 |ahrenheit..%s is
000320 409596A3 408140A5 81938984 40A38594 |not a valid tem

```

Figure 12. Example of a C listing (Part 29 of 31)

```

5694A01 V1 R6 z/OS C          'TSCTEST.ZOSV1R6.SCCNSAM(CCNUAAM)': convert      11/21/2003 18:02:22  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
000330 97859981 A3A49985 1500          |perature..      |
5694A01 V1 R6 z/OS C          'TSCTEST.ZOSV1R6.SCCNSAM(CCNUAAM)':      11/21/2003 18:02:22  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
000340 1CCEA106                =F'483303686'    Flags
000344 000003B0                =A(PPA2-main)
000348 00000000                =F'0'            No PPA3
00034C 00000000                =F'0'            No EPD
000350 FF000000                =F'-16777216'   Register save mask
000354 00000000                =F'0'            Member flags
000358 90                        =AL1(144)        Flags
000359 000000                =AL3(0)          Callee's DSA use/8
00035C 0040                    =H'64'           Flags
00035E 0012                    =H'18'           Offset/2 to CDL
000360 00000000                =F'0'            Reserved
000364 500000E8                =F'1342177512'  CDL function length/2
000368 FFFFFFFC08                =F'-808'         CDL function EP offset
00036C 38260000                =F'942014464'   CDL prolog
000370 400900DF                =F'1074331871'  CDL epilog
000374 00000000                =F'0'            CDL end
000378 0004 ****                AL2(4),C'main'
                                PPA1 End
                                PPA1: Entry Point Constants
000380 1CCEA106                =F'483303686'    Flags
000384 000001D0                =A(PPA2-convert)
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 50000050                =F'1342177360'  CDL function length/2
0003A8 FFFFFFFE78                =F'-392'         CDL function EP offset
0003AC 38260000                =F'942014464'   CDL prolog
0003B0 40080048                =F'1074266184'  CDL epilog
0003B4 00000000                =F'0'            CDL end
0003B8 0007 ****                AL2(7),C'convert'
                                PPA1 End
                                PPA2: Compile Unit Block
0003C8 0300 2202                =F'50340354'    Flags
0003CC FFFF FC38                =A(CEESTART-PPA2)
0003D0 0000 0000                =F'0'            No PPA4
0003D4 FFFF FC38                =A(TIMESTMP-PPA2)
0003D8 0000 0000                =F'0'            No primary
0003DC 0000 0000                =F'0'            Flags
                                PPA2 End

```

Figure 12. Example of a C listing (Part 30 of 31)

```

5694A01 V1 R6 z/OS C          'TSCTEST.ZOSV1R6.SCCNSAM(CCNUAAM)'          11/21/2003 18:02:22 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
CONVERT   PC    1 000000 0003E0      MAIN      LD    0 000018 000001
PRINTF    LD    0 0001F8 000001      CEESG003   ER    2 000000
SSCANF    ER    3 000000      SCANF      ER    4 000000
CEEMAIN   ER    5 000000      CEESTART   ER    6 000000
MAIN      SD    7 000000 00000C      EDCINPL    ER    8 000000
5694A01 V1 R6 z/OS C          'TSCTEST.ZOSV1R6.SCCNSAM(CCNUAAM)'          11/21/2003 18:02:22 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 R6 z/OS C          'TSCTEST.ZOSV1R6.SCCNSAM(CCNUAAM)'          11/21/2003 18:02:22 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>:<FILE LINE NO>
argc            7-0:7            Class = parameter,      Location = 0(r1),      Length = 4
argv            7-0:7            Class = parameter,      Location = 4(r1),      Length = 4
c_temp          9-0:9            Class = automatic,      Location = 176(r13),   Length = 8
i               24-0:24           Class = automatic,      Location = 184(r13),   Length = 4
c_temp          36-0:36           Class = parameter,      Location = 0(r1),      Length = 8
f_temp          37-0:37           Class = automatic,      Location = 176(r13),   Length = 8
                                * * * * *   E N D   O F   S T O R A G E   O F F S E T   L I S T I N G   * * * * *
                                * * * * *   E N D   O F   C O M P I L A T I O N   * * * * *

```

Figure 12. 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 272 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 right-most 32 characters appear in the listing.

Prolog Section

The Prolog section provides information about the compile-time library, file identifiers, compiler options, and other items in effect when the compiler was invoked.

All options except those with no default (for example, DEFINE) are shown in the listing. Any problems with the compiler options appear after the body of the Prolog section.

IPA Considerations: If you specify IPA suboptions that are irrelevant to the IPA Compile step, the Prolog does not display them. If IPA processing is not active, IPA suboptions do not appear in the Prolog.

The following 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 *z/OS 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 104, 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 noline` and the `COMPACT` compiler option are not in effect.
 - The F indicates that the subprogram was declared inline, either by `#pragma inline` for C or the `inline` keyword for C++.
 - The C indicates that the `COMPACT` compiler option is specified for `#pragma_override(FuncName, "OPT (COMPACT, yes) "` is specified in the source code.
 - The M indicates that C++ routine is an inline member routine.
 - The A indicates automatic inlining acted on the subprogram.
 - The - indicates there was no reason to inline the subprogram.
- Action on a subprogram:
 - Subprogram was inlined at least once.
 - Subprogram was not inlined because of initial size constraints.
 - Subprogram was not inlined because of expansion beyond size constraint.
 - Subprogram was a candidate for inlining, but was not inlined.
 - Subprogram was a candidate for inlining, but was not referenced.
 - The subprogram is directly recursive, or some calls have mismatching parameters.

Note: "Called" and "Calls" in the actions section of the inline report indicate how many times a function has been called or has called other functions, regardless of whether or not the callers or callees have been inlined.

- Status of original subprogram after inlining:
 - Subprogram is discarded because it is no longer referenced and is defined as `static internal`.
 - Subprogram was not discarded for various reasons :
 - Subprogram is external. (It can be called from outside the compilation unit.)
 - A call to this subprogram remains.
 - Subprogram has its address taken.
- Initial relative size of subprogram (in Abstract Code Units (ACU)).
- Final relative size of subprogram (in ACUs) after inlining.
- Number of calls within the subprogram and the number of these calls that were inlined into subprogram.
- Number of times the subprogram is called by others in the compile unit and the number of times the subprogram was inlined.
- Mode that is selected and the value of *threshold* and *limit* specified for the compilation.

The detailed call structure contains specific information of each subprogram such as:

- Subprograms that it calls
- Subprograms that call it

- Subprograms in which it is inlined

The information can help you to better analyze your program if you want to use the inliner in selective mode.

Inlining may result in additional messages. For example, if inlining a subprogram with automatic storage increases the automatic storage of the subprogram it is being inlined into by more than 4K, a message is generated.

Pseudo Assembly Listing

The 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.

Notes:

1. Although the compiler listing is for your use, it is not a programming interface and is subject to change.

2. The compiler always attempts to put diagnostic messages in the listing, as close as possible to the location where the condition occurred. The exact location or line number within the listing may not be the same from release to release.

IPA Considerations

The listings that the IPA Compile step produces are basically the same as those that a regular compilation produces. Any differences are noted throughout this section.

The IPA Link step listing has a separate format from the 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 272 for information about IPA Link step listings.

Example of a C++ Compiler Listing

Figure 13 on page 259 shows an example of a z/OS C++ compiler listing. Vertical ellipses indicate sections that have been truncated.

```

5694A01 V1 R6 z/OS C++          //'TSCTEST.ZOSV1R6.SCCNSAM(CCNUBRC)'          11/21/03 06:01:56
                                * * * * * P R O L O G * * * * *
Compiler options. . . . . :AGGRCOPY(NOOVERLAP)      ANSIALIAS      ARCH(5)      ARGPARSE      NOASCII
                          :NOATTRIBUTE          BITFIELD(UNSIGNED)  CHARS(UNSIGNED) NOCOMPACT      NOCOMPRESS
                          :NOCSECT              CVFT              DIGRAPH      DLL(NOCALLBACKANY)  ENUMSIZE(SMALL)
                          :NOEVENTS            EXECOPS           EXH           NOEXPMAC       NOEXPORTALL  NOFASTTEMPINC
                          :FLAG(I)              NOGOFF           NOGONUMBER   HALT(16)       NOIGNERRNO   ILP32
                          :NOINITAUTO          INLRPT           NOLIBANSI    LIST           LONGNAME      LONGLONG
                          :NOMARGINS            MAXMEM(2097152)  MEMORY       NAMEMANGLING=ANSI  NESTINC(255)
                          :OBJECT              OBJECTMODEL(COMPAT) NOE           NOOFFSET       NOOPTIMIZE
                          :PLIST(HOST)         NOPORT           NOPPONLY     REDIR          ROSTRING      ROCONST
                          :NORTTI              NOSEQUENCE       NOSHOWINC    SOURCE         SPILL(128)    NOSQL
                          :START              NOSTATICINLINE   STRICT       NOSTRICT_INDUCTION
                          :TARGET(LE,CURRENT)   NOTEMPLATEREGISTRY  TEMPLATERECOMPILE
                          :TERMINAL            NOTEST(HOOK)     TMLPARSE(NO) TUNE(5)       UNROLL(AUTO)  NOWARN64
                          :NWSIZEOF           XREF
                          :NOCONVLIT
                          :NODEBUG
                          :FLOAT(HEX,FOLD,AFP) ROUND(Z)
                          :INFO(LAN)
                          :INLINE(AUTO,REPORT,100,1000)
                          :NOIPA
                          :LANGLVL(ANONSTRUCT,ANONUNION,ANSIFOR,DBCS,NODOLLARINAMES,EMPTYSTRUCT,ILLPTOM,IMPLICITINT,LIBEXT,
                          :LONGLONG,OFFSETNONPOD,NOOLDDIGRAPH,OLDFRIEND,NOOLDMATH,OLDTEMPACC,NOOLDTMLALIGN,OLDTMPLSPEC,
                          :TRAILENUM,TYPEDEFCLASS,NOUCS,ZEROEXTARRAY)
                          :NOLOCALE
                          :LSEARCH()
                          :OPTFILE(DD:OPTS)
                          :SEARCH('//'TSCTEST.CEEZ160.SCEEH.+' //'TSCTEST.ZOSV1R6.SCLBH.+' )
                          :NOSERVICE
                          :NOSUPPRESS
                          :TEMPINC(/tempinc)
                          :NOXLINK(NOBACKCHAIN,NOCALLBACK,GUARD,OSCALL(UPSTACK),NOSTOREARGS)
Version Macros. . . . . : __COMPILER_VER__=0x41060000
                          : __LIBREL__=0x41060000
                          : __TARGET_LIB__=0x41060000
Source margins. . . . . :
  Varying length. . . . . : 1 - 32767
  Fixed length. . . . . : 1 - 32767
Sequence columns. . . . . :
  Varying length. . . . . : none
  Fixed length. . . . . : none
Listing name. . . . . : DD:SYSCPRT
                                * * * * * E N D   O F   P R O L O G   * * * * *

```

```

5694A01 V1 R6 z/OS C++          //'TSCTEST.ZOSV1R6.SCCNSAM(CCNUBRC)'          11/21/03 06:01:56

```

```

                                * * * * * 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
21     BioRhythm bio;
22     int code;
23
24     if (!bio.ok()) {
25         cerr << "Error in birthdate specification - format is yyyy/mm/dd";
26         code = 8;
27     }
28     else {
29         cout << bio; // write out birthdate for bio
30         code = 0;
31     }
32     return(code);

```

Figure 13. Example of a C++ Compiler Listing (Part 1 of 14)

```

33 }
34
35 ostream& operator<<(ostream& os, BioRhythm& bio) {
36     os << "Total Days : " << bio.AgeInDays() << "\n";
37     os << "Physical : " << bio.Physical() << "\n";
38     os << "Emotional : " << bio.Emotional() << "\n";
39     os << "Intellectual: " << bio.Intellectual() << "\n";
40
41     return(os);
42 }
43
44 Date::Date() {
45     time_t lTime;
46     struct tm *newTime;
47
48     time(&lTime);
49     newTime = localtime(&lTime);
50     cout << "local time is " << asctime(newTime) << endl;
51
52     curYear = newTime->tm_year + 1900;
53     curDay = newTime->tm_yday + 1;
54 }
55
56 BirthDate::BirthDate(const char *birthText) {
57     strcpy(text, birthText);
58 }
59
60 BirthDate::BirthDate() {
61     cout << "Please enter your birthdate in the form yyyy/mm/dd\n";
62     cin >> setw(dateLen+1) >> text;
63 }
64
65 Date::DaysSince(const char *text) {
66
67     int year, month, day, totDays, delim;
68     int daysInYear = 0;
69     int i;
70     int leap = 0;
71
72     int rc = sscanf(text, "%4d%c%2d%c%2d",
73                   &year, &delim, &month, &delim, &day);
74     --month;
75     if (rc != 5 || year < 0 || year > 9999 ||
76         month < 0 || month > 11 ||
77         day < 1 || day > 31 ||
78         (day > numDays[month]&& month != 1)) {
79         return(-1);
80     }
81     if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
82         leap = 1;
83
84     if (month == 1 && day > numDays[month]) {
85         if (day > 29)
86             return(-1);
87         else if (!leap)
88             return (-1);
89     }
90
91     for (i=0; i<month> 1 || (month == 1 && day == 29))
92         ++daysInYear;
93
94     totDays = (curDay - daysInYear) + (curYear - year)*365;
95
96     // now, correct for leap year
97     for (i=year+1; i < curYear; ++i) {
98         if ((i % 4 == 0 && i % 100 != 0) || i % 400 == 0) {
99             ++totDays;
100         }
101     }
102     return(totDays);
103 }
104
105 * * * * *   E N D   O F   S O U R C E   * * * * *

```

Figure 13. Example of a C++ Compiler Listing (Part 2 of 14)

```

5694A01 V1 R6 z/OS C++          //'TSCTEST.ZOSV1R6.SCCNSAM(CCNUBRC)'          11/21/03 06:01:56
                                * * * * * I N C L U D E S * * * * *
1 = //'TSCTEST.CEEZ160.SCEEH.H(STDIO)'
2 = //'TSCTEST.CEEZ160.SCEEH.H(FEATURES)'
3 = //'TSCTEST.CEEZ160.SCEEH.SYS.H(TYPES)'
4 = //'TSCTEST.CEEZ160.SCEEH.H(STRING)'
5 = //'TSCTEST.CEEZ160.SCEEH.H(MATH)'
6 = //'TSCTEST.CEEZ160.SCEEH.H(BUILTINS)'
7 = //'TSCTEST.CEEZ160.SCEEH.H(TIME)'
8 = //'TSCTEST.CEEZ160.SCEEH(IOSTREAM)'
9 = //'TSCTEST.CEEZ160.SCEEH(ISTREAM)'
10 = //'TSCTEST.CEEZ160.SCEEH(OSTREAM)'
11 = //'TSCTEST.CEEZ160.SCEEH.H(YVALS)'
12 = //'TSCTEST.CEEZ160.SCEEH(IOS)'
13 = //'TSCTEST.CEEZ160.SCEEH(XLOCNUM)'
14 = //'TSCTEST.CEEZ160.SCEEH(CERRNO)'
15 = //'TSCTEST.CEEZ160.SCEEH.H(ERRNO)'
16 = //'TSCTEST.CEEZ160.SCEEH(CLIMITS)'
17 = //'TSCTEST.CEEZ160.SCEEH.H(LIMITS)'
18 = //'TSCTEST.CEEZ160.SCEEH(CSTDIO)'
19 = //'TSCTEST.CEEZ160.SCEEH(CSTDLIB)'
20 = //'TSCTEST.CEEZ160.SCEEH.H(STDLIB)'
21 = //'TSCTEST.CEEZ160.SCEEH(STREAMBU)'
22 = //'TSCTEST.CEEZ160.SCEEH(XIOSBASE)'
:
                                * * * * * E N D O F I N C L U D E S * * * * *

```

Figure 13. Example of a C++ Compiler Listing (Part 3 of 14)

```

5694A01 V1 R6 z/OS C++          //'TSCTEST.ZOSV1R6.SCCNSAM(CCNUBRC)'          11/21/03 06:01:56
                                * * * * * C R O S S R E F E R E N C E L I S T I N G * * * * *
__valist          :
1:133 (D)         1:136 (R)
__abs             :
20:308 (R)        20:389 (R)
__absd           :
5:932 (R)         5:1009(R)
__absf           :
5:931 (R)         5:1008(R)
__absl           :
5:933 (R)         5:1010(R)
__acos           :
5:917 (R)         5:1012(R)
__acosf          :
5:934 (R)         5:1011(R)
__acosl          :
5:935 (R)         5:1013(R)
:
                                * * * * * E N D O F C R O S S R E F E R E N C E L I S T I N G * * * * *

```

Figure 13. Example of a C++ Compiler Listing (Part 4 of 14)

```

5694A01 V1 R6 z/OS C++          //'TSCTEST.ZOSV1R6.SCCNSAM(CCNUBRC)'          11/21/03 06:01:56
          * * * * * 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             2
          * * * * * E N D   O F   M E S S A G E   S U M M A R Y   * * * * *
15694A01 V1 R6 z/OS C++          CCNUBRC          11/21/03 06:01:56

```

Inline Report (Summary)

```

Reason:  P : noline 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	D	52 (28)	3/2	3/3	std::ostreambuf_iterator<char, std::char_traits<char> >::operator(char)
A	I	D	22 (6)	1/1	2/2	std::_EBCDIC::_LFS_OFF::basic_string<char, std::char_traits<char>, std::allocator<char> >::operator=(const std::_EBCDIC::_LFS_OFF::basic_string<char, std::char_traits<char>, std::allocator<char> >& std::ostreambuf_iterator<char, std::char_traits<char> >::ostream_buf_iterator(std::_EBCDIC::_LFS_OFF::basic_streambuf<char, std::char_traits<char> >*)
A	N	-	10	0/0	0/0	std::bad_cast::bad_cast(const char*)
A	I	D	27 (10)	1/1	3/3	std::bad_cast::bad_cast(const std::bad_cast&)
M	I	D	28 (10)	1/1	1/1	std::bad_cast::bad_cast(const std::bad_cast&)

Mode = AUTO Inlining Threshold = 100 Expansion Limit = 1000

Figure 13. Example of a C++ Compiler Listing (Part 5 of 14)

```

15694A01 V1 R6 z/OS C++          CCNUBRC          11/21/03 06:01:56
          Inline Report (Call Structure)
Defined Function      : std::ostreambuf_iterator<char, std::char_traits<char> >::operator=(char)
Calls To(3,2)       : std::char_traits<char>::eof()(1,1)
                    : std::char_traits<char>::eq_int_type(const int&, const int&)(1,1)
                    : std::_EBCDIC::_LFS_OFF::basic_streambuf<char, std::char_traits<char> >::sputc(char)(1,0)
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)

```

Figure 13. Example of a C++ Compiler Listing (Part 6 of 14)

Inline Report (Additional Information)

```
INFORMATIONAL CCN1052: Function BioRhythm::BioRhythm() is (or grows) too large to be inlined.
INFORMATIONAL CCN1052: Function operator<<(std::_EBCDIC::_LFS_ is (or grows) too large to be inlined.
INFORMATIONAL CCN1052: Function std::ostreambuf_iterator<char, is (or grows) too large to be inlined.
INFORMATIONAL CCN1052: Function std::ostreambuf_iterator<char, is (or grows) too large to be inlined.
INFORMATIONAL CCN1052: Function std::_EBCDIC::_LFS_OFF::num_pu is (or grows) too large to be inlined.
INFORMATIONAL CCN1052: Function std::_EBCDIC::_LFS_OFF::basic_ is (or grows) too large to be inlined.
INFORMATIONAL CCN1052: Function std::_EBCDIC::_LFS_OFF::basic_ is (or grows) too large to be inlined.
INFORMATIONAL CCN1052: Function std::_EBCDIC::_LFS_OFF::num_pu is (or grows) too large to be inlined.
:
```

Figure 13. Example of a C++ Compiler Listing (Part 7 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

Timestamp and Version Information

```
000000 F2F0 F0F3                                   =C'2003'           Compiled Year
000004 F1F1 F2F1                                   =C'1121'           Compiled Date MMDD
000008 F1F8 F0F1 F5F9                           =C'180159'         Compiled Time HHMMSS
00000E F0F1 F0F6 F0F0                           =C'010600'         Compiler Version
```

Timestamp and Version End

OFFSET OBJECT CODE LINE# FILE# P S E U D O A S S E M B L Y L I S T I N G

```
000001 * //
000002 * // Sample Program: Biorhythm
000003 * // Description : Calculates biorhythm based on the current
000004 * // system date and birth date entered
000005 * //
000006 * // File 2 of 2-other file is CCNUBRH
000007 *
000008 * #include <stdio.h>
000009 * #include <string.h>
000010 * #include <math.h>
000011 * #include <time.h>

000012 * #include <iostream>
000013 * #include <iomanip>
000014 *
000015 * using namespace std;
000016 *
000017 * #include "ccnubr.h" //BioRhythm class and Date class
000018 *
000019 * int main(void) {
000019 main    DS    0D
000019        B    40(,r15)
00001C           CEE eyecatcher
000020           DSA size
000024           =A(PPA1-main)
000028 47F0 F001      000019        B    1(,r15)
00002C 58F0 C31C      000019        L    r15,796(,r12)
000030 184E            000019        LR    r4,r14
000032 05EF            000019        BALR r14,r15
000034 00000000               =F'0'
000038 0540            000019        BALR r4,0
00003A 4140 401E      000019        LA    r4,30(,r4)
00003E 07F4            000019        BR    r4
000040 90E6 D00C      000019        STM   r14,r6,12(r13)
000044 58E0 D04C      000019        L    r14,76(,r13)
000048 4100 E0E0      000019        LA    r0,224(,r14)
00004C 5500 C314      000019        CL    r0,788(,r12)
000050 4140 F040      000019        LA    r4,64(,r15)
000054 4720 F014      000019        BH    20(,r15)
000058 5000 E04C      000019        ST    r0,76(,r14)
00005C 9210 E000      000019        MVI   0(r14),16
000060 50D0 E004      000019        ST    r13,4(,r14)
000064 18DE            000019        LR    r13,r14
000066                End of Prolog

000066 5800 C1F4      000019        L    r0,_CEECAA_(,r12,500)
00006A 5000 D0DC      000019        ST    r0,#CEECAACRENT_1(,r13,220)
00006E 5810 D0DC      000019        L    r1,#CEECAACRENT_1(,r13,220)
000072 5820 4144      000019        L    r2,=Q(@STATIC)(,r4,324)
000076 4152 1000      000019        LA    r5,=Q(@STATIC)(r2,r1,0)
00007A C060 0000 6803 000000        LARL r6,F'26627'
000080 4100 0000      000019        LA    r0,0
000084 5000 D098      000019        ST    r0,__es__l00(,r13,152)
000088 4110 5118      000019        LA    r1,__fsm_tab(,r5,280)
00008C 5010 D09C      000019        ST    r1,__es__t04(,r13,156)
000090 5000 D0A0      000019        ST    r0,__es__this08(,r13,160)
000094 5000 D0A4      000019        ST    r0,__es__i0c(,r13,164)
```

Figure 13. Example of a C++ Compiler Listing (Part 8 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
000020      *
000021      *   BioRhythm bio;
000021      LA   r1,bio,r13,168)
000098 4110 D0A8
00009C 5800 D0DC
0000A0 58F0 4148
0000A4 4DE0 F010
0000A8 4700 0004
0000AC 9201 D09B
000021      MVI  __es.__c4@3(r13,155),1
000022      *   int code;
000023      *
000024      *   if (!bio.ok()) {
000024      LA   r1,bio,r13,168)
0000B4 5010 D0C8
0000B8 5810 1000
0000BC 1211
0000BE 4100 0001
0000C2 A744 0004
0000C6 4100 0000
0000CA 5700 413C
0000CE 5400 4140
0000D2 5400 4140
0000D6 5000 D0CC
0000DA
0000DA 5800 D0CC
0000DE 1200
0000E0 4770 40B0
000024      ST   r1,this:ok__9BioRhythmFv(,r13,200)
000060      L    r1,(BioRhythm).age@0(,r1,0)
000060      LTR  r1,r1
000060      LA   r0,1
000060      JL   H'4'
000060      LA   r0,0
000060      X    r0,=F'1'
000060      N    r0,=F'255'
000060      N    r0,=F'255'
000060      ST   r0,retval:ok__9BioRhythmFv(,r13,204)
000061      DS   0H
000061      L    r0,retval:ok__9BioRhythmFv(,r13,204)
000061      LTR  r0,r0
000061      BNE @1L27
000025      *   cerr << "Error in birthdate specification - format is yyyy/mm/dd";
000025      L    r1,cerr_Q3_3std7_EBCDIC8_LFS_OFF(,r5,0)
000025      L    r0,#CEECAACRENT_1(,r13,220)
000025      L    r15,=V(std::_EBCDIC::LFS_OFF::basic_ostream<char,std::cha...(,r4,332)
000025      LA   r2,+CONSTANT_AREA(,r6,121)
000025      BAS  r14,16(,r15)
000025      NOP  8
000026      *   code = 8;
000026      LA   r0,8
000026      ST   r0,code(,r13,192)
000027      *   }
000027      B    @1L29
000027      DS   0H
000028      *   else {
000029      *   cout << bio; // write out birthdate for bio
000029      L    r1,cout_Q3_3std7_EBCDIC8_LFS_OFF(,r5,4)
000029      LA   r2,bio(,r13,168)
000029      L    r0,#CEECAACRENT_1(,r13,220)
000029      L    r15,=A(operator<<(std::_EBCDIC::LFS_OFF::basic_ostream<ch...(,r4,336)
000029      BAS  r14,16(,r15)
000029      NOP  8
000030      *   code = 0;
000030      LA   r0,0
000030      ST   r0,code(,r13,192)
000031      *   }
000031      @1L29 DS   0H
000032      *   return(code);
000032      L    r0,code(,r13,192)
00012C 5000 D0C4
000130 9200 D09B
000032      ST   r0,__14(,r13,196)
000032      MVI  __es.__c4@3(r13,155),0

```

Figure 13. 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
000134 4100 0002      000032 |          LA    r0,2
000138 5000 D0D4      000032 |          ST    r0,_dctorFlags:_dt__9BioRhythmFv(,r13,212)
00013C 4100 D0A8      000032 |          LA    r0,bio(,r13,168)
000140 5000 D0D0      000032 |          ST    r0,this:_dt__9BioRhythmFv(,r13,208)
000144 5800 D0D4      000045 |          6 +    L    r0,_dctorFlags:_dt__9BioRhythmFv(,r13,212)
000148 5400 413C      000045 |          6 +    N    r0,='1'
00014C 1200          000045 |          6 +    LTR  r0,r0
00014E 4780 4112      000045 |          6 +    BE   @1L35
000152 5810 D0D0      000045 |          6 +    L    r1,this:_dt__9BioRhythmFv(,r13,208)
000156 5820 5008      000045 |          6 +    L    r2,_d1_FPv(,r5,8)
00015A 58F0 2008      000045 |          6 +    L    r15,&Func_&WSA(,r2,8)
00015E 5800 200C      000045 |          6 +    L    r0,&Func_&WSA(,r2,12)
000162 4DE0 F010      000045 |          6 +    BAS  r14,16(,r15)
000166 4700 0004      000045 |          6 +    NOP  4

00016A          000045 |          6 +@1L35 DS   0H
00016A 5800 D0D0      000045 |          6 +    L    r0,this:_dt__9BioRhythmFv(,r13,208)
00016E 5000 D0D8      000045 |          6 +    ST    r0,retval:_dt__9BioRhythmFv(,r13,216)
000172          000045 |          6 +@1L39 DS   0H
000172 5800 D0D8      000045 |          6 +    L    r0,retval:_dt__9BioRhythmFv(,r13,216)
000176 58F0 D0C4      000045 |          6 +    L    r15,_14(,r13,196)
00017A          000033 |          *   }
00017A          000033 |          @1L2660 DS  0H
00017A 5800 D0DC      000019 |          L    r0,#CEECAACRENT_1(,r13,220)
00017E 5000 C1F4      000019 |          ST    r0,_CEECAA_(,r12,500)

000182          Start of Epilog
000182 180D          000033 |          LR    r0,r13
000184 58D0 D004      000033 |          L    r13,4(,r13)
000188 58E0 D00C      000033 |          L    r14,12(,r13)
00018C 9826 D01C      000033 |          LM   r2,r6,28(r13)
000190 051E          000033 |          BALR r1,r14
000192 0707          000033 |          NOPR 7

000194          Start of Literals
000194 00000001          =F'1'
000198 000000FF          =F'255'
00019C 00000000          =Q(@STATIC)

0001A0 00000000          =V(BioRhythm::BioRhythm())

0001A4 00000000          =V(std::_EBCDIC::_LFS_OFF::basic_ostream<char,std::char_traits<char> >
0001A8 00000AB8          =A(operator<<(std::_EBCDIC::_LFS_OFF::basic_ostream<char,std::char_tra
0001AC          End of Literals

*** General purpose registers used: 1110111000001111
*** Floating point registers used: 1111111100000000
*** Size of register spill area: 128(max) 0(used)
*** Size of dynamic storage: 224
*** Size of executable code: 380
:

```

Figure 13. Example of a C++ Compiler Listing (Part 10 of 14)

EXTERNAL SYMBOL DICTIONARY

TYPE	ID	ADDR	LENGTH	NAME
SD	1	000000	011160	@STATICP
PR	2	000000	0006E0	@STATIC
PR	3	000000	000004	dateLen__4Date
PR	4	000000	000004	numMonths__4Date
PR	5	000000	000004	pCycle__9BioRhythm
PR	6	000000	000004	eCycle__9BioRhythm
PR	7	000000	000004	iCycle__9BioRhythm
PR	8	000000	000030	numDays__4Date
PR	9	000000	000004	_Psave_use_facet_Q3_3std7_EBCDIC8_LFS_OFFHQ4_3std7_EBCDIC8_LFS_OFF5cty peXtc__RCQ4_3std7_EBCDIC8_LFS_OFF61o cale_RCQ4_3std7_EBCDIC8_LFS_OFF5ctyp eXtc__2
PR	10	000000	000004	_Psave_use_facet_Q3_3std7_EBCDIC8_LFS_OFFHQ4_3std7_EBCDIC8_LFS_OFF7num _putXtcTQ2_3std19ostreambuf_iterator XtcTQ2_3std11char_traitsXtc__RCQ4_3std7_EBCDIC8_LFS_OFF61ocale_RCQ4_3s td7_EBCDIC8_LFS_OFF7num_putXtcTQ2_3s td19ostreambuf_iteratorXtcTQ2_3std11 char_traitsXtc__2
PR	11	000000	000004	id_Q4_3std7_EBCDIC8_LFS_OFF7num_put XtcTQ2_3std19ostreambuf_iteratorXtcT Q2_3std11char_traitsXtc__
PR	12	000000	000030	_vftQ2_3std8bad_castQ2_3std9excepti on__3std
PR	13	000000	000004	_Facsav_Q4_3std7_EBCDIC8_LFS_OFF8_T idyfacXTQ4_3std7_EBCDIC8_LFS_OFF5cty peXtc__

:

Figure 13. Example of a C++ Compiler Listing (Part 11 of 14)

EXTERNAL SYMBOL CROSS REFERENCE

ORIGINAL NAME	EXTERNAL SYMBOL NAME
@STATICP	@STATICP
@STATIC	@STATIC
Date::dateLen	dateLen__4Date
Date::numMonths	numMonths__4Date
BioRhythm::pCycle	pCycle__9BioRhythm
BioRhythm::eCycle	eCycle__9BioRhythm
BioRhythm::iCycle	iCycle__9BioRhythm
Date::numDays	numDays__4Date
const std::_EBCDIC::_LFS_OFF::ctype <char>& 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_OFF61o cale_RCQ4_3std7_EBCDIC8_LFS_OFF5ctyp eXtc__2

:

Figure 13. Example of a C++ Compiler Listing (Part 12 of 14)

```

***** STORAGE OFFSET LISTING *****

IDENTIFIER      DEFINITION      ATTRIBUTES
                  <SEQNBR>-<FILE NO>:<FILE LINE NO>

code            22-0:22      Class = automatic,      Location = 192(r13),      Length = 4
bio            21-0:21      Class = automatic,      Location = 168(r13),      Length = 24
newTime       46-0:46      Class = automatic,      Location = 204(r13),      Length = 4
lTime        45-0:45      Class = automatic,      Location = 200(r13),      Length = 4
birthText     56-0:56      Class = parameter,      Location = 192(r13),      Length = 4
totDays      67-0:67      Class = automatic,      Location = 260(r13),      Length = 4
i            69-0:69      Class = automatic,      Location = 256(r13),      Length = 4
day          67-0:67      Class = automatic,      Location = 252(r13),      Length = 4
month        67-0:67      Class = automatic,      Location = 248(r13),
:
***** END OF STORAGE OFFSET LISTING *****

```

Figure 13. Example of a C++ Compiler Listing (Part 13 of 14)

```

***** STATIC MAP *****

OFFSET (HEX)  LENGTH (HEX)  NAME
0             4      cerr_Q3_3std7_EBCDIC8_LFS_OFF
4             4      cout_Q3_3std7_EBCDIC8_LFS_OFF
8             4      __dl__FPv
C             4      time
10            4      localtime
14            4      asctime
18            4      endl_Q3_3std7_EBCDIC8_LFS_OFFHcQ2_3std11char_traitsXtc__RQ4_3std7_EBCDIC8_LFS_OFF13basic_ostream
Q2_3std11char_traitsXtc__RQ4_3std7_EBCDIC8_LFS_OFF13basic_ostreamXtcQ2_3std11char_traitsXtc__
1C            4      cin_Q3_3std7_EBCDIC8_LFS_OFF
20            4      setw_Q3_3std7_EBCDIC8_LFS_OFFFi
24            4      sscanf
28            4      numDays__4Date
2C            4      clear_Q4_3std7_EBCDIC8_LFS_OFF8ios_baseFib
30            4      __CleanupCatch
34            4      __fmod
38            4      __sin
3C            4      __dt_Q2_3std7_LockitFv
40            4      id_Q4_3std7_EBCDIC8_LFS_OFF5ctypeXtc_
:
***** END OF STATIC MAP *****
***** END OF COMPILATION *****

```

Figure 13. 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 272 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 right-most 32 characters appear in the listing.

Prolog Section

The Prolog section provides information about the compile-time library, file identifiers, compiler options, and other items in effect when the compiler was invoked.

All options except those with no default (for example, DEFINE) are shown in the listing. Any problems with the compiler options appear after the body of the Prolog section.

IPA Considerations: If you specify IPA suboptions that are irrelevant to the IPA Compile step, the Prolog does not display them. If IPA processing is not active, IPA suboptions do not appear in the Prolog.

The following 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 *z/OS 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 noline 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 13, “Filter Utility,” on page 433.

Storage Offset Listing

If you specify the `XREF` option, the listing file includes offset information on identifiers.

Static Map

Static Map displays the contents of the `@STATIC` data area, which holds the file scope read/write static variables. It displays the offset (as a hexadecimal number), the length (as a hexadecimal number), and the names of the objects mapped to `@STATIC`. Under certain circumstances, the compiler may decide to map other objects to `@STATIC`.

If you specify the ATTR or XREF option, the listing file includes offset information for file scope read/write static variables.

Using the IPA Link Step Listing

The IPA Link step generates a listing file if you specify any of the following options:

- ATTR
- INLINE(,REPORT,,)
- INLRPT
- IPA(MAP)
- LIST
- XREF

Note: IPA does not support source listings or source annotations within Pseudo Assembly listings. The Pseudo Assembly listings do display the file and line number of the source code that contributed to a segment of pseudo assembly code.

Example of an IPA Link Step Listing

Figure 14 on page 273 shows an example of an IPA Link step listing.

***** PROLOG *****

```

Compile Time Library . . . . . : 41060000
Command options:
Primary input name. . . . . : 'TSIPA.TEST.LINKCNTL(INCLCNTL)'
Compiler options. . . . . : *IPA(LINK,MAP,NOREFMAP,LEVEL(1),DUP,ER,NONCAL,NOUNPCASE,NOPDF1,NOPDF2,NOPDFNAME,NOCONTROL)
                          : *NOGONUMBER *NOALIAS *TERMINAL *LIST *XREF *NOATTR *NOOFFSET
                          : *MEMORY *NOCSECT *LIBANSI *FLAG(1)
                          : *NOTEST(NOSYM,NOBLOCK,NOLINE,NOPATH,HOOK) *OPTIMIZE(2)
                          : *INLINE(AUTO,REPORT,1000,8000) *OPTFILE(DD:OPTION) *NOSERVICE *NOOE
                          : *NOLOCALE *HALT(16) *NOGOFF
    
```

***** END OF PROLOG *****

***** OBJECT FILE MAP *****

```

*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
    
```

***** END OF OBJECT FILE MAP *****

***** COMPILER OPTIONS MAP *****

```

SOURCE FILE ID  COMPILER OPTIONS
1
*AGGRCOPY(NOOVERLAP) *NOALIAS *ANSIALIAS *ARCH(5) *ARGPARSE
*CHARSET(BIAS=EBCDIC,LIB=EBCDIC) *NOCOMPACT *NOCOMPRESS *NODLL(NOCALLBACKANY) *ENV(MVS)
*EXECOPS *FLOAT(HEX,FOLD,AFP) *GONUMBER *NOIGNERRNO *ILP32 *NOINITAUTO
*IPA(NOLINK,NOOBJECT,COMPRESS) *NOLIBANSI *NOLIST *NOLOCALE *LONGNAME
*MAXMEM(2097152) *OPTIMIZE(2) *PLIST(HOST) *REDIR *NORENT *NOROCONST *SPILL(128)
*NOSTART *STRICT *NOSTRICT_INDUCTION *TEST(NOSYM,NOBLOCK,LINE,NOPATH,HOOK) *TUNE(5)
*NOXPLINK *NOXREF

2
*AGGRCOPY(NOOVERLAP) *NOALIAS *ANSIALIAS *ARCH(5) *ARGPARSE
*CHARSET(BIAS=EBCDIC,LIB=EBCDIC) *NOCOMPACT *NOCOMPRESS *NODLL(NOCALLBACKANY) *ENV(MVS)
*EXECOPS *FLOAT(HEX,FOLD,AFP) *NOGONUMBER *NOIGNERRNO *ILP32 *NOINITAUTO
*IPA(NOLINK,NOOBJECT,COMPRESS) *NOLIBANSI *NOLIST *NOLOCALE *LONGNAME
*MAXMEM(2097152) *OPTIMIZE(2) *PLIST(HOST) *REDIR *NORENT *NOROCONST *SPILL(128)
*NOSTART *STRICT *NOSTRICT_INDUCTION *NOTEST *TUNE(5) *NOXPLINK *NOXREF

3
*AGGRCOPY(NOOVERLAP) *NOALIAS *ANSIALIAS *ARCH(5) *ARGPARSE
*CHARSET(BIAS=EBCDIC,LIB=EBCDIC) *NOCOMPACT *NOCOMPRESS *NODLL(NOCALLBACKANY) *ENV(MVS)
*EXECOPS *FLOAT(HEX,FOLD,AFP) *NOGONUMBER *NOIGNERRNO *ILP32 *NOINITAUTO
*IPA(NOLINK,NOOBJECT,COMPRESS) *NOLIBANSI *NOLIST *NOLOCALE *LONGNAME
*MAXMEM(2097152) *OPTIMIZE(2) *PLIST(HOST) *REDIR *NORENT *NOROCONST *SPILL(128)
*NOSTART *STRICT *NOSTRICT_INDUCTION *NOTEST *TUNE(5) *NOXPLINK *NOXREF
    
```

***** END OF COMPILER OPTIONS MAP *****

Figure 14. Example of an IPA Link Step Listing (Part 1 of 8)

***** I N L I N E R E P O R T *****

IPA Inline Report (Summary)

Reason: P : #pragma noline 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	-	200 (102)	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

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 14. Example of an IPA Link Step Listing (Part 2 of 8)

```

***** PARTITION MAP *****

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(5)    *ARGPARSE    *CHARSET(BIAS=EBCDIC,LIB=EBCDIC)    *NOCOMPACT    *NOCOMPRESS
*NOCSECT    *NODLL    *ENV(MVS)    *EXECOPS    *FLOAT(HEX,FOLD,AFP)    *NOGOFF    *NOGONUMBER    *NOIGNERRNO    *ILP32
*NOINITAUTO    *IPA(LINK)    *LIBANSI    *LIST    *NOLOCALE    *LONGNAME    *MAXMEM(2097152)    *OPTIMIZE(2)    *PLIST(HOST)
*REDIR    *NORENT    *NOROCONST    *SPILL(128)    *START    *STRICT    *NOSTRICT_INDUCTION    *NOTEST    *TUNE(5)
*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

***** END OF PARTITION MAP *****

```

Figure 14. Example of an IPA Link Step Listing (Part 3 of 8)

```

OFFSET OBJECT CODE    LINE#    FILE#    PSEUDO    ASSEMBLY    LISTING

Timestamp and Version Information
000000    F2F0    F0F4                    =C'2004'    Compiled Year
000004    F0F2    F0F6                    =C'0206'    Compiled Date MMDD
000008    F1F1    F0F0    F5F2                    =C'110052'    Compiled Time HHMMSS
00000E    F0F1    F0F6    F0F0                    =C'010600'    Compiler Version

Timestamp and Version End

```

```

OFFSET OBJECT CODE    LINE#    FILE#    PSEUDO    ASSEMBLY    LISTING

```

```

EXTERNAL SYMBOL DICTIONARY

TYPE    ID    ADDR    LENGTH    NAME
SD      1    000000    000018    @STATICP
SD      2    000000    000004    gbl

```

Figure 14. Example of an IPA Link Step Listing (Part 4 of 8)

EXTERNAL SYMBOL CROSS REFERENCE

ORIGINAL NAME	EXTERNAL SYMBOL NAME
@STATICP	@STATICP
gb1	gb1

***** PARTITION MAP *****

PARTITION 1 OF 1

PARTITION SIZE:
Actual: 3624
Limit: 102400

PARTITION CSECT NAMES:
Code: none
Static: none
Test: none

PARTITION DESCRIPTION:
Primary partition

COMPILER OPTIONS FOR PARTITION 1:

*AGGRCOPY (NOOVERLAP)	*NOALIAS	*ARCH(5)	*ARGPARSE	*CHARSET(BIAS=EBCDIC, LIB=EBCDIC)	*NOCOMPACT	*NOCOMPRESS
*NOCSECT	*NODLL	*ENV(MVS)	*EXECOPS	*FLOAT(HEX, FOLD, AFP)	*NOGDIFF	*NOIGNERRNO
*NOINITAUTO	*IPA(LINK)	*LIBANSI	*LIST	*NOLocale	*LONGNAME	*MAXMEM(2097152)
*REDIR	*NORENT	*NOROCONST	*SPILL(128)	*START	*STRICT	*NOSTRICT_INDUCTION
*NOXPLINK	*XREF					*NOTEST
						*TUNE(5)

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

***** END OF PARTITION MAP *****

Figure 14. Example of an IPA Link Step Listing (Part 5 of 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

Timestamp and Version Information
000000 F2F0 F0F4          =C'2004'          Compiled Year
000004 F0F2 F0F6          =C'0206'          Compiled Date MMDD
000008 F1F1 F0F0 F5F2    =C'110052'        Compiled Time HHMMSS
00000E F0F1 F0F6 F0F0    =C'010600'        Compiler Version

Timestamp and Version End
    
```

```

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          000010 | 1  main  DS  0D
000018 47F0 F022 000010 | 1          B  34(,r15)
00001C 01C3C5C5          CEE eyecatcher
000020 00000098          DSA size
000024 00000098          =A(PPA1-main)
000028 47F0 F001 000010 | 1          B  1(,r15)
00002C 58F0 C31C 000010 | 1          L  r15,796(,r12)
000030 184E          000010 | 1          LR r4,r14
000032 05EF          000010 | 1          BALR r14,r15
000034 00000000          =F'0'
000038 07F3          000010 | 1          BR  r3
00003A 90E4 D00C 000010 | 1          STM r14,r4,12(r13)
00003E 58E0 D04C 000010 | 1          L  r14,76(,r13)
000042 4100 E098 000010 | 1          LA  r0,152(,r14)
000046 5500 C314 000010 | 1          CL  r0,788(,r12)
00004A 4130 F03A 000010 | 1          LA  r3,58(,r15)
00004E 4720 F014 000010 | 1          BH  20(,r15)
000052 5000 E04C 000010 | 1          ST  r0,76(,r14)
000056 9210 E000 000010 | 1          MVI 0(r14),16
00005A 50D0 E004 000010 | 1          ST  r13,4(,r14)
00005E 18DE          000010 | 1          LR  r13,r14
000060          End of Prolog

000060 180F          000005 | 3 +          LR  r0,r15
000062 1A00          000005 | 3 +          AR  r0,r0
000064 181F          000005 | 3 +          LR  r1,r15
000066 B252 0010 000005 | 3 +          MSR  r1,r0
00006A 180F          000005 | 3 +          LR  r0,r15
00006C B252 0001 000005 | 3 +          MSR  r0,r1
000070 181F          000005 | 3 +          LR  r1,r15
000072 B252 0010 000005 | 3 +          MSR  r1,r0
000076 180F          000005 | 3 +          LR  r0,r15
000078 B252 0001 000005 | 3 +          MSR  r0,r1
00007C 181F          000005 | 3 +          LR  r1,r15
00007E B252 0010 000005 | 3 +          MSR  r1,r0
000082 180F          000005 | 3 +          LR  r0,r15
000084 B252 0001 000005 | 3 +          MSR  r0,r1
000088 181F          000005 | 3 +          LR  r1,r15
00008A B252 0010 000005 | 3 +          MSR  r1,r0
00008E 180F          000005 | 3 +          LR  r0,r15
000090 B252 0001 000005 | 3 +          MSR  r0,r1
000094 B252 00F0 000005 | 3 +          MSR  r15,r0
000098          000022 | 1 @1L18 DS  0H

000098          Start of Epilog
000098 180D          000023 | 1          LR  r0,r13
00009A 58D0 D004 000023 | 1          L  r13,4(,r13)
00009E 58E0 D00C 000023 | 1          L  r14,12(,r13)
0000A2 9824 D01C 000023 | 1          LM  r2,r4,28(r13)
0000A6 051E          000023 | 1          BALR r1,r14
0000A8 0707          000023 | 1          NOPR 7

*** General purpose registers used: 110110000001111
*** Floating point registers used: 0000000000000000
*** Size of register spill area: 128(max) 0(used)
*** Size of dynamic storage: 152
    
```

Figure 14. Example of an IPA Link Step Listing (Part 6 of 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

*** Size of executable code: 146

0000AA 0000

0000AC 0000 0000

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	50000049	=F'1342177353'	CDL function length/2
0000D8	FFFFFF68	=F'-152'	CDL function EP offset
0000DC	38240000	=F'941883392'	CDL prolog
0000E0	40090040	=F'1074331712'	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(TIMESTAMP-PPA2)	
000100	0000 0000	=F'0'	No primary
000104	0000 0000	=F'0'	Flags

PPA2 End

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

Figure 14. Example of an IPA Link Step Listing (Part 7 of 8)

EXTERNAL SYMBOL CROSS REFERENCE

ORIGINAL NAME	EXTERNAL SYMBOL NAME
@STATICP	@STATICP
main	main
CEESG003	CEESG003
CEESTART	CEESTART
@PPA2	@PPA2
CEEMAIN	CEEMAIN
EDCINPL	EDCINPL

***** SOURCE FILE MAP *****

*ORIGIN	OBJECT FILE ID	SOURCE FILE ID	SOURCE FILE NAME
P	2	1	//'TSIPA.TEST.C(INCLMAIN)' - Compiled by 15694A01 1600 on 02/06/2004 11:00:52
P	3	2	//'TSIPA.TEST.C(INCLRTN1)' - Compiled by 15694A01 1600 on 02/06/2004 11:00:55
P	4	3	//'TSIPA.TEST.C(INCLRTN2)' - Compiled by 15694A01 1600 on 02/06/2004 11:00:58

ORIGIN: P=primary input PI=primary INCLUDE

***** END OF SOURCE FILE MAP *****

***** MESSAGE SUMMARY *****

TOTAL	UNRECOVERABLE (U)	SEVERE (S)	ERROR (E)	WARNING (W)	INFORMATIONAL (I)
0	0	0	0	0	0

***** END OF MESSAGE SUMMARY *****

***** END OF COMPILATION *****

Figure 14. Example of an IPA Link Step Listing (Part 8 of 8)

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, DEFINE). If you specify IPA suboptions that are irrelevant to the IPA Link step, the Prolog does not display them. Any problems with compiler options appear after the body of the Prolog section and before the End of Prolog section.

Object File Map

The Object File Map displays the names of the object files that were used as input to the IPA Link step. Specify any of the following options to generate the Object File Map:

- IPA(MAP)
- LIST

Other listing sections, such as the Source File Map, use the File ID numbers that appear in this listing section.

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 noline` 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 is 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 111, “IPA | NOIPA” on page 122, and “LIST | NOLIST” on page 143 for information about source and line numbers in the listing section.

External Symbol Dictionary

The External Symbol Dictionary lists the names that the IPA Link step generates for the current partition of the object module. It includes address information and size information about each symbol.

External Symbol Cross Reference

The IPA Link step generates this section if you specify the ATTR or XREF compiler option. It shows how the IPA Link step maps internal and ESD names for external symbols that are defined or referenced in the current partition of the object module.

Storage Offset Listing

The Storage Offset listing section displays the offsets for the data in the current partition of the object module.

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

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

If this is done and the compilation unit includes variables, the IPA Link step may generate a Storage Offset listing.

If you specify the ATTR or XREF option on the IPA Link step, and any of the compilation units that contributed variables to a particular partition had storage offset information encoded in the IPA object file, the IPA Link step generates a Storage Offset listing section for that partition.

The Storage Offset listing displays the variables that IPA did not coalesce. The symbol definition information appears as file#:line#.

Static Map

If you specify the ATTR or XREF option, the listing file includes offset information for file scope read/write static variables.

Messages

If the IPA Link step detects an error, or the possibility of an error, it issues one or more diagnostic messages, and generates the Messages listing section. This listing section contains a summary of the messages that are issued during IPA Link step processing.

The IPA Link step listing sorts the messages by severity. The Messages listing section displays the listing page number where each message was originally shown. It also displays the message text, and optionally, information relating the error to a file name, line (if known), and column (if known).

For more information on compiler messages, see “FLAG | NOFLAG” on page 104, 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 5. Binder options and control statements

This chapter lists the binder options, suboptions, and control statements that are considered important for a C or C++ programmer. For a detailed description of all the binder options and control statements, see *z/OS MVS Program Management: User's Guide and Reference*.

C or C++ programmers should be familiar with the following binder options and relevant suboptions:

- ALIASES
- AMODE
- CALL
- CASE
- COMPAT
- DYNAM
- LET
- LIST
- MAP
- OPTIONS
- REUS
- RMODE
- UPCASE
- XREF

C or C++ programmers should be familiar with the following control statements:

- AUTOCALL
- ENTRY
- IMPORT
- INCLUDE
- LIBRARY
- NAME
- RENAME

Chapter 6. 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 *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 *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 have matching compiler options. If you specify the compiler option, it takes 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.

You can specify multiple run-time options per directive or multiple directives per compilation unit. If you want to specify the ARGPARSE or REDIR options, the #pragma runopts directive must be in the same compilation unit as main(). Neither run-time option has an effect on programs invoked under the z/OS shell. This is because the shell program handles the parsing and redirection of command line arguments within that environment. Even though you can specify this directive in multiple compilation units, the specification that will take effect depends on the order of linking. It is advisable to specify it only once, and in the same compilation unit as main().

When you specify multiple instances of #pragma runopts in separate compilation units, the compiler generates a CSECT for each compilation unit that contains a #pragma runopts directive. When you link multiple compilation units that specify #pragma runopts, the linkage editor takes only the first CSECT, thereby ignoring your other option statements. Therefore, you should always specify your #pragma runopts directive in the same source file that contains the function main().

For more information on the #pragma runopts preprocessor directive, see *z/OS 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 7, “Compiling,” on page 291
- Chapter 8, “Using the IPA Link step with z/OS C/C++ programs,” on page 329
- Chapter 9, “Binding z/OS C/C++ programs,” on page 355
- Chapter 10, “Binder processing,” on page 383
- Chapter 11, “Running a C or C++ application,” on page 409

Chapter 7. 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 4, “Compiler Options,” on page 43.

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 compilations under z/OS batch, TSO, or the UNIX System Services environment

Note: As of z/OS V1R5 C/C++, the compiler will only work if both the SCEERUN and SCEERUN2 Language Environment libraries are available.

Input to the z/OS 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 more information about input for IPA, refer to Chapter 8, “Using the IPA Link step with z/OS C/C++ programs,” on page 329.

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
- CEE.SCEERUN2

The locations of the compiler and the run-time library were determined by the system programmer who installed the product. The compiler and library should be in the STEPLIB, JOBLIB, LPA, or LNKLST concatenations. LPA can be from either specific modules (IEALPAXx) or a list (LPALSTxx). See the cataloged procedures shipped with the product in Appendix D, “Cataloged procedures and REXX EXECs,” on page 591.

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, 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 files in an HFS directory

Secondary input

For a C or C++ program, secondary input to the compiler consists of data sets or directories that contain include files. Use the LSEARCH and SEARCH compiler options, or the SYSLIB DD statement when compiling in batch, to specify the location of the include files.

For more information on the use of these compiler options, see “LSEARCH | NOLSEARCH” on page 150 and “SEARCH | NOSEARCH” on page 184. For more information on naming include files, see “Specifying include file names” on page 317. For information on how the compiler searches for include files, see “Search sequences for include files” on page 324. For more information on include files, refer to “Using include files” on page 316.

Note: The LRECL for the SCLBH.H data set has changed from 80 to 120. You should ensure that SCLBH.H is the first data set in your SYSLIB concatenation. Do not use the SYSLIB concatenation to search for C++ header files with the compiler because searching the SYSLIB concatenation cannot distinguish between the old UNIX System Laboratories header files and new ISO Standard Library header files. For example, #include <iostream.h> (old USL) and #include <iostream> (ISO Standard) are indistinguishable using the SYSLIB concatenation. Use the SEARCH compiler option so that the correct header files are included.

Output from the compiler

You can specify compiler output files as one or more of the following:

- A sequential data set
- A member of a partitioned data set
- A partitioned data set
- A Hierarchical File System (HFS) file
- An HFS directory

For valid combinations of input file types and output file types, refer to Table 29 on page 295.

Specifying output files

You can use compile options to specify compilation output files as follows:

Table 27. Compile options that provide output file names

Output File Type	Compiler Option
Object Module	OBJECT(filename)
Listing File	SOURCE (filename), LIST(filename), INLRPT(filename) (Note: All listings must go to the same file. The last given location is used.)
Preprocessor Output	PPONLY(filename)
Events File	EVENTS(filename)
Template Output	TEMPINC(location)

Table 27. Compile options that provide output file names (continued)

Output File Type	Compiler Option
Template Registry	TEMPLATEREGISTRY(filename)

When compiler options that generate output files are specified without suboptions to identify the output files, and, in the case of a batch job, the designated ddnames are not allocated, the output file names are generated based on the name of the source file.

Note: The exception to this case is Template Registry, which is fixed to `templreg`, and Template Output, which is fixed to `tempinc`.

For data sets, the compiler generates a low-level qualifier by appending a suffix to the data set name of the source, as Table 28 shows.

If you compile source from HFS files without specifying output file names in the compiler options, the compiler writes the output files to the current working directory. The compiler does the following to generate the output file names:

- Appends a suffix, if it does not exist
- Replaces the suffix, if it exists

The following default suffixes are used:

Table 28. 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 a z/OS data set if the source resides in a data set.
2. If you have specified the 0E option, see “OE | NOOE” on page 165 for a description of the default naming convention.
3. If you supply inline source in your JCL, the compiler will not generate an output file name automatically. You can specify a file name either as a suboption for a compiler option, or on a ddname in your JCL.
4. If you are using #pragma options to specify a compile-time option that generates an output file, you must use a ddname to specify the output file name when compiling under batch. The compiler will not automatically generate file names for output that is created by #pragma options.

Example: Under TSO, the compiler generates the object file 'userid.TEST.SRC.OBJ' if you compile the following:

```
cc TEST.SRC (OBJ)
```

The compiler generates the object file 'userid.TEST.SRC.OBJ(HELLO)' if you compile the following:

```
cc 'h1qua1.TEST.SRC(HELLO)' (OBJ)
```

Listing output

Note: Although the compiler listing is for your use, it is not a programming interface and is subject to change.

To create a listing file that contains source, object, or inline reports use the SOURCE, LIST, or INLRPT compile options, respectively. The listing includes the results of the default or specified options of the CPARM parameter (that is, the diagnostic messages and the object code listing). If you specify *filename* with two or more of these compile options, the compiler combines the listings and writes them to the last file specified in the compile options. If you did not specify *filename*, the listing will go to the SYSCPRT DD name, if you allocated it. Otherwise, the compiler generates a default file name as described in “LIST | NOLIST” on page 143.

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 162.

Under UNIX System Services, an object name specified with -o will take priority over the file name specified with the OBJECT option.

Differences in object modules under IPA: The 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. 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” on page 295 for information about valid input/output file types.

Preprocessor output

If you specify *filename* with the PPOONLY compile option, the compiler writes the preprocessor output to that file. If you do not specify *filename* with the PPOONLY 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 “PPOONLY | NOPPOONLY” on page 175.

Template instantiation output

If you specify *location*, which is either an HFS file or a sequential file/PDS member, with the TEMPLATEREGISTRY compile option, the compiler writes the template registry to that location. If you do not specify *location* with the TEMPLATEREGISTRY option, the compiler determines a default destination for the template registry file. See “TEMPLATEREGISTRY | NOTEMPLATEREGISTRY” on page 205 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 203 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 29. 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	<ol style="list-style-type: none"> 1. If the file exists as a sequential data set, overwrites it 2. If the file does not exist, creates sequential data set 3. Otherwise compilation fails 	<ol style="list-style-type: none"> 1. If the PDS does not exist, creates PDS and member 2. If the PDS exists and member does not exist, adds member 3. If the PDS and member both exist, then overwrites the member 	<ol style="list-style-type: none"> 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)	<ol style="list-style-type: none"> 1. If the file exists as a sequential data set, overwrites it 2. If the file exists as a PDS, creates or overwrites member 3. If the file does not exist, creates PDS and member 	<ol style="list-style-type: none"> 1. If the PDS does not exist, creates PDS and member 2. If the PDS exists and member does not exist, adds member 3. If the PDS and member both exist, then overwrites the member 	<ol style="list-style-type: none"> 1. If the directory does not exist, compilation fails 2. If the directory exists and the file with the specified file name does not exist, creates file 3. If the directory exists and the file exists, overwrites file 	<ol style="list-style-type: none"> 1. If the directory does not exist, compilation fails 2. If the directory exists and the file with the file name <i>MEMBER.ext</i> does not exist, creates file 3. If the directory exists and the file with the file name <i>MEMBER.ext</i> also exists, overwrite file

Table 29. 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
All members of a PDS, for example A.B	<ol style="list-style-type: none"> 1. If the file exists as a PDS, creates or overwrites members 2. If the file does not exist, creates PDS and members 3. Otherwise compilation fails 	Not Supported	Not Supported	<ol style="list-style-type: none"> 1. If the directory does not exist, compilation fails 2. If the directory exists and the files with the file names <i>MEMBER.ext</i> do not exist, creates files 3. If the directory exists and the files with the file names <i>MEMBER.ext</i> exist, overwrites files
HFS file, for example /a/b/d.c	<ol style="list-style-type: none"> 1. If the file exists as a sequential data set, overwrites file 2. If the file does not exist, creates sequential data set 3. Otherwise compilation fails 	<ol style="list-style-type: none"> 1. If the PDS does not exist, creates the PDS and stores a member into the data set 2. If the PDS exists and member does not exist, then adds the member in the PDS 3. If the PDS and member both exist, then overwrites the member 	<ol style="list-style-type: none"> 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 	<ol style="list-style-type: none"> 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	<ol style="list-style-type: none"> 1. If the directory does not exist, compilation fails 2. If the directory exists and the files to be written do not exist, creates files 3. If the directory exists and the files to be written already exist, overwrites files

Compiling under z/OS batch

To compile your C/C++ source program under batch, you can either use cataloged procedures that IBM supplies, or write your own JCL statements.

Using cataloged procedures for z/OS C

You can use one of the following IBM-supplied cataloged procedures. Each procedure includes a compilation step to compile your program.

	EDCC	Compile a 31-bit or 64-bit program
	EDCCB	Compile and bind a 31-bit program
	EDXCXB	Compile and bind a 31-bit XPLINK C program
	EDCQCB	Compile and bind a 64-bit C program
	EDCCL	Compile and link-edit a 31-bit naturally re-entrant program
	EDCCBG	Compile, bind, and run a 31-bit program
	EDXCXBG	
		Compile, bind, and run a 31-bit XPLINK C program
	EDCQCBG	
		Compile, bind, and run a 64-bit C program
	EDCCLG	Compile, link-edit, and run a 31-bit program
	EDCCPLG	
		Compile, prelink, link-edit, and run a 31-bit program
	EDCCLIB	
		Compile and maintain an object library for a 31-bit or 64-bit application
	EDCI	Run the IPA Link step for a non-XPLINK 31-bit application
	EDCXI	Run the IPA Link step for an XPLINK compiled program or a 64-bit compiled program
	CCNPD1B	
		Bind a 31-bit C program that was compiled with IPA(PDF1)
	CCNXP1B	
		Bind a 31-bit XPLINK C program that was compiled with IPA(PDF1)
	CCNQPD1B	
		Bind a C 64-bit program that was compiled with IPA(PDF1)
	EDCQB	Bind a 64-bit C program
	EDCQBG	Bind, and run a 64-bit C program

IPA considerations

The EDCC procedure should be used for the IPA Compile step. Only the EDCI and EDCXI procedures apply to the IPA Link step.

To run the IPA Compile step, use the EDCC procedure, and ensure that you specify the IPA(NOLINK) or IPA compiler option. Note that you must also specify the LONGNAME compiler option or the #pragma longname directive.

To create an IPA-optimized object module, you must run the IPA Compile step for each source file in your program, and the IPA Link step once for the entire program. Once you have successfully created an IPA-optimized object module, you must bind it to create the final executable.

For further information on IPA, see Chapter 8, “Using the IPA Link step with z/OS C/C++ programs,” on page 329.

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.

	CBC	Compile a 31-bit or 64-bit program
	CBCB	Compile and bind a 31-bit non-XPLINK program
	CBCXB	Compile and bind a 31-bit XPLINK C++ program
	CBCQB	Compile and bind a 64-bit C++ program
	CBCCL	Compile, prelink, and link for a 31-bit non-XPLINK program
	CBCCBG	Compile, bind, and run a 31-bit non-XPLINK program
	CBCXCBG	Compile, bind, and run a 31-bit XPLINK C++ program
	CBCQCBG	Compile, bind, and run a 64-bit C++ program
	CBCCLG	Compile, prelink, link, and run a 31-bit non-XPLINK program
	CBCI	Run the IPA Link step for a 31-bit non-XPLINK program
	CBCXI	Run the IPA Link step for an XPLINK compiled 31-bit program or a 64-bit program
	CCNPD1B	Bind a C++ program that was compiled with IPA(PDF1)
	CCNQPD1B	Bind a C++ 64-bit program that was compiled with IPA(PDF1)
	CCNXPD1B	Bind an XPLINK C++ program that was compiled with IPA(PDF1)
	CBCB	Bind a 31-bit non-XPLINK application
	CBCXB	Bind a 31-bit XPLINK C++ program
	CBCQB	Bind a 64-bit C++ program
	CBCXBG	Bind and run a 31-bit XPLINK C++ program
	CBCQBG	Bind and run a 64-bit C++ program
	CBCLG	Prelink, link, and run a 31-bit non-XPLINK program
	CBCG	Run a 31-bit or 64-bit C++ program
	CBCXG	Run a 31-bit or a 64-bit C++ program

See Appendix D, “Cataloged procedures and REXX EXECs,” on page 591 for more information on cataloged procedures.

IPA considerations

The CBC procedure should be used for the IPA Compile step. Only the CBCI and CBCXI procedures apply to the IPA Link step.

To run the IPA Compile step, use the CBC procedure, and ensure that you specify the IPA(NOLINK) or IPA compiler option. Note that for C you must also specify the LONGNAME compiler option or the #pragma longname directive. For C++, you don't have to do this since C++ always uses LONGNAME. You should not specify the NOLONGNAME option.

To create an IPA-optimized object module, you must run the IPA Compile step for each source file in your program, and the IPA Link step once for the entire program. Once you have successfully created an IPA-optimized object module, you must bind it to create the final executable.

For further information on IPA, see Chapter 8, “Using the IPA Link step with z/OS C/C++ programs,” on page 329.

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 catalogued procedure, 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'
```

Examples of compiling programs using your own JCL

The following example shows sample JCL for compiling a 32-bit C program:

```
//jobname JOB acctno,name...
//COMPILE EXEC PGM=CCNDRVR,
// PARM='/SEARCH(''CEE.SCEEH.+'') NOOPT SO OBJ'
//STEPLIB DD DSN=CEE.SCEERUN,DISP=SHR
// DD DSN=CEE.SCEERUN2,DISP=SHR
// DD DSN=CBC.SCCNCMP,DISP=SHR
//SYSLIN DD DSN=MYID.MYPROG.OBJ(MEMBER),DISP=SHR
//SYSPRINT DD SYSOUT=*
//SYSIN DD DATA,DLM=@@
#include <stdio.h>
:
:
int main(void)
{
/* comment */
:
}
@@
//SYSUT1 DD DSN=...
:
:
/*
```

Figure 15. JCL for compiling a 32-bit C program (for NOOPT, SOURCE, and OBJ)

The following example shows sample JCL for compiling a 64-bit C program:

```

| //jobname JOB acctno,name...
| //COMPILE EXEC PGM=CCNDRVR,
| // PARM='/SEARCH(''CEE.SCEEH.+'') NOOPT SO LP64 OPTFILE(DD:CPATH) '
| //STEPLIB DD DSN=CEE.SCEERUN,DISP=SHR
| //          DD DSN=CEE.SCEERUN2,DISP=SHR
| //          DD DSN=CBC.SCCNCMP,DISP=SHR
| //SYSLIN DD DSN=MYID.MYPROG.OBJ(MEMBER),DISP=SHR
| //SYSPRINT DD SYSOUT=*
| //SYSIN DD DATA,DLM=@@
| #include <stdio.h> ...
| int main(void)
| {
|     /* comment */
|     :
| }
| @@
| //SYSUT1 DD DSN=...
| :
| /*

```

Figure 16. JCL for compiling a 64-bit C program (for NOOPT, SOURCE, and LP64)

The following example shows sample JCL for compiling a 32-bit C++ program:

```

| //jobname JOB acctno,name...
| //COMPILE EXEC PGM=CCNDRVR,
| // PARM='/CXX SEARCH(''CEE.SCEEH.+'',''CBC.SCLBH.+''),NOOPT,SO,OBJ'
| //STEPLIB DD DSN=CEE.SCEERUN,DISP=SHR
| //          DD DSN=CEE.SCEERUN2,DISP=SHR
| //          DD DSN=CBC.SCCNCMP,DISP=SHR
| //SYSLIN DD DSN=MYID.MYPROJ.OBJ,DISP=SHR
| //SYSPRINT DD SYSOUT=*
| //SYSIN DD DATA,DLM=@@
| #include <stdio.h>
| #include <iostream.h>
| :
| int main(void)
| {
|     // comment
|     :
| }
| @@
| //SYSUT1 DD DSN=...
| :
| /*

```

Figure 17. JCL for compiling a 32-bit C++ program (for NOOPT, SOURCE, and OBJ)

The following example shows sample JCL for compiling a 64-bit C++ program:

```

| //jobname JOB acctno,name...
| //COMPILE EXEC PGM=CCNDRVR,
| // PARM='/CXX SEARCH(''CEE.SCEEH.+'', ''CBC.SCLBH.+''),NOOPT,SO,LP64'
| //STEPLIB DD DSN=CEE.SCEERUN,DISP=SHR
| //          DD DSN=CEE.SCEERUN2,DISP=SHR
| //          DD DSN=CBC.SCCNCMP,DISP=SHR
| //SYSLIN DD DSN=MYID.MYPROJ.OBJ,DISP=SHR
| //SYSPRINT DD SYSOUT=*
| //SYSIN DD DATA,DLM=@@
| #include <stdio.>
| #include <iostream.h> ...
| int main(void)
| {
| // comment
|
|     :
| }
| @@
| //SYSUT1 DD DSN=...
|
|     :
| // *

```

Figure 18. JCL for compiling a 64-bit C++ program (for NOOPT, SOURCE, and LP64)

Specifying source files

For non-HFS files, use this format of the SYSIN DD statement:

```
//SYSIN DD DSN=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 DD statement:

```
//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 following form of the SYSIN DD statement:

```
//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 *z/OS Information Roadmap*.

Specifying include files

Example: Use the SEARCH option to specify system include files, and the LSEARCH option to specify your include files:

```
//C EXEC PGM=CCNDRVR,PARM='/CXX SEARCH(''CEE.SCEEH.+'', ''CBC.SCLBH.+'')
```

You can also use the SYSLIB and USERLIB DD statements (note that the SYSLIB DD statement has a different use if you are running the IPA Link step). To specify more than one library, concatenate multiple DD statements as follows:

```
//SYSLIB DD DSNAME=USERLIB,DISP=SHR
// DD DSNAME=DUPX,DISP=SHR
```

Note: If the concatenated data sets have different block sizes, either specify the data set with the largest block size first, or use the DCB=*dsname* subparameter on the first DD statement. For example:

```
//USERLIB DD DSNAME=TINYLIB,DISP=SHR,DCB=BIGLIB
// DD DSNAME=BIGLIB,DISP=SHR
```

where BIGLIB has the largest block size. For rules regarding concatenation of data sets in JCL, refer to *z/OS 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

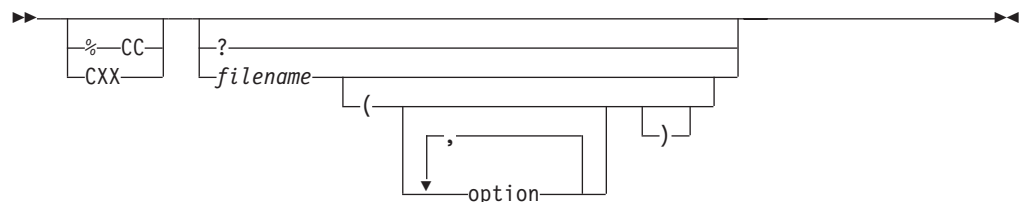
| You can invoke the z/OS C/C++ compiler under TSO by foreground execution from
 | TSO READY. This method of foreground execution calls the CC or CXX REXX
 | EXECs supplied by IBM.

Note: To run the compiler under TSO, you must have access to the run-time libraries. To ensure that you have access to the run-time library and compiler, do one of the following:

- Have your system programmer add the libraries to the LPALST or LPA
- Have your system programmer add the libraries to the LNKLIST
- 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

option is any valid compiler option

filename can be one of the following:

- A sequential data set
- A member of a partitioned data set
- All members of a partitioned data set
- A Hierarchical File System (HFS) file
- All files in an HFS 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 *z/OS C/C++ Programming Guide*.

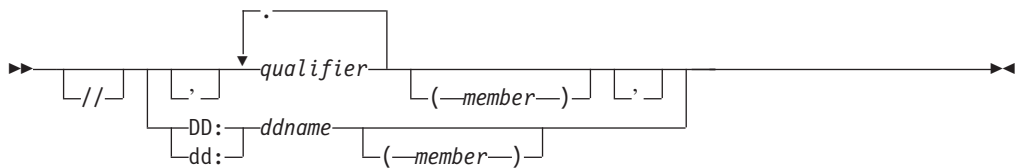
If you invoke either `CC` or `CXX` with no arguments or with only a single question mark, the appropriate preceding syntax diagram is displayed.

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 indicate 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 file name 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 (.

Example: The following example specifies, multiple compiler options with the sequential file STUDENT.GRADES.CXX:

```
CXX 'STUDENT.GRADES.CXX'
    ( LIST,TEST,
      LSEARCH(MASTER.STUDENT,COURSE.TEACHER),
      SEARCH(VGM9.FINANCE,SYSABC.REPORTS),
      OBJ('GRADUATE.GRADES.OBJ(REPORT)')
```

See “Summary of compiler options” on page 50 for more information on compiler options.

Compiling and binding in the UNIX System Services environment

z/OS UNIX System Services 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.

Both the SCEERUN and the SCEERUN2 libraries must be available when compiling in the UNIX System Services environment.

You can compile and bind application source code at one time, or compile the source and then bind at another time with other application source files or compiled objects.

As of z/OS V1R6, there are two utilities that enable you to invoke the compiler. The c89 utility enables compiler invocation using host environment variables and the xlc utility uses an external configuration file to control the invocation of the compiler. The following list highlights the differences between the xlc and c89 utilities:

- xlc utility uses the c89 utility to invoke the binder and the assembler and it has no direct interface to them
- xlc does not require that lp64 and xplink be explicitly specified as options on the command line for both the compile and the bind step; it uses _64 and _x command name suffixes to ensure 64-bit and XPLINK compiles and binds

- xlc utility supports -q options syntax as the primary method of specifying options on the command line
- xlc utility is unaffected by the value assigned to the STEPLIB environment variable in the UNIX Systems Services session; it obtains the STEPLIB from the configuration file
- xlc utility supports the same command names as the c89 utility (cc, c89, c++, and cxx), so the PATH environment variable must contain the path to the xlc "bin" directory ahead of the /bin directory if the xlc version of cc, c89, c++, and cxx is desired
- xlc utility does not support -WI for invoking IPA; it uses -O4 and -O5 or -qipa as the mechanism for invoking IPA

Note: For more information on the xlc utility, see Chapter 19, "xlc — Compiler invocation using a customizable configuration file," on page 513.

The c89 utility and xlc utility invoke the binder by default, unless the output file of the link-editing phase (-o option) is a PDS, in which case the prelinker is used.

For information on customizing your environment to compile and bind in the z/OS UNIX System Services environment, see "Environment variables" on page 486 or "Setting up a configuration file" on page 516.

Use the c89 utility or the xlc utility to compile and bind a C application program from the z/OS shell. The syntax is:

```
c89 [-options ...] [file.c ...] [file.a ...] [file.o ...] [-l libname]
```

where:

- options* are c89 or xlc options.
- file.c* is a source file. Note that C source files have a file extension of lowercase c.
- file.o* is an object file.
- file.a* is an archive file.
- libname* is an archive library.

The c89 and xlc utilities support IPA. For information on how to invoke the IPA Compile step using c89 or xlc, refer to "Invoking IPA using the c89 or xlc utilities" on page 310.

You can also use the cc command to compile a C application program from the z/OS shell. For more information, see Chapter 18, "c89 — Compiler invocation using host environment variables," on page 471 or the xlc command names described in Chapter 19, "xlc — Compiler invocation using a customizable configuration file," on page 513.

Use the c++ command to compile and bind a C++ application program from the z/OS shell. The syntax for c++ is:

```
c++ [-options ...] [file.C ...] [file.a ...] [file.o ...] [-l libname]
```

where:

- options* are C++ options.
- file.C* is a source file. Note that C++ files have a file extension of

uppercase C. The `_CXX_CXXSUFFIX` environment variable or `cxxsuffix` configuration file attribute can also be used to control which extensions are recognized as C++ file source extensions.

file.o is an object file.

file.a is an archive file.

libname is an archive library.

Another name for the `c++` command is `cxx`. The `cxx` command and the `c++` command are identical. You can use `cxx` instead of `c++` in all the examples that are shown in this section. If you are using the `x1c` utility, you can also use the `x1C` and the `x1c++` commands, which are identical to `c++` and `cxx`.

For a complete list of `c++` options, and for more information on `cxx`, see Chapter 18, “c89 — Compiler invocation using host environment variables,” on page 471 and Chapter 19, “x1c — Compiler invocation using a customizable configuration file,” on page 513.

Note: You can compile and bind application program source and objects from within the shell using the `c89` or `x1c` utilities. If you use one of these utilities, you must keep track of and maintain all the source and object files for the application program. You can use the `make` utility to maintain your z/OS UNIX System Services application source files and object files automatically when you update individual modules. The `make` utility will only compile files that have changed since the last `make` run.

For more information on using the `make` utility, see Chapter 16, “Archive and Make Utilities,” on page 465 and *z/OS UNIX System Services Programming Tools*.

Compiling without binding using compiler invocation command names supported by `c89` and `x1c`

To compile source files without binding them, enter one of the supported command names (for example, `c89` or `c++`) with the `-c` option to create object file output. Use the `-o` option to specify placement of the application program executable file to be generated. The placement of the intermediate object file output depends on the location of the source file:

- If the z/OS 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 file name 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 and x1c recognize that a file is a C source file by the .c suffix for HFS files, and the .C low-level qualifier for data sets. They recognize 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 32-bit object file usersource.o in your working HFS directory, specify:

```
c89 -c usersource.c
```

To compile z/OS C source to create the default 64-bit object file usersource.o in your working HFS directory, specify the following using the c89 utility:

```
c89 -c -Wc,lp64 usersource.c
```

The following shows the same example using the x1c utility:

```
c89_64 -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

c89 and xlc recognize that a file is a C++ source file by the .C suffix for HFS files, and the .CXX low-level qualifier for data sets. They recognize 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 32-bit object file usersource.o in your working HFS directory, specify the following:

```
c++ -c usersource.C
```

To compile z/OS C++ source to create the default 64-bit object file usersource.o in your working HFS directory, using the c89 utility specify:

```
c++ -c -Wc,lp64 usersource.C
```

The following shows the same example using the xlc utility:

```
c++_64 usersource.C
```

To compile z/OS C++ source to create an object file as a member in the PDS 'JONATHAN.APPROG.OBJ', specify:

```
c++ -c "'/'jonathan.approg.CXX(usersrc)'"
```

z/OS C++ Note:

To use the TSO utility 0GET 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 if you are using the c89 utility:

```
c++ -cV mypgm.C | awk '/^[^$]/ {print} /^$/ {printf "%s \n", $0}'  
> mypgm.lst
```

The following shows the same example using the x1c utility:

```
x1C -c -qsource mypgm.C | awk '/^[^$]/ {print} /^[^$]/
{printf "%s \n", $0}' > mypgm.lst
```

Compiling and binding application source to produce an application executable file

To compile an application source file to create the 32-bit object file usersource.o in the 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 an application source file, to create the 64-bit object file usersource.o in the HFS working directory and the executable file mymod.out in the /app/bin directory, specify the following using the c89 utility

```
c89 -Wc,lp64 -Wl,lp64 -o /app/bin/mymod.out usersource.c
```

The following shows the same example using the x1c utility:

```
c89_64 -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/myapps/bin/mainbal.out, specify:

```
c89 -o /u/claudio/myapps/bin/mainbal.out "'/claudio.pgms.C(MAINBAL)'"
```

Compiling and binding in one step using compiler invocation command names supported by c89 and x1c

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 commands are identical. You can use cxx instead of c++ in all the examples that are shown in this section. If you are using the x1c utility, you can also use the x1C and x1c++ commands, which are identical to c++ and cxx.

The c89 utility and x1c utility invoke the binder by default, unless the output file of the link-editing phase (-o option) is a PDS, in which case the prelinker is used.

- To compile and bind an application source file to create the 32-bit default executable file a.out in the HFS working directory, specify:

```
c89 usersource.c
c++ usersource.C
```

- To compile and bind an application source file to create the 64-bit default executable file a.out in the HFS working directory, specify:

```
c89 -Wc,lp64 -Wl,lp64 usersource.c
c++ -Wc,lp64 -Wl,lp64 usersource.C
x1C_64 usersource.C
```

- To compile and bind an application source file to create the mymod.out executable file in your /app/bin directory, specify:

```
c89 -o /app/bin/mymod.out usersource.c
c++ -o /app/bin/mymod.out usersource.C
```

- To compile and bind several application source files to create the mymod.out executable file in your /app/bin directory, specify:

```
c89 -o /app/bin/mymod.out usrsrc.c otsrc.c "'/'MUSR.C(PWAPP)'"
c++ -o /app/bin/mymod.out usrsrc.C otsrc.C "'/'MUSR.C(PWAPP)'"
```

- To compile and bind an application source file to create the MYLOADMD member of your 'APPROG.LIB' PDS, specify:

```
c89 -o "'/'APPROG.LIB(MYLOADMD)'" usersource.c
c++ -o "'/'APPROG.LIB(MYLOADMD)'" usersource.C
```

- To compile and bind an application source file with several previously compiled object files to create the executable file zinfo in your /prg/lib 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 barry1.c > barry1.lst
c++ -V barry1.C > barry1.lst
```

Note: -V does not cause all listings to be emitted when you invoke the compiler using x1c. Use, for example, -qsource or -qlist instead.

Building an application with XPLINK using the c89 or x1c utilities

To build an application with XPLINK using the c89 utility you must specify the XPLINK compiler option (i.e., -Wc,xplink) and the XPLINK binder option (i.e., -Wl,xplink). The binder option is not actually passed to the binder. It is used by c89 to set up the appropriate link data sets.

To build an application with XPLINK using the x1c utility, you do not have to explicitly specify the xplink option on the command line for either the compile or the bind step. x1c uses the _x command name suffix to ensure XPLINK compiles and binds.

Building a 64-bit application using the c89 or x1c utilities

To build a 64-bit application using the c89 utility, you must use the LP64 compiler option (i.e., -Wc,lp64) and the LP64 binder option (i.e., -Wl,lp64). The binder option is not actually passed to the binder. It is used by c89 to set up the appropriate link data sets.

To build a 64-bit application using the x1c utility, you do not have to explicitly specify the lp64 option on the command line for either the compile or the bind step. x1c uses the _64 command name suffix to ensure 64-bit compiles and binds.

Invoking IPA using the c89 or x1c utilities

You can invoke the IPA Compile step, the IPA Link step, or both using the c89 or x1c utilities. The step that you invoke depends upon the invocation parameters and type of files specified. To invoke IPA using c89, you must specify the I phase indicator along with the W option of the c89 utility. You can specify IPA suboptions as comma-separated keywords. To invoke IPA using x1c, you must use the -qip, -04, or -05 options. You can specify IPA suboptions as colon-separated keywords.

If you invoke the `c89` utility or `xlc` utility by specifying the `-c` compiler option and at least one source file, `c89` or `xlc` automatically specifies `IPA(NOLINK)` and automatically invokes the IPA Compile step. For example, the following `c89` command invokes the IPA Compile step for the source file `hello.c`:

```
c89 -c -WI,noobject hello.c
```

The following `xlc` command invokes the IPA Compile step for the source file `hello.c`:

```
xlc -c -qipa=noobject hello.c
```

If you invoke `c89` or `xlc` with at least one source file for compilation and any number of object files, and do not specify the `-c` option, `c89` or `xlc` invokes the IPA Compile step once for each compilation unit. It then invokes the IPA Link step once for the entire program, and then invokes the binder.

Example: The following `c89` command invokes the IPA Compile step, the IPA Link step, and the bind step while creating program `foo`:

```
c89 -o foo -WI,object foo.c
```

The following shows the same example using the `xlc` utility:

```
xlc -o foo -qipa=object foo.c
```

Refer to Chapter 18, “`c89` — Compiler invocation using host environment variables,” on page 471 for more information about the `c89` utility or Chapter 19, “`xlc` — Compiler invocation using a customizable configuration file,” on page 513 for more information about the `xlc` utility.

Specifying options for the IPA Compile step

You can pass options to the IPA Compile step, as follows:

- You can pass IPA compiler option suboptions by specifying `-WI`, for `c89` or `-qipa=` for `xlc`, followed by the suboptions.
- You can pass compiler options by specifying `-Wc`, for `c89` or `-q` for `xlc`, followed by the options.

Using IPA(OBJONLY) with the `c89` or `xlc` utilities

A compilation using `IPA(OBJONLY)` is simply a standard non-IPA compilation with this option added. Do not use the `-WI` flag with `c89` or `-qipa` with `xlc`, as this would convert the compilation into an IPA Compile step.

Example: The following `c89` command results in an `OPT(2)` `IPA(OBJONLY)` compilation for the source file `hello.c`:

```
c89 -c -Wc,ipa\objonly\ -2 hello.c
```

The following shows the same example using the `xlc` utility:

```
xlc -c -Wc,ipa\objonly\ -02 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.

Example: To create `myapp1` you compile and bind two source parts `mymain.c` and `mysub.c`. This dependency is captured in makefile `/u/jake/myapp1/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 *z/OS UNIX System Services Command Reference* for a description of the make utility. For a detailed discussion on how to create and use makefiles to manage application parts, see *z/OS UNIX System Services Programming Tools*.

Compiling with IPA

If you request Interprocedural Analysis (IPA) through the IPA compiler option, the compilation process changes significantly. IPA instructs the compiler to optimize your z/OS 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 *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 compilation process are noted throughout this chapter.

Figure 19 shows the flow of processing for a regular compilation:

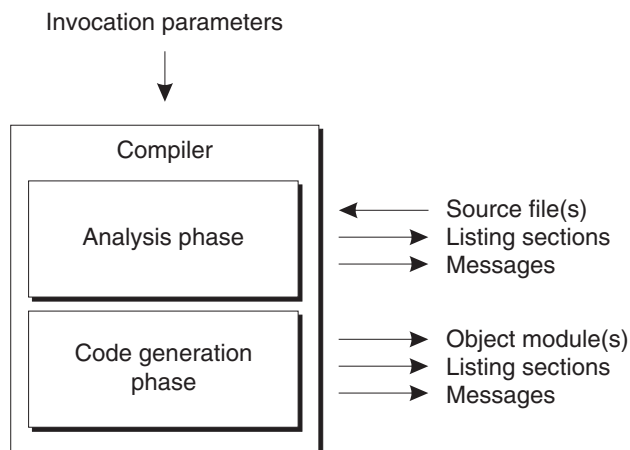


Figure 19. Flow of regular compiler processing

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 20 shows the flow of IPA Compile step processing.

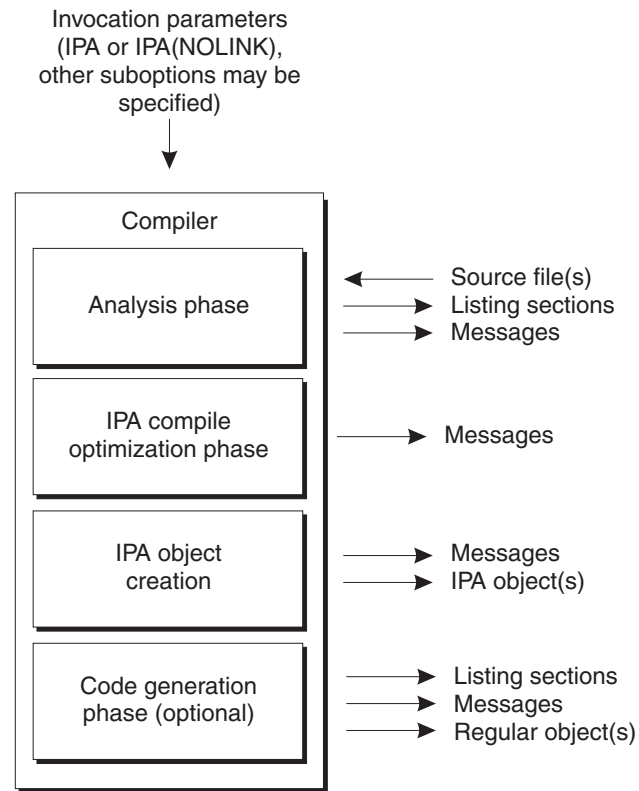


Figure 20. IPA Compile step processing

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 21 on page 314 shows the flow of IPA Link step processing:

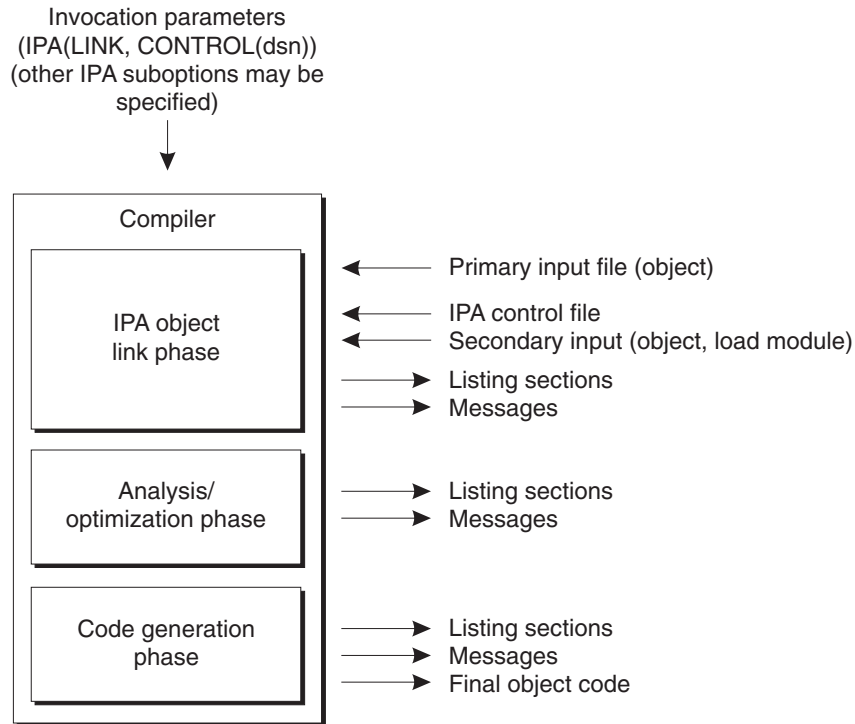


Figure 21. IPA Link step processing

Only c89, c++ and z/OS batch support the IPA Link step. Refer to Chapter 8, “Using the IPA Link step with z/OS C/C++ programs,” on page 329 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 22 on page 315 shows the flow of processing for an IPA(OBJONLY) compilation.

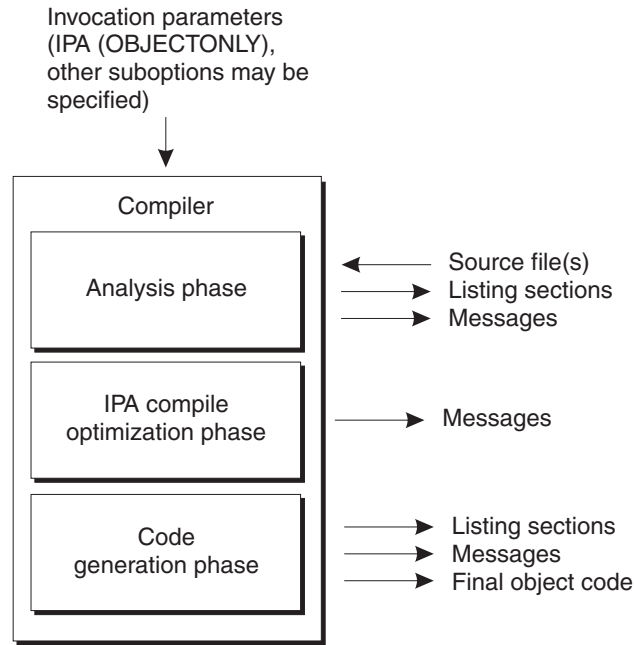


Figure 22. Compiling with IPA(OBJECTONLY)

Working with object files

z/OS object files are composed of a stream of 80 byte records. These may be binary object records, or link control statements. It is useful to be able to browse the contents of an object file, so that some basic information can be determined.

Browsing object files

Object files, which are sequential data sets or are members of a PDS or PDSE object library, can be browsed directly using the Program Development Facility (PDF) edit and browse options.

Object files, which are 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 "@@IPA0BJ". 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 "@@IPA0BJ". The IPA information will be separated from the "real" code and data by a delimiter END record with the comment "of IPA object". After the real code and data, there will be a second delimiter END record with the comment "of object".
4. IPA(OBJONLY)
Option text has "IPA (OBJONLY)".

Using feature test macros

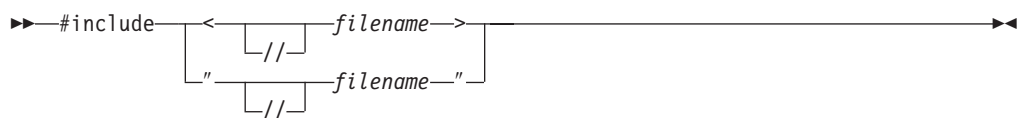
The compiler predefines feature test macros when certain features are available. For example, the `_LONG_LONG` macro is predefined if the compiler supports the long long data type. (Please refer to *z/OS C/C++ Language Reference* for a list of the feature macros).

Using include files

The `#include` preprocessor directive allows you to retrieve source statements from secondary input files and incorporate them into your C/C++ program.

z/OS C/C++ Language Reference describes the `#include` directive. Its syntax is:

```
▶▶ #include <filename> | "filename" | //filename //
```



The angle brackets specify system include files, and double quotation marks specify user include files.

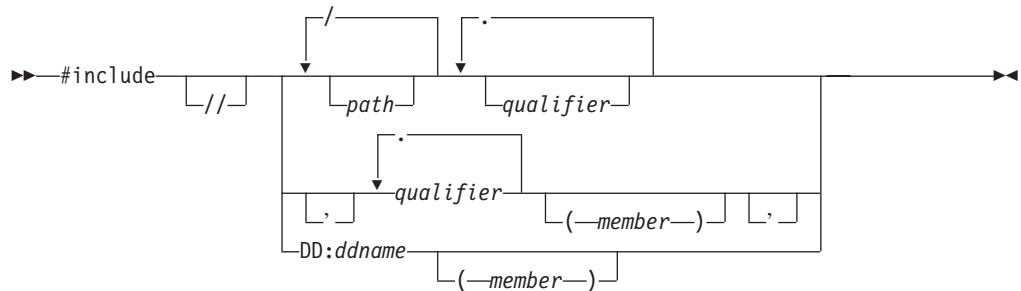
When you use the `#include` directive, you must be aware of the following:

- The *library search sequence*, the search order that C/C++ uses to locate the file. See "Search sequences for include files" on page 324 for more information on the library search sequence.
- The file-naming conversions that the C/C++ compiler performs.
- The area of the input record that contains sequence numbers when you are including files with different record formats. See *z/OS 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 150 and “SEARCH | NOSEARCH” on page 184.

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.

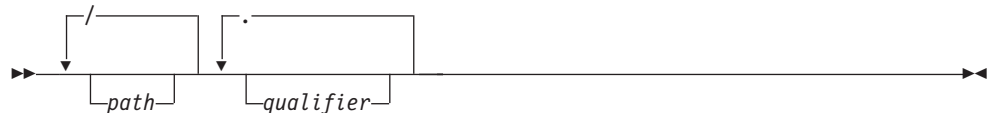
Note:

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 file names are specified by putting a leading slash (/) as the first character in the file name.
4. ddnames are always considered absolute.

Forming file names

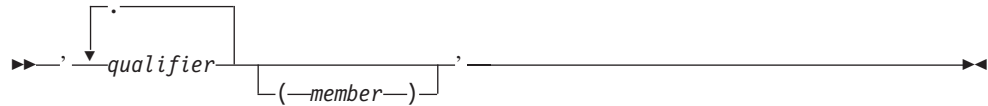
Refer to “Determining whether the file name is in absolute form” on page 321 for information on absolute file names. When the compiler performs a library search, it treats *filename* as either an HFS file name or a data set name. This depends on whether the library being searched is HFS or MVS. If the 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:



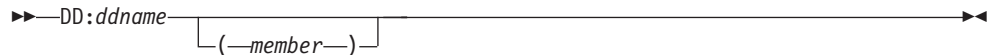
The compiler:

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 the '@' character (hex 7c)
- For the second DSN format:



The compiler:

1. Uppercases the *qualifier* and *member*
 2. Converts the underscore character (which is invalid for a DSN) to the '@' character (hex 7c)
- For the third DSN format:



The compiler:

1. Uppercases the DD:, *ddname*, and *member*
2. Converts the underscore character (which is invalid for a DSN) to the '@' character (hex 7c)

Forming data set names with LSEARCH | SEARCH options

When the *filename* specified in the `#include` directive is not in absolute form, the compiler combines it with different types of libraries to form complete data set specifications. These libraries may be specified by the LSEARCH or SEARCH compiler options. When the LSEARCH or SEARCH option indicates a data set then depending on whether it is a ddname, sequential data set, or PDS, different parts of *filename* are used to form the ddname or data set name.

Forming DDname

Example: The leftmost qualifier of the *filename* in the `#include` directive is used when the *filename* is to be a ddname:

Invocation:

```
SEARCH(DD:SYSLIB)
```

Include directive:

```
#include "sys/afile.g.h"
```

Resulting ddname:

```
DD:SYSLIB(AFILE)
```

In 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

Example: You specify libraries in the SEARCH | LSEARCH options as sequential data sets by using a trailing period followed by an asterisk (`.*`), or by a single asterisk (`*`). See “Specifying sequential data sets and PDSs” on page 153 to understand how to specify sequential data sets. All *qualifiers* and periods (`.`) in *filename* are used for sequential data set specification.

Invocation:

```
SEARCH(AA.*)
```

Include directive:

```
#include "sys/afile.g.h"
```

Resulting fully qualified data set name:

```
userid.AA.AFIL.E.G.H
```


Forming PDS name with LSEARCH | SEARCH + specification

Example: To specify libraries in the SEARCH and LSEARCH options as PDSs, use a period that is followed by a plus sign (.+), or a single plus sign (+). See “Specifying sequential data sets and PDSs” on page 153 to understand how PDSs are specified. When this is the case then all the *paths*, slashes (replaced by periods), and any *qualifiers* following the leftmost *qualifier* of the *filename* are appended to form the data set name. The leftmost *qualifier* is then used as the member name.

Invocation:

```
SEARCH('AA.+')
```

Include directive:

```
#include "sys/afile.g.h"
```

Resulting fully qualified data set name:

```
AA.SYS.G.H(AFILE)
```

and

Invocation:

```
SEARCH('AA.+')
```

Include directive:

```
#include "sys/bfile"
```

Resulting fully qualified data set name:

```
AA.SYS(BFILE)
```

Forming PDS with LSEARCH | SEARCH Options with No +

Example: When the LSEARCH or SEARCH option specifies a library but it neither ends with an asterisk (*) nor a plus sign (+), it is treated as a PDS. The leftmost qualifier of the *filename* in the `#include` directive is used as the member name.

Invocation:

```
SEARCH('AA')
```

Include directive:

```
#include "sys/afile.g.h"
```

Resulting fully qualified data set name:

```
AA(AFILE)
```

Examples of forming data set names

The following table gives the original format of the *filename* and the resulting converted name when you specify the NOOE option:

Table 30. Include filename conversions when NOOE is specified

#include Directive	Converted Name
Example 1. This <i>filename</i> 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 <i>filename</i> 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 <i>filename</i> is in absolute form. SEARCH is ignored.	
#include <'COMIC/BOOK.OLDIES.K'>	COMIC/BOOK.OLDIES.K
Example 3.	

Table 30. Include filename conversions when NOOE is specified (continued)

#include Directive	Converted Name
SEARCH(LIB1.*,LIB2.+,LIB3) #include "sys/abc/xx"	<ul style="list-style-type: none"> • first <i>opt</i> in SEARCH SEQUENTIAL FILE = <i>userid</i>.LIB1.XX • second <i>opt</i> in SEARCH PDS = <i>userid</i>.LIB2.SYS.ABC(XX) • third <i>opt</i> in SEARCH PDS = <i>userid</i>.LIB3(XX)
Example 4.	
SEARCH(LIB1.*,LIB2.+,LIB3) #include "Sys/ABC/xx.x"	<ul style="list-style-type: none"> • first <i>opt</i> in SEARCH SEQUENTIAL FILE = <i>userid</i>.LIB1.XX.X • second <i>opt</i> in SEARCH PDS = <i>userid</i>.LIB2.SYS.ABC.X(XX) • third <i>opt</i> in SEARCH PDS = <i>userid</i>.LIB3(XX)
Example 5.	
SEARCH(LIB1.*,LIB2.+,LIB3) #include <sys/name_1>	<ul style="list-style-type: none"> • first <i>opt</i> in SEARCH SEQUENTIAL FILE = <i>userid</i>.LIB1.NAME@1 • second <i>opt</i> in SEARCH PDS = <i>userid</i>.SYS(NAME@1) • third <i>opt</i> in SEARCH PDS = <i>userid</i>.LIB3(NAME@1)
Example 6.	
SEARCH(LIB1.*,LIB2.+,LIB3) #include <Name2/App1.App2.H>	<ul style="list-style-type: none"> • first <i>opt</i> in SEARCH SEQUENTIAL FILE = <i>userid</i>.LIB1.APP1.APP2.H • second <i>opt</i> in SEARCH PDS = <i>userid</i>.LIB2.NAME2.APP2.H(APP1) • third <i>opt</i> in SEARCH PDS = <i>userid</i>.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 <i>filename</i> 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:

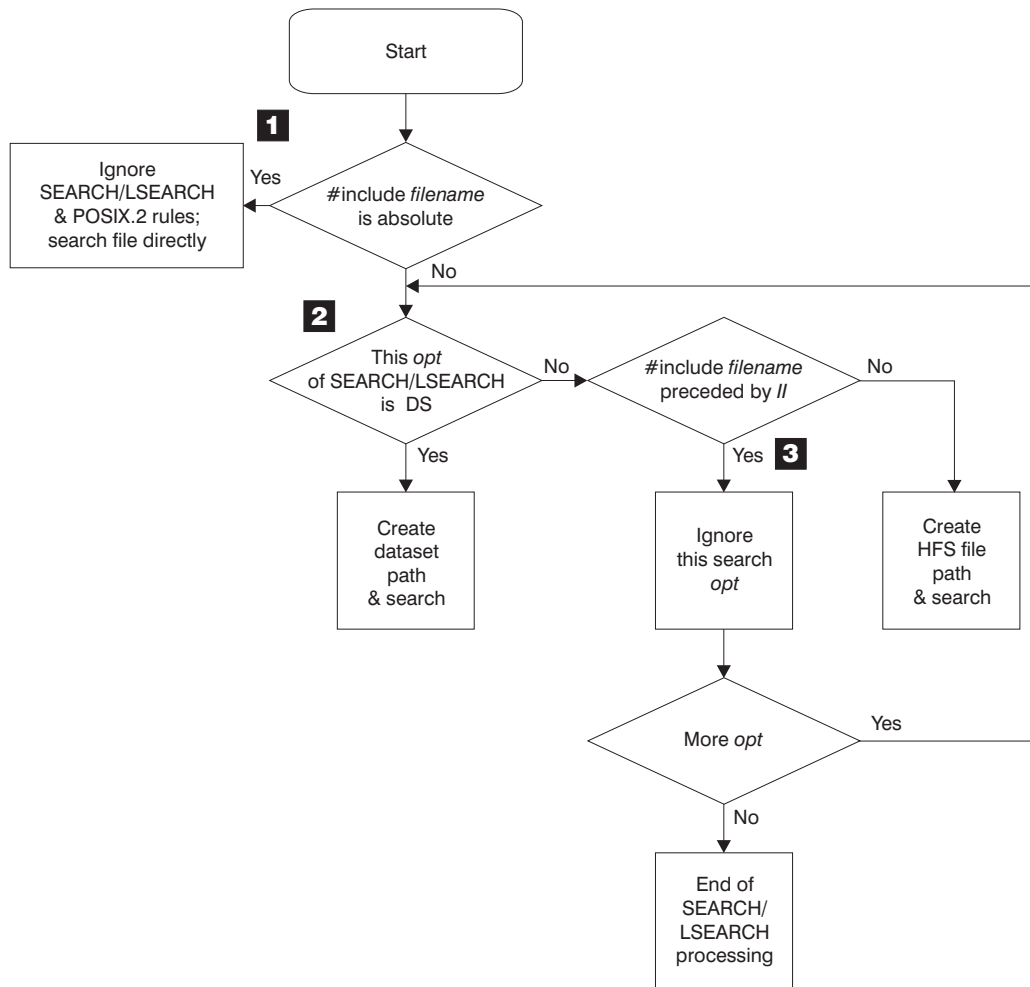


Figure 23. Overview of include file searching

- 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 24 on page 322 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 25 on page 323.
- 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:

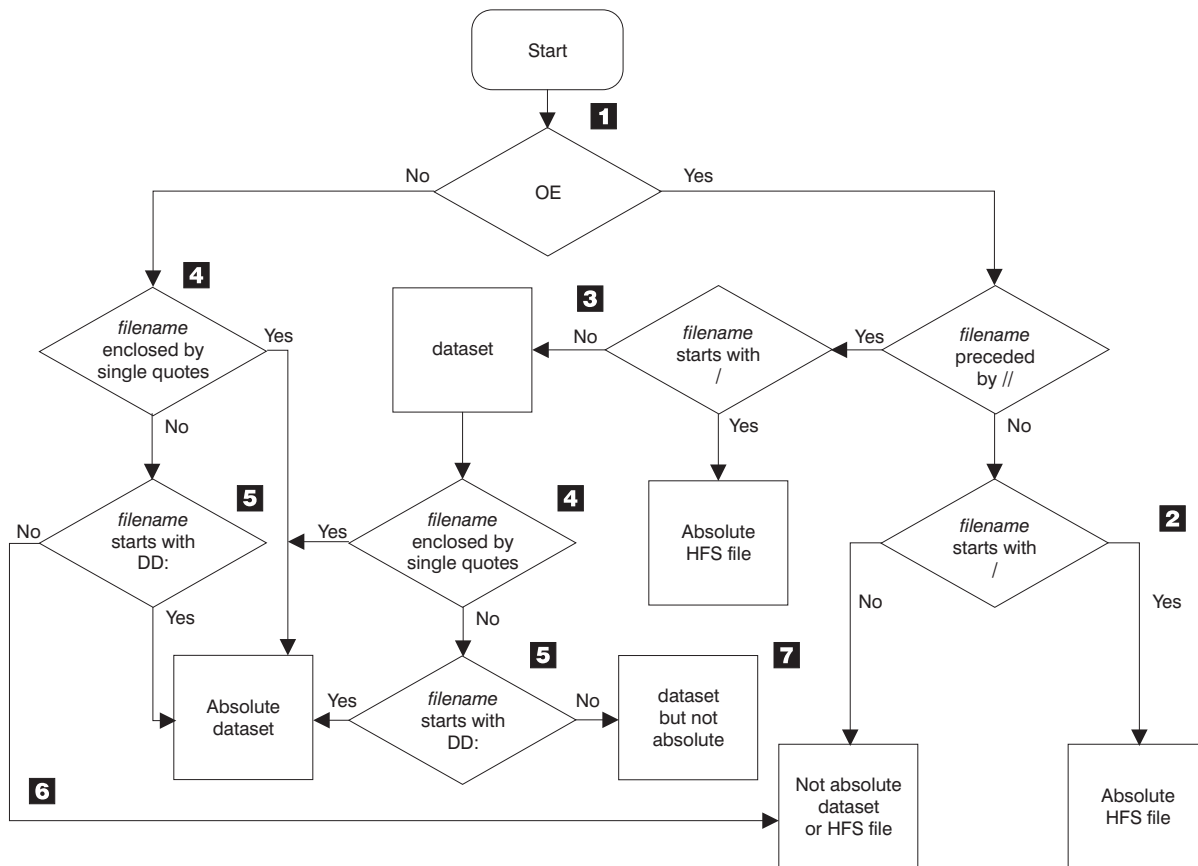


Figure 24. Testing if filename is in absolute form

- 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. If there are any invalid characters in *filename*, the compiler converts the invalid characters to at signs (@, hex 7c).
- 5** If you used the ddname format of the #include directive, the compiler uses the file associated with the ddname and directly opens the file as a data set. The libraries that are specified in the LSEARCH or SEARCH options are ignored.
- 6** If none of the 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 *opts* 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 *opts* in SEARCH are used to find system include files and the *opts* 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:

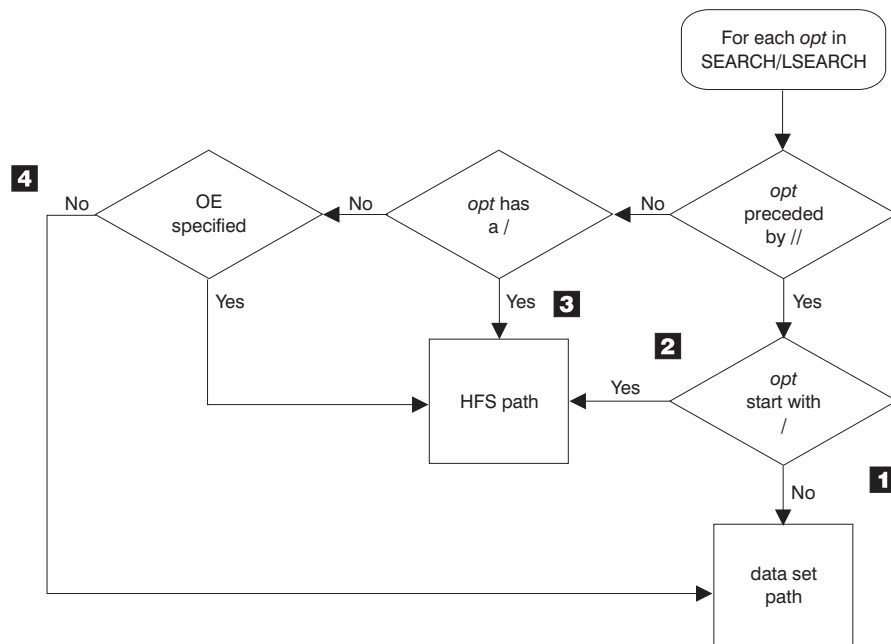


Figure 25. Determining if the SEARCH/LSEARCH *opt* is an HFS path

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

Example: When combining the library with the file name specified on the `#include` directive, it is the form of the library that determines how the include file name is to be transformed:

Options specified:

```
NOOE LSEARCH(Z, /u/myincs, (*.h)=(LIB(mac1)))
```

Include Directive:

```
#include "apath/afile.h"
```

Resulting fully qualified include names:

1. *userid.Z(AFILE)* (Z is non-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)

Example: 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 (*/*).

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 150 and “`SEARCH | NOSEARCH`” on page 184 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 321 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

Example: 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 31. 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 321 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

Example: The following shows an example where you are given a file /r/you/cproc.c that contains the following #include directives:

```
#include "/u/usr/header1.h"
#include "//aa/bb/header2.x"
#include "common/header3.h"
#include <header4.h>
```

And the following options:

```
OE(/u/crossi/myincs/cproc)
SEARCH(/V.+ , /new/inc1, /new/inc2)
LSEARCH(/(*.x)=(lib(AAA)), /c/c1, /c/c2)
```

The include files would be searched as follows:

Table 32. Examples of search order for z/OS UNIX System Services

#include Directive	Filename	Files in Search Order
Example 1. This is an absolute pathname, so no search is performed.		
#include	"/u/usr/header1.h"	1. /u/usr/header.h
Example 2. This is a data set (starts with //) and is treated as such.		
"//aa/bb/header2.x"		1. <i>userid</i> .AAA(HEADER2) 2. DD:USERLIB(HEADER2) 3. <i>userid</i> .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. <i>userid</i> .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. <i>userid</i> .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: ⁴

- 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` utility.

IBM supplies this option as input to the Installation and Customization of the compiler. Your system programmer can modify it as required for your installation.

The cataloged procedures, REXX EXECs, and panels that are supplied by IBM for C specify the following data sets for the SYSLIB ddname by default:

- CEE.SCEEH.H (standard header files)
- CEE.SCEEH.SYS.H (standard system header files)

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++:⁴

- 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/cbclib/include`. To specify that the compiler search these directories, code the option:

```
SEARCH(/usr/include/,/usr/lpp/cbclib/include/)
```

This option is the default for the `cxz` z/OS UNIX System Services command.

IBM supplies this option as input to the installation and customization of the compiler. Your system programmer can modify it as required for your installation.

4. The high-level qualifier may be different at your installation.

Chapter 8. Using the IPA Link step with z/OS C/C++ programs

Traditional optimizers only have the ability to optimize within a function (intra-procedural optimization) or at most within a compile unit (a single source file and its corresponding header files). This is because traditional optimizers are only given one compile unit at a time.

Interprocedural optimizations are a class of optimizations that operate across function boundaries. IBM's Interprocedural Analysis (IPA) optimizer is designed to optimize complete modules at a time. This allows for increased optimization. By seeing more of the application at once, IPA is able to find more opportunities for optimization and this can result in much faster code.

In order to get a global module view of the application, IPA uses the following two pass process:

- The first pass is called an IPA compile. During this pass, IPA collects all of the relevant information about the compile unit and stores it in the object file. This collected information is referred to as an IPA Object. The user can optionally request that both an IPA object and a traditional object are created from an IPA compile.
- The second pass is called the IPA Link. During this step, IPA acts like a traditional linker, and all object files, object libraries and side decks are fed to IPA so that it can completely optimize the module. The IPA Link step involves two separate optimizers. The IPA optimizer is run first and focuses optimizations across the module. IPA then breaks down the module into logical chunks called partitions and invokes the traditional optimizer with these partitions.

Whenever a compiler attempts to perform more optimizations, or looks at a larger portion of an application, more time, and more memory are required. Since IPA does both more optimizations than either OPT(2) or OPT(3) and has a global view of the module, the compile time and memory used by the IPA compile/link process is more than that used by a traditional OPT(2) or OPT(3) compilation.

The first two sections of this chapter provide several examples on how to create modules (with a main) or DLLs using IPA. The third section discusses the Profile-Directed Feedback option that can be used with IPA to get even more performance benefits. The fourth section gives some reference information on IPA-specific subjects, like the IPA control file. The final section of this chapter provides some hints and tips for troubleshooting situations that come up when compiling and debugging IPA applications. All example source can be found in the sample data set SCCNSAM. The names of the sample data set members are given in each example below.

Creating a module with IPA

This section describes creating a module that contains the function main.

Example 1. all C parts

The simplest case for IPA is an application that does not import any information from a DLL, and that is all in a single language that supports IPA. The following example covers this case. The sample programs mentioned here can be found in the sample data set with the member names given here.

The first example shows a simple application that is made up of three source files. The target is to compile it with IPA(Leve1 (2)) and OPT(2). We also want a full inline report and pseudo-assembly listing. This is the only example where the full source will be shown in this chapter.

CCNGHI1.C

```
hello1.c:
    int seen_main;
    int seen_unused3;

    char *string1 = "Hello";
    char *stringU1 = "I'm not going to use this one!";

    int func2( char *);

    int main (void) {

        seen_main++;
        func2(string1);

        return 0;
    }

    float unused3( int a ) {
        seen_unused3++;
        return (float) a+seen_unused3;
    }
```

Figure 26. *hello1.c* example source code

CCNGHI2.C

```
hello2.c:
    #include <stdio.h>

    int seen_func2;
    int seen_unused2;
    char *string2 = "world!";

    int func3 (char *);

    int func2( char * s1) {

        seen_func2++;

        printf("%s ",s1);

        return func3(string2);
    }

    double unused2(float x) {

        seen_unused2++;

        return x+ seen_unused2;
    }
```

Figure 27. *hello2.c* example source code

CCNGHI3.C

```
hello3.c:
#include <stdio.h>

int seen_func3;
int seen_unused1;

int unused1(int x) {
    seen_unused1++;
    return x+ seen_unused1;
}

int func3( char * string2) {
    seen_func3++;
    printf("%s\n",string2);
    return seen_func3;
}
```

Figure 28. hello3.c example source code

Building example 1. under UNIX System Services (USS)

For this example, the following table shows the mapping of SCCNSAM data set members to given file names:

SCCNSAM member name	Name used in this example
CCNGHI1	hello1.c
CCNGHI2	hello2.c
CCNGHI3	hello3.c

The following commands can be used to create this module under USS:

```
c89 -c -2 -WI,NOOBJECT,LIST hello1.c hello2.c hello3.c
c89 -2 -WI,MAP,LEVEL\{2\} -Wl,I,INLRPT,LIST\{hello.lst\} -o hello hello1.o
hello2.o hello3.o
```

The first c89 command performs an IPA compile on hello1.c, hello2.c, and hello3.c. The options after -WI are IPA suboptions, which are described below (for further information on these suboptions, see “IPA | NOIPA” on page 122 in the Chapter 4, “Compiler Options,” on page 43 chapter in this document):

NOOBJECT

This compile is doing an IPA compile (since -c was specified). This option specifies that only IPA objects should be generated by the IPA compile step. The optional traditional object should not be generated. The NOOBJECT option should be used unless the traditional object is needed for debugging purposes. NOOBJECT significantly shortens the overall compile time.

LIST This option tells IPA to save enough information that a listing with source file and line number information can be generated during the IPA(LINK) phase.

Note: -2 was specified on the IPA compile step. While it is not strictly necessary, it does allow for faster code to be generated in some cases.

The second c89 command does the IPA Link processing. Since -WI and W1,I were specified with .o files, c89 automatically turns on the LINK suboption of IPA. IPA(LINK), or -WI,LINK, does not need to be specified when compiling with c89. The -WI suboptions within this command are those that are valid for IPA(LINK):

MAP Generates some extra breakdown that shows where variables and data came from

LEVEL(2)

Specifies the maximum level of IPA optimization is to be used

The -W1,I option keyword specifies that these are compiler options that are to be passed to the IPA(LINK) step. Chapter 4, “Compiler Options,” on page 43 documents the compiler options and whether they are valid during the IPA Link step. INLRPT triggers an inline report showing what inlining was done by IPA. LIST triggers a pseudo assembly listing for each partition.

Notes:

1. In this case, the name of the output file for the listing was provided as a suboption.
2. Even with IPA, the -2 or -3 option should be used to specify the opt level that the traditional optimizer should be called with.

This example shows the advantage of using discrete listing options (MAP, LIST, INLRPT) over using -V. -V may give you so much information that it creates a huge file. By using the individual options, you get more control and (with LIST) the ability to route the listing to the location of your choice without redirecting the output of your c89 command.

Building example 1. in batch

For this example the following table shows the mapping of SCCNSAM data set members to given file names:

SCCNSAM member name	Name used in this example
CCNGHI1	IPA.SOURCE(HELLO1)
CCNGHI2	IPA.SOURCE(HELLO2)
CCNGHI3	IPA.SOURCE(HELLO3)

The following JCL can be used to create an object deck that can be linked to create the module (the link JCL is omitted for brevity):

```

/USERID1A JOB (127A,0329),'$MEM$',
// MSGLEVEL=(2,0),MSGCLASS=S,CLASS=A,
// NOTIFY=USERID1,REGION=1024M
//PROC JCLLIB ORDER=(CBC.SCCNPRC)
/*-----
/* IPA Compile Step for hello1.c
/*-----
//C001F336 EXEC EDCC,
//          INFILE='USERID1.IPA.SOURCE(HELLO1)',
//          OUTFILE='USERID1.IPA.OBJECT(HELLO1),DISP=SHR',
//          CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS DD *
          IPA(NOOBJECT,LIST) RENT LONG OPT(2)

```

```

/*
/*-----
/* IPA Compile Step for hello2.c
/*-----
//C001F336 EXEC EDCC,
//      INFILE='USERID1.IPA.SOURCE(HELLO2)',
//      OUTFILE='USERID1.IPA.OBJECT(HELLO2),DISP=SHR',
//      CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS DD *
//      IPA(NOOBJECT,LIST) RENT LONG OPT(2)
/*
/*-----
/* IPA Compile Step for hello3.c
/*-----
//C001F336 EXEC EDCC,
//      INFILE='USERID1.IPA.SOURCE(HELLO3)',
//      OUTFILE='USERID1.IPA.OBJECT(HELLO3),DISP=SHR',
//      CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS DD *
//      IPA(NOOBJECT,LIST) RENT LONG OPT(2)
/*
/*-----
/* IPA Link Step for the hello module
/*-----
//C001F336 EXEC EDCI,
//      OUTFILE='USERID1.IPALINK.OBJECT(HELLO),DISP=SHR',
//      IPARM='OPTFILE(DD:OPTIONS)'
/* The following line sets up an input file that just includes all
/* the IPA Compile Step object files.
//SYSIN DD DATA,DLM='>'
//      INCLUDE OBJECT(HELLO1,HELLO2,HELLO3)
/>
/* The following line redirects the listing
//SYSPRT DD DSN=USERID1.IPA.LISTING(HELLO),DISP=SHR
/* These are the options used
//OPTIONS DD DATA,DLM='>'
//      IPA(LINK,MAP,LEVEL(2)) OPT(2) INLRPT LIST RENT LONGNAME
/>
/* The following line gives the object library
//OBJECT DD DSN=USERID1.IPA.OBJECT,DISP=SHR

```

The options used are the same as those given in the USS example above with the exception that IPA(LINK) should be explicitly specified, and RENT, and LONGNAME are not the default for C in batch so they also need to be specified. This sample JCL was created using the standard cataloged procedures shipped with the z/OS C/C++ compiler.

The generated file hello.lst is as follows:

***** PROLOG *****

Compile Time Library : 41060000

Command options:

Primary input name. : DD:SYSIN

Compiler options. : *IPA(LINK,MAP,NOREFMAP,LEVEL(2),DUP,ER,NONCAL,NOUNPCASE,NOPDF1,NOPDF2,NOPDFNAME,NOCONTROL)
 : *NOGONUMBER *NOALIAS *TERMINAL *LIST *NOXREF *NOATTR *NOOFFSET
 : *MEMORY *NOCSECT *LIBANSI *FLAG(1)
 : *NOTEST(NOSYM,NOBLOCK,NOLINE,NOPATH,HOOK) *OPTIMIZE(2)
 : *INLINE(AUTO,REPORT,1000,8000) *OPTFILE(DD:OPTIONS) *NOSERVICE *NOOE
 : *NOLOCALE *HALT(16) *NOGOFF

***** END OF PROLOG *****

***** OBJECT FILE MAP *****

*ORIGIN	IPA	FILE ID	FILE NAME
P		1	//DD:SYSIN
PI	Y	2	USERID1.IPA.OBJECT(HELLO1)
PI	Y	3	USERID1.IPA.OBJECT(HELLO2)
PI	Y	4	USERID1.IPA.OBJECT(HELLO3)
L		5	CEEZ160.SCEELKED(PRINTF)
L		6	CEEZ160.SCEELKED(CEESG003)

ORIGIN: P=primary input PI=primary INCLUDE SI=secondary INCLUDE IN=internal
 A=automatic call U=UPCASE automatic call R=RENAME card L=C Library

***** END OF OBJECT FILE MAP *****

Figure 29. Example of an IPA listing (Part 1 of 8)

* * * * * C O M P I L E R O P T I O N S M A P * * * * *

```

SOURCE FILE ID  COMPILER OPTIONS
1               *AGGRCOPY(NOOVERLAP) *NOALIAS *ANSIALIAS *ARCH(5) *ARGPARSE
                *CHARSET(BIAS=EBCDIC,LIB=EBCDIC) *NOCOMPACT *NOCOMPRESS *NODLL(NOCALLBACKANY) *ENV(MVS)
                *EXECOPS *FLOAT(HEX,FOLD,AFP) *NOGONUMBER *NOIGNERRNO *ILP32 *NOINITAUTO
                *IPA(NOLINK,NOOBJECT,COMPRESS) *NOLIBANSI *NOLIST *NOLOCALE *LONGNAME
                *MAXMEM(2097152) *OPTIMIZE(2) *PLIST(HOST) *REDIR *RENT *NOROCONST *SPILL(128)
                *NOSTART *STRICT *NOSTRICT_INDUCTION *NOTEST *TUNE(5) *NOXPLINK *NOXREF

2               *AGGRCOPY(NOOVERLAP) *NOALIAS *ANSIALIAS *ARCH(5) *ARGPARSE
                *CHARSET(BIAS=EBCDIC,LIB=EBCDIC) *NOCOMPACT *NOCOMPRESS *NODLL(NOCALLBACKANY) *ENV(MVS)
                *EXECOPS *FLOAT(HEX,FOLD,AFP) *NOGONUMBER *NOIGNERRNO *ILP32 *NOINITAUTO
                *IPA(NOLINK,NOOBJECT,COMPRESS) *NOLIBANSI *NOLIST *NOLOCALE *LONGNAME
                *MAXMEM(2097152) *OPTIMIZE(2) *PLIST(HOST) *REDIR *RENT *NOROCONST *SPILL(128)
                *NOSTART *STRICT *NOSTRICT_INDUCTION *NOTEST *TUNE(5) *NOXPLINK *NOXREF

3               *AGGRCOPY(NOOVERLAP) *NOALIAS *ANSIALIAS *ARCH(5) *ARGPARSE
                *CHARSET(BIAS=EBCDIC,LIB=EBCDIC) *NOCOMPACT *NOCOMPRESS *NODLL(NOCALLBACKANY) *ENV(MVS)
                *EXECOPS *FLOAT(HEX,FOLD,AFP) *NOGONUMBER *NOIGNERRNO *ILP32 *NOINITAUTO
                *IPA(NOLINK,NOOBJECT,COMPRESS) *NOLIBANSI *NOLIST *NOLOCALE *LONGNAME
                *MAXMEM(2097152) *OPTIMIZE(2) *PLIST(HOST) *REDIR *RENT *NOROCONST *SPILL(128)
                *NOSTART *STRICT *NOSTRICT_INDUCTION *NOTEST *TUNE(5) *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 * * * * *

* * * * * I N L I N E R E P O R T * * * * *

IPA Inline Report (Summary)

```

Reason:  P : #pragma noline was specified for this routine
         F : #pragma inline was specified for this routine
         A : Automatic inlining
         C : Partition conflict
         N : Not IPA Object
         - : No reason

Action:  I : Routine is inlined at least once
         L : Routine is initially too large to be inlined
         T : Routine expands too large to be inlined
         C : Candidate for inlining but not inlined
         N : No direct calls to routine are found in file (no action)
         U : Some calls not inlined due to recursion or parameter mismatch
         - : No action

Status:  D : Internal routine is discarded
         R : A direct call remains to internal routine (cannot discard)
         A : Routine has its address taken (cannot discard)
         E : External routine (cannot discard)
         - : Status unchanged

Calls/I   : Number of calls to defined routines / Number inline
Called/I   : Number of times called / Number of times inlined

```

Reason	Action	Status	Size (init)	Calls/I	Called/I	Name
A	I	D	0 (40)	2/1	1/1	func2
A	I	D	0 (32)	1/0	1/1	func3
A	N	-	38 (28)	1/1	0	main
N	-	E	0	0	2/0	PRINTF

Mode = AUTO Inlining Threshold = 1000 Expansion Limit = 8000

Figure 29. Example of an IPA listing (Part 2 of 8)

IPA Inline Report (Call Structure)

```

Defined Subprogram : main
  Calls To(1,1)    : func2(1,1)
  Called From      : 0

Defined Subprogram : func2
  Calls To(2,1)    : func3(1,1)
                  : PRINTF(1,0)
  Called From(1,1) : main(1,1)

Defined Subprogram : PRINTF
  Calls To          : 0
  Called From(2,0) : func3(1,0)
                  : func2(1,0)

Defined Subprogram : func3
  Calls To(1,0)    : PRINTF(1,0)
  Called From(1,1) : func2(1,1)

```

***** END OF INLINE REPORT *****

***** PARTITION MAP *****

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(5) *ARGPARSE *CHARSET(BIAS=EBCDIC,LIB=EBCDIC) *NOCOMPACT *NOCOMPRESS
*NOCSECT *NODLL *ENV(MVS) *EXECOPS *FLOAT(HEX,FOLD,AFP) *NOGOFF *NOGONUMBER *NOIGNERRNO *ILP32
*NOINITAUTO *IPA(LINK) *LIBANSI *LIST *NOLOCALE *LONGNAME *MAXMEM(2097152) *OPTIMIZE(2) *PLIST(HOST)
*REDIR *RENT *NOROCONST *SPILL(128) *START *STRICT *NOSTRICT_INDUCTION *NOTEST *TUNE(5)
*NOXPLINK *XREF

```

SYMBOLS IN PARTITION 0:

```

*TYPE FILE ID SYMBOL
D      1      string1
D      2      string2
D      1      seen_main
D      1      seen_unused3
D      1      stringU1
D      2      seen_func2
D      2      seen_unused2
D      3      seen_func3
D      3      seen_unused1

```

TYPE: F=function D=data

SOURCE FILES FOR PARTITION 0:

```

*ORIGIN FILE ID SOURCE FILE NAME
P       1      //'USERID1.IPA.SOURCE(HELLO1)'
P       2      //'USERID1.IPA.SOURCE(HELLO2)'
P       3      //'USERID1.IPA.SOURCE(HELLO3)'

```

ORIGIN: P=primary input PI=primary INCLUDE

***** END OF PARTITION MAP *****

Figure 29. Example of an IPA listing (Part 3 of 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

```

Timestamp and Version Information
000000 F2F0 F0F3           =C'2003'           Compiled Year
000004 F1F1 F2F5           =C'1125'           Compiled Date MMDD
000008 F0F9 F1F4 F2F4     =C'091424'         Compiled Time HHMMSS
00000E F0F1 F0F6 F0F0     =C'010600'         Compiler Version
Timestamp and Version End
    
```

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

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	000028	@STATICP
PR	2	000000	000004	string1
PR	3	000000	000004	string2
PR	4	000000	000004	seen_main
PR	5	000000	000004	seen_unused3
PR	6	000000	000004	stringU1
PR	7	000000	000004	seen_func2
PR	8	000000	000004	seen_unused2
PR	9	000000	000004	seen_func3
PR	10	000000	000004	seen_unused1

* * * * * P A R T I T I O N M A P * * * * *

PARTITION 1 OF 1

PARTITION SIZE:
Actual: 456
Limit: 102400

PARTITION CSECT NAMES:
Code: none
Static: none
Test: none

PARTITION DESCRIPTION:
Primary partition

COMPILER OPTIONS FOR PARTITION 1:

*AGGRCOPY(NOOVERLAP)	*NOALIAS	*ARCH(5)	*ARGPARSE	*CHARSET(BIAS=EBCDIC,LIB=EBCDIC)	*NOCOMPACT	*NOCOMPRESS
*NOCSECT	*NODLL	*ENV(MVS)	*EXECOPS	*FLOAT(HEX,FOLD,AFP)	*NOGOF	*NOIGNERRNO
*NOINITAUTO	*IPA(LINK)	*LIBANSI	*LIST	*NOLOCALE	*LONGNAME	*MAXMEM(2097152)
*REDIR	*RENT	*NOROCONST	*SPILL(128)	*START	*STRICT	*NOSTRICT_INDUCTION
*NOXPLINK	*XREF				*NOTEST	*TUNE(5)

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	//'USERID1.IPA.SOURCE(HELLO1)'
P	2	//'USERID1.IPA.SOURCE(HELLO2)'
P	3	//'USERID1.IPA.SOURCE(HELLO3)'

ORIGIN: P=primary input PI=primary INCLUDE

* * * * * E N D O F P A R T I T I O N M A P * * * * *

Figure 29. Example of an IPA listing (Part 4 of 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

Timestamp and Version Information
000000 F2F0 F0F3          =C'2003'          Compiled Year
000004 F1F1 F2F5          =C'1125'          Compiled Date MMDD
000008 F0F9 F1F4 F2F4    =C'091424'       Compiled Time HHMMSS
00000E F0F1 F0F6 F0F0    =C'010600'       Compiler Version

Timestamp and Version End

```

```

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          000010 | 1  main  DS  0D
000018 47F0 F022      000010 | 1          B  34(,r15)
00001C 01C3C5C5          CEE eyecatcher
000020 000000A8          DSA size
000024 000000C0          =A(PPA1-main)
000028 47F0 F001      000010 | 1          B  1(,r15)
00002C 58F0 C31C      000010 | 1          L  r15,796(,r12)
000030 184E          000010 | 1          LR r4,r14
000032 05EF          000010 | 1          BALR r14,r15
000034 00000000          =F'0'
000038 07F3          000010 | 1          BR  r3
00003A 90E4 D00C      000010 | 1          STM r14,r4,12(r13)
00003E 58E0 D04C      000010 | 1          L  r14,76(,r13)
000042 4100 E0A8      000010 | 1          LA  r0,168(,r14)
000046 5500 C314      000010 | 1          CL  r0,788(,r12)
00004A 4130 F03A      000010 | 1          LA  r3,58(,r15)
00004E 4720 F014      000010 | 1          BH  20(,r15)
000052 58F0 C280      000010 | 1          L  r15,640(,r12)
000056 90F0 E048      000010 | 1          STM r15,r0,72(r14)
00005A 9210 E000      000010 | 1          MVI 0(r14),16
00005E 5000 E004      000010 | 1          ST  r13,4(,r14)
000062 18DE          000010 | 1          LR  r13,r14
000064          End of Prolog

000064 58E0 C1F4      000010 | 1          L  r14,_CEECAA_(,r12,500)
000068 5840 306E      000010 | 1          L  r4,=Q(string1)(,r3,110)
00006C C020 0000 0032 000000 LARL r2,F'50'
000072 58F0 3072      000013 | 2 +        L  r15,=V(sprintf)(,r3,114)
000076 4110 D098      000013 | 2 +        LA  r1,#MX_TEMP1(,r13,152)
00007A 5804 E000      000010 | 1          L  r0,string1(r4,r14,0)
00007E 5020 D098      000013 | 2 +        ST  r2,#MX_TEMP1(,r13,152)
000082 5000 D09C      000013 | 2 +        ST  r0,#MX_TEMP1(,r13,156)
000086 05EF          000013 | 2 +        BALR r14,r15
000088 58E0 3076      000013 | 2 +        L  r14,=Q(string2)(,r3,118)
00008C 5840 C1F4      000013 | 2 +        L  r4,_CEECAA_(,r12,500)
000090 4100 2004      000017 | 3 +        LA  r0,+CONSTANT_AREA(,r2,4)
000094 58F0 3072      000017 | 3 +        L  r15,=V(sprintf)(,r3,114)
000098 4110 D098      000017 | 3 +        LA  r1,#MX_TEMP1(,r13,152)
00009C 58EE 4000      000013 | 2 +        L  r14,string2(r14,r4,0)
0000A0 5000 D098      000017 | 3 +        ST  r0,#MX_TEMP1(,r13,152)
0000A4 50E0 D09C      000017 | 3 +        ST  r14,#MX_TEMP1(,r13,156)
0000A8 05EF          000017 | 3 +        BALR r14,r15
0000AA 41F0 0000      000015 | 1          LA  r15,0
0000AE          000015 | 1 @1L3    DS  0H

0000AE          Start of Epilog
0000AE 180D          000016 | 1          LR  r0,r13
0000B0 58D0 D004      000016 | 1          L  r13,4(,r13)
0000B4 58E0 D00C      000016 | 1          L  r14,12(,r13)
0000B8 9824 D01C      000016 | 1          LM  r2,r4,28(r13)
0000BC 051E          000016 | 1          BALR r1,r14
0000BE 0707          000016 | 1          NOPR 7

0000C0          Start of Literals
0000C0 00000000          =Q(string1)
0000C4 00000000          =V(sprintf)

```

Figure 29. Example of an IPA listing (Part 5 of 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
0000C8 00000000                =Q(string2)
0000CC                               End of Literals

*** General purpose registers used: 1111100000001111
*** Floating point registers used: 1111111100000000
*** Size of register spill area: 128(max) 0(used)
*** Size of dynamic storage: 168
*** Size of executable code: 168
    
```

0000CC 0000 0000

```

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

0000D8 1CCEA106                =F'483303686'   Flags
0000DC 00000100                =A(PPA2-main)
0000E0 00000000                =F'0'           No PPA3
0000E4 00000000                =F'0'           No EPD
0000E8 FE000000                =F'-33554432'   Register save mask
0000EC 00000000                =F'0'           Member flags
0000F0 90                      =AL1(144)       Flags
0000F1 000000                =AL3(0)         Callee's DSA use/8
0000F4 0040                  =H'64'          Flags
0000F6 0012                  =H'18'          Offset/2 to CDL
0000F8 00000000                =F'0'           Reserved
0000FC 50000054                =F'1342177364' CDL function length/2
000100 FFFFFFF4                =F'-192'        CDL function EP offset
000104 38260000                =F'942014464'   CDL prolog
000108 4009004B                =F'1074331723' CDL epilog
00010C 00000000                =F'0'           CDL end
000110 0004 ****                AL2(4),C'main'

PPA1 End

PPA2: Compile Unit Block

000118 0300 2202                =F'50340354'   Flags
00011C FFFF FEE8                =A(CEESTART-PPA2)
000120 0000 0000                =F'0'           No PPA4
000124 FFFF FEE8                =A(TIMESTAMP-PPA2)
000128 0000 0000                =F'0'           No primary
00012C 0000 0000                =F'0'           Flags

PPA2 End
    
```

Figure 29. Example of an IPA listing (Part 6 of 8)

EXTERNAL SYMBOL DICTIONARY				
TYPE	ID	ADDR	LENGTH	NAME
SD	1	000000	000130	@STATICP
LD	0	000018	000001	main
ER	2	000000		CEESG003
PR	3	000000	000000	string1
ER	4	000000		PRINTF
PR	5	000000	000000	string2
ER	6	000000		CEESTART
SD	7	000000	000008	@PPA2
SD	8	000000	00000C	CEEMAIN
ER	9	000000		EDCINPL

Figure 29. Example of an IPA listing (Part 7 of 8)

```

***** SOURCE FILE MAP *****
OBJECT      SOURCE
*ORIGIN     FILE ID  FILE ID  SOURCE FILE NAME
P           2       1      //'USERID1.IPA.SOURCE(HELLO1)'  

           3       2      //'USERID1.IPA.SOURCE(HELLO2)'  

           4       3      //'USERID1.IPA.SOURCE(HELLO3)'  

ORIGIN: P=primary input  PI=primary INCLUDE
***** END OF SOURCE FILE MAP *****

```

```

***** MESSAGE SUMMARY *****
TOTAL  UNRECOVERABLE  SEVERE  ERROR  WARNING  INFORMATIONAL
        (U)          (S)      (E)      (W)      (I)
    0          0          0          0          0
***** END OF MESSAGE SUMMARY *****
***** END OF COMPILATION *****

```

Figure 29. Example of an IPA listing (Part 8 of 8)

After a traditional compile, there are three object files, six external functions, and eight external variables. Without a global view of the application, the compiler looks at hello1.c and cannot tell that unused3 is really unused and that stringU1 is never referenced. So the compiler has to keep all of the code and variables. IPA has the global view so it can remove the unused functions. As you can see from listing file above, only the main function remains. The other functions were inlined, and because they were not exported, and their address was not taken, they were removed.

Example 2. all C parts built with XPLINK

The second example is a variation of the first example. The purpose of this example is to show how easy it is to build an application with both XPLINK and IPA. To simplify the options even more, this example will not generate any listings. Please refer to the appropriate sections of “Example 1. all C parts” on page 329 to map the given names to the members of the SCCNSAM data set.

Building example 2. under USS

The only addition to the IPA compile step is the required addition of the XPLINK option. The GOFF option has also been added (this option defaults **on** when XPLINK is specified) for convenience purposes.

```
c89 -c -2 -WI,N00BJECT -Wc,XPLINK,GOFF hello1.c hello2.c hello3.c
```

For the IPA Link step, the changes are similar to the compile step, and the basic changes that must be done to use XPLINK under USS. The option -W1,XPLINK is added to guide c89 to include the XPLINK libraries in the IPA link step.

```
c89 -2 -WI,LEVEL\{2\} -W1,XPLINK -o hello hello1.o
    hello2.o hello3.o
```

Building example 2. in batch

In batch, the same basic changes are made. XPLINK and GOFF are added to the IPA Compile steps and the XPLINK proc EDCXI is used instead of EDCI. A few extra

includes (CELHS003,CELHS001) are placed in the IPA input to allow IPA to resolve XPLINK library references. This job will result in an object deck that can then be linked to create the module.

```
//USERID1A JOB (127A,0329),'$MEM$',
// MSGLEVEL=(2,0),MSGCLASS=S,CLASS=A,
// NOTIFY=USERID1,REGION=1024M
//PROC JCLLIB ORDER=(CBC.SCCNPRC)
//*-----
//* IPA Compile Step for hello1.c
//*-----
//C001F336 EXEC EDCC,
//      INFILE='USERID1.IPA.SOURCE(HELLO1)',
//      OUTFILE='USERID1.IPA.OBJECT(HELLOX1),DISP=SHR',
//      CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS DD *
//      IPA(NOOBJECT) RENT LONG OPT(2) XPLINK GOFF
/*
//*-----
//* IPA Compile Step for hello2.c
//*-----
//C001F336 EXEC EDCC,
//      INFILE='USERID1.IPA.SOURCE(HELLO2)',
//      OUTFILE='USERID1.IPA.OBJECT(HELLOX2),DISP=SHR',
//      CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS DD *
//      IPA(NOOBJECT) RENT LONG OPT(2) XPLINK GOFF
/*
//*-----
//* IPA Compile Step for hello3.c
//*-----
//C001F336 EXEC EDCC,
//      INFILE='USERID1.IPA.SOURCE(HELLO3)',
//      OUTFILE='USERID1.IPA.OBJECT(HELLOX3),DISP=SHR',
//      CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS DD *
//      IPA(NOOBJECT) RENT LONG OPT(2) XPLINK GOFF
/*
//*-----
//* IPA Link Step for the hello module
//*-----
//C001F336 EXEC EDCXI,
//      OUTFILE='USERID1.IPALINK.OBJECT(HELLOXP),DISP=SHR',
//      IPARM='OPTFILE(DD:OPTIONS)'
//* The following line sets up an input file that just includes all
//* the IPA Compile Step object files.
//SYSIN DD DATA,DLM='>'
//      INCLUDE OBJECT(HELLOX1,HELLOX2,HELLOX3)
//      INCLUDE SYSLIB(CELHS003,CELHS001)
/>
//* These are the options used
//OPTIONS DD DATA,DLM='>'
//      IPA(LINK,LEVEL(2)) OPT(2) RENT LONGNAME
//      XPLINK GOFF
/>
//* The following line gives the object library
//OBJECT DD DSN=USERID1.IPA.OBJECT,DISP=SHR
```

Creating a DLL with IPA

This section gives several examples, which describe the aspects of building a simple DLL, as well as how to use some of IPAs advanced features to build a faster DLL. By default, IPA will try to remove unused code and variables (even global variables). In DLL situations, (or with exported variables) this ability becomes limited. For modules with a main function, IPA can build a function call tree and

determine which functions are or may be called. This list of functions is used to remove unused functions and variables. For DLLs, IPA must treat the list of exported functions as potential entry points, and all exported variables as used. For this reason, the use of the EXPORTALL compiler option is not recommended. IPA provides a control file option that allows you to specify exactly which functions and variables you wish to be exported. This gives the programmer who cannot change the source another way to avoid EXPORTALL. For an example of this, please see "Example 2. using the IPA control file" on page 344.

Example 1. a mixture of C and C++

For this example, the following table shows the mapping of SCCNSAM data set members to given file names. The main program is provided to allow the user to run the created DLL, it is not used in the following example.

SCCNSAM member name	Name used in this example
CCNGID1	GlobInfo.h
CCNGID2	UserInt.h
CCNGID3	UserInterface.C
CCNGID4	c_DLL.c
CCNGID5	c_DLL.h
CCNGID6	cpp_DLL.C
CCNGID7	cpp_DLL.h
CCNGIDM	main.C

This example involves the creation of a C/C++ DLL. This DLL consists of 1 C part and 2 C++ parts. For your convenience, a main SCCNSAM(CCNGIDM) is provided so that the program can be executed. Instructions to build the main will not be given in this example. In general, IPA DLLs are created in the same manner as IPA modules with the extra commands for DLLs added in for the IPA Link step.

Building example 1. under USS

First, IPA must compile each source file. Since NOOBJECT is the default, it is not specifically mentioned in this example. -WI is specified to trigger an IPA compile.

```
c89 -c -2 -WI -Wc,"FLAG(I),DLL" c_DLL.c
c++ -c -2 -WI -Wc,"FLAG(I)" -+ cpp_DLL.C
c++ -c -2 -WI -Wc,"FLAG(I),EXPORTALL" -+ UserInterface.C
```

Next, the IPA Link step is performed. In this case, IPA level(1) optimizations are used:

```
c++ -2 -WI,"LEVEL(1)" -Wl,I,DLL -Wl,DLL,DYNAM=DLL -o mydll
UserInterface.o c_DLL.o cpp_DLL.o
```

The LEVEL(1) suboption is fed to IPA. The DLL option is given to the traditional optimizer using the -Wl,I option and the usual linker commands for DLLs are given.

Building example 1. under batch

For this example, the following table shows the mapping of SCCNSAM data set members to given PDS member names. The main program is provided to allow the user to run the created DLL, it is not used in the following example.

SCCNSAM member name	Name used in this example
CCNGID1	IPA.H(GLOBINFO)

SCCNAM member name	Name used in this example
CCNGID2	IPA.H(USERINT)
CCNGID3	IPA.SOURCE(USERINT)
CCNGID4	IPA.SOURCE(CDLL)
CCNGID5	IPA.H(C@DLL)
CCNGID6	IPA.SOURCE(CPPDLL)
CCNGID7	IPA.H(CPP@DLL)
CCNGIDM	IPA.SOURCE(MAIN)

```

//USERID1A JOB (127A,0329),'$MEM$',
// MSGLEVEL=(2,0),MSGCLASS=S,CLASS=A,
// NOTIFY=USERID1,REGION=1024M
//PROC JCLLIB ORDER=(CBC.SCCNPRC)
/*-----
/* IPA Compile Step for CDLL
/*-----
//C001F336 EXEC EDCC,
//      INFILE='USERID1.IPA.SOURCE(CDLL)',
//      OUTFILE='USERID1.IPA.OBJECT(CDLL),DISP=SHR',
//      CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS DD *
//      IPA(NOOBJECT) RENT LONG OPT(2) DLL
//      LSEARCH('USERID1.IPA.+')
//      SEARCH('CEE.SCEEH.+')
/*
/*-----
/* IPA Compile Step for CPPDLL
/*-----
//C001F336 EXEC CBCC,
//      INFILE='USERID1.IPA.SOURCE(CPPDLL)',
//      OUTFILE='USERID1.IPA.OBJECT(CPPDLL),DISP=SHR',
//      CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS DD *
//      IPA(NOOBJECT) OPT(2)
//      LSEARCH('USERID1.IPA.+')
//      SEARCH('CEE.SCEEH.+')
/*
/*-----
/* IPA Compile Step for USERINT
/*-----
//C001F336 EXEC CBCC,
//      INFILE='USERID1.IPA.SOURCE(USERINT)',
//      OUTFILE='USERID1.IPA.OBJECT(USERINT),DISP=SHR',
//      CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS DD *
//      IPA(NOOBJECT) OPT(2) EXPORTALL
//      LSEARCH('USERID1.IPA.+')
//      SEARCH('CEE.SCEEH.+')
/*
/*-----
/* IPA Link Step for the hello module
/*-----
//C001F336 EXEC CBCI,
//      OUTFILE='USERID1.IPALINK.OBJECT(MYDLL),DISP=SHR',
//      IPARM='OPTFILE(DD:OPTIONS)'
/* The following line sets up an input file that just includes all
/* the IPA Compile Step object files.
//SYSIN DD DATA,DLM='>'
//      INCLUDE OBJECT(USERINT,CDLL,CPPDLL)
//      INCLUDE SYSLIB(C128,IOSTREAM,COMPLEX)
/>
/* These are the options used

```

```
//OPTIONS DD DATA,DLM='>'
      IPA(LINK,MAP,LEVEL(1)) OPT(2) RENT LONGNAME
/>
/* The following line gives the object library
//OBJECT DD DSN=USERID1.IPA.OBJECT,DISP=SHR
```

Example 2. using the IPA control file

The following example uses the IPA control file to choose which functions should be exported from UserInterface.C. This allows the IPA compile step to be done without the EXPORTALL option. The first step is to construct an IPA control file. The function names appearing in the IPA control file must be mangled names if the names in the source file are going to be mangled by the compiler. The file content is as follows:

```
export=get_user_input__7UIclassFv,
      get_user_sort_method__7UIclassFRi,
      call_user_sort_method__7UIclassFi,
      print_sort_result__7UIclassFv
```

Please refer to the appropriate sections of “Example 1. a mixture of C and C++” on page 342 to map the given names to the members of the SCCNSAM data set.

Building example 2. under USS

First, IPA must compile each source file using the following commands:

```
c89 -c -2 -WI -Wc,"FLAG(I),DLL" c_DLL.c
c++ -c -2 -WI -Wc,"FLAG(I)" -+ cpp_DLL.C
c++ -c -2 -WI -Wc,"FLAG(I)" -+ UserInterface.C
```

Next, the IPA Link step is run to specify a control file:

```
Uc++ -2 -WI,"LEVEL(1),CONTROL(myd11.cntl)" -Wl,I,DLL -Wl,DLL,DYNAM=DLL -o myd11
      UserInterface.o c_DLL.o cpp_DLL.o
```

This creates a DLL where only the specified functions are exported from UserInterface.C.

Building example 2. in batch

```
//USERID1A JOB (127A,0329),'$MEM$',
// MSGLEVEL=(2,0),MSGCLASS=S,CLASS=A,
// NOTIFY=USERID1,REGION=1024M
//PROC JCLLIB ORDER=(CBC.SCCNPRC)
//*-----
/* IPA Compile Step for CDLL
//*-----
//C001F336 EXEC EDCC,
//      INFILE='USERID1.IPA.SOURCE(CDLL)',
//      OUTFILE='USERID1.IPA.OBJECT(CDLL),DISP=SHR',
//      CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS DD *
      IPA(NOOBJECT) RENT LONG OPT(2) DLL
      LSEARCH('USERID1.IPA.+')
      SEARCH('CEE.SCEEH.+')
/*
//*-----
/* IPA Compile Step for CPPDLL
//*-----
//C001F336 EXEC CBCC,
//      INFILE='USERID1.IPA.SOURCE(CPPDLL)',
//      OUTFILE='USERID1.IPA.OBJECT(CPPDLL),DISP=SHR',
//      CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS DD *
      IPA(NOOBJECT) OPT(2)
      LSEARCH('USERID1.IPA.+')
```

```

SEARCH('CEE.SCEEH.+')
/*
/*-----
/* IPA Compile Step for USERINT
/*-----
//C001F336 EXEC CBCB,
//      INFILE='USERID1.IPA.SOURCE(USERINT)',
//      OUTFILE='USERID1.IPA.OBJECT(USERINT),DISP=SHR',
//      CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS DD *
//      IPA(NOOBJECT) OPT(2)
//      LSEARCH('USERID1.IPA.+')
//      SEARCH('CEE.SCEEH.+')
/*
/*-----
/* IPA Link Step for the hello module
/*-----
//C001F336 EXEC CBCI,
//      OUTFILE='USERID1.IPALINK.OBJECT(MYDLL),DISP=SHR',
//      IPARM='OPTFILE(DD:OPTIONS)'
/* The following line sets up an input file that just includes all
/* the IPA Compile Step object files.
//SYSIN DD DATA,DLM='>'
//      INCLUDE OBJECT(USERINT,CDLL,CPDLL)
//      INCLUDE SYSLIB(C128,IOSTREAM,COMPLEX)
/>
/* These are the options used
//OPTIONS DD DATA,DLM='>'
//      IPA(LINK,LEVEL(1),CONTROL('USERID1.ipa.cntl(dllx)'))
//      OPT(2) RENT LONGNAME
/>
/* The following line gives the object library
//OBJECT DD DSN=USERID1.IPA.OBJECT,DISP=SHR

```

In the resultant object deck (MYDLL), only functions that are explicitly exported using #pragma export and the four functions given in the control file are exported.

Using Profile-Directed Feedback (PDF)

In any large application, there are sections of code that are not often executed, such as code for error-handling. A traditional compiler cannot tell what these low frequency sections of code or functions are, and may spend a lot of time optimizing code that will never be executed. Profile-Directed Feedback (PDF) can be used to collect information about the way the program is really used and the compiler can use this information when optimizing the code. PDF also enables you to receive estimates on how many times loops are iterated.

Steps for completing the PDF process

Perform the following four basic steps to complete the PDF process:

1. Compile some or all of the source files in a program with the IPA PDF1 suboption. The OPTIMIZE(2) option, or preferably the OPTIMIZE(3) option, as well as the IPA(LEVEL(1|2)) option should be specified for optimization. Special attention should be paid to the compiler options that are used to compile the files because the same options (other than IPA(PDF1)) must be used later. In a large application, the use of the PDF1 suboption should be concentrated on those areas of the code that can benefit most from optimization. You do not need to compile all of the application's code with the PDF1 suboption but you do need to compile the main function with the PDF1 suboption.

2. Preallocate the PDF data set using RECFM = U and LRECL = 0 if you are using an MVS data set for your PDF file.
 3. Run the program built from step 1 with typical input data. The program records profiling information when it finishes. The program can be run multiple times with different input data sets, and the profiling information is accumulated to provide an accurate count of how often branches are taken and blocks of code are executed. It is critically important that the data used is representative of the data that will be used during a normal run of the finished program.
-
4. Re-build your program using the identical set of source files with the identical compiler options that you used in step 1, but change the PDF1 suboption to PDF2. In this second stage, the accumulated profiling information is used to fine-tune the optimizations. The resulting program contains no profiling overhead and runs at full speed.
-

Specifically, the following JCL can be used to perform a PDF1 compile of the hello world program that is shown above (see “Example 1. all C parts” on page 329).

```
//USERID1A JOB (127A,0329),'$MEM$',
// MSGLEVEL=(2,0),MSGCLASS=S,CLASS=A,
// NOTIFY=USERID1,REGION=1024M
//PROC JCLLIB ORDER=(CBC.SCCNPRC)
//*-----
//* IPA Compile Step for hello1.c
//*-----
//C001F336 EXEC EDCC,
//      INFILE='USERID1.IPA.SOURCE(HELLO1)',
//      OUTFILE='USERID1.IPA.OBJECT(HELLOX1),DISP=SHR',
//      CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS DD *
//      IPA(NOOBJECT,PDF1) RENT LONG OPT(2) XPLINK GOFF
/*
//*-----
//* IPA Compile Step for hello2.c
//*-----
//C001F336 EXEC EDCC,
//      INFILE='USERID1.IPA.SOURCE(HELLO2)',
//      OUTFILE='USERID1.IPA.OBJECT(HELLOX2),DISP=SHR',
//      CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS DD *
//      IPA(NOOBJECT,PDF1) RENT LONG OPT(2) XPLINK GOFF
/*
//*-----
//* IPA Compile Step for hello3.c
//*-----
//C001F336 EXEC EDCC,
//      INFILE='USERID1.IPA.SOURCE(HELLO3)',
//      OUTFILE='USERID1.IPA.OBJECT(HELLOX3),DISP=SHR',
//      CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS DD *
//      IPA(NOOBJECT,LIST,PDF1) RENT LONG OPT(2) XPLINK GOFF
/*
//*-----
//* IPA Link Step for the hello module
//*-----
//C001F336 EXEC EDCXI,
//      OUTFILE='USERID1.IPALINK.OBJECT(HELLOXP),DISP=SHR',
//      IPARM='OPTFILE(DD:OPTIONS)'
//* The following line sets up an input file that just includes all
//* the IPA Compile Step object files.
//SYSIN DD DATA,DLM='>'
```

```

INCLUDE OBJECT(HELLOX1,HELLOX2,HELLOX3)
INCLUDE SYSLIB(CELHS003,CELHS001)
/>
/* These are the options used
//OPTIONS DD DATA,DLM='>'
IPA(LINK,LEVEL(2),MAP,PDF1,PDFNAME(//'USERID1.MY.PDF'))
OPT(2) RENT LONGNAME LIST
XPLINK GOFF
/>
/* The following line gives the object library
//OBJECT DD DSN=USERID1.IPA.OBJECT,DISP=SHR
/*-----
/* LINK the hello module
/*-----
//C001F336 EXEC CCNXP1B,
//          INFILE='USERID1.IPALINK.OBJECT(HELLOXP)',
//          OUTFILE='USERID1.DEV.LOAD1(HELLOXP),DISP=SHR'

```

Note: The IPA(PDF1) option is specified on each of the IPA compiles, and the PDFNAME suboption is specified on the IPA Link step. This PDFNAME suboption gives the name of the file where the statistics about the program will be stored, this file is referred to as the PDF file. While it is not strictly required to preallocate the PDF file, when using a PS or PDS file, the data set may be required to preallocate to ensure the file is large enough. If the PDF file is preallocated, it should be allocated with an LRECL of 0 and a RECFM of U.

Finally, instead of using a traditional link proc, the link of the PDF1 code should be done with the CCNPD1B proc (for non-XPLINK code) or CCNXP1B proc (for XPLINK code). This proc provides all the libraries necessary to allow the object file created by the IPA Link step to be linked with the PDF run-time function that stores the statistical information.

A PDF2 IPA compile job looks very similar to the above except the:

- PDF2 suboption is used in every place that PDF1 is used above
- Traditional EDCB proc can be used to bind the object created during the IPA Link step into a module

Steps for building a module in USS using PDF

Perform the following steps in USS to build a module using the PDF process:

1. Build the PDF1 module using the following commands:

```

c89 -c -2 -WI,PDF1 hello1.c hello2.c hello3.c
c89 -2 -WI,PDF1 -WI,PDF1,PDFNAME\(/hello.pdf\),LEVEL\{2\} -o
hello hello1.o hello2.o hello3.o

```

2. Run the module, to create hello.pdf:

```
hello
```

3. Rebuild the module using the information in hello.pdf using the following commands:

```

c89 -c -2 -WI,PDF2 hello1.c hello2.c hello3.c
c89 -2 -WI,PDF2,PDFNAME\(/hello.pdf\),LEVEL\{2\} -o hello hello1.o
hello2.o hello3.o

```

Reference Information

The following section provides reference information concerning invoking IPA from the c89 utility, the IPA Link step control file, and object file directives understood by IPA.

Invoking IPA from the c89 utility

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 keywords separated by commas.

If you invoke the c89 utility with at least one source file and the -c option and the -WI option, c89 automatically specifies the IPA(NOLINK) option and invokes the IPA Compile step. For example, the following command invokes the IPA Compile step for the source file hello.c:

```
c89 -c -WI hello.c
```

If you invoke the c89 utility with the -WI option and with at least one object file, do not specify the -c option and do not specify any source files. c89 automatically specifies IPA(LINK) and automatically invokes that 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 the -WI option and with at least one source file for compilation and any number of object files, and do not specify the -c c89 compiler option, c89 automatically invokes the IPA Compile step once for each compilation unit and the IPA Link step once for the entire program. It then invokes the binder. For example, the following command invokes the IPA Compile step, the IPA Link step, and the binder to create a program called foo:

```
c89 -o foo -WI,object foo.c
```

When linking an application built with IPA(PDF1), the user must specify -Wl,PDF1 to ensure the application links correctly.

Specifying options

When using c89, you can pass options to IPA, as follows:

- If you specify -WI, followed by IPA suboptions, c89 passes those suboptions to both the IPA Compile step and the IPA Link step
- 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
```

If you specify the previous command, you pass the IPA(NOOBJECT) option to both the IPA Compile and IPA Link steps, the SOURCE option to only the IPA Compile step, and the MAXMEM(2048) option to only the IPA Link step.

Other considerations

The c89 utility automatically generates all INCLUDE and LIBRARY IPA Link control statements.

IPA under c89 supports the following types of files:

- MVS PDS members
- Sequential data sets
- Hierarchical File System (HFS) files
- z/OS UNIX archive (.a) files

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.

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 G0FF option.

The default for this directive is `medium`.

partitionlist=*partition_number*[,*partition_number*]

Used to reduce the size of an IPA Link listing. If the IPA Link control file contains this directive and the LIST option is active, a pseudo-assembly listing is generated for only these partitions.

partition_number is a decimal number representing an unsigned int.

safe=*name*[,*name*]

Specifies a list of "safe" functions that are not compiled as IPA objects. These are functions that do not call a visible (not missing) function either through a direct call or a function pointer. Safe functions can modify global variables, but may not call functions that are not compiled as IPA objects.

isolated=*name*[,*name*]

Specifies a list of "isolated" functions that are not compiled as IPA objects. Neither isolated functions nor functions within their call chain can refer to global variables. IPA assumes that functions that are bound from shared libraries are isolated.

pure=*name*[,*name*]

Specifies a list of "pure" functions that are not compiled as IPA objects. These are functions that are "safe" and "isolated" and do not indirectly alter storage accessible to visible functions. A "pure" function has no observable internal state nor has side-effects, defined as potentially altering any data visible to the caller. This means that the returned value for a given invocation of a function is independent of any previous or future invocation of the function.

unknown=*name*[,*name*]

Specifies a list of "unknown" functions that are not compiled as IPA objects. These are functions that are not safe, isolated, or pure. This is the default for all functions defined within non-IPA objects. Any function specified as "unknown" can make calls to other parts of the program compiled as IPA objects and modify global variables and dummy arguments. This option greatly restricts the amount of interprocedural optimization for calls to "unknown" functions.

missing=*attribute*

Specifies the characteristics of "missing" functions. There are two types of "missing" functions:

- Functions dynamically linked from another DLL (defined using an IPA Link IMPORT control statement)

- Functions that are statically available but not compiled with the IPA option

IPA has no visibility to the code within these functions. You must ensure that all user references are resolved at IPA Link time with user libraries or run-time libraries.

The default setting for this directive is unknown. This instructs IPA to make pessimistic assumptions about the data that may be used and modified through a call to such a missing function, and about the functions that may be called indirectly through it.

You can specify the following attributes for this directive:

safe	Specifies that the missing functions are "safe". See the description for the <code>safe</code> directive, above.
isolated	Specifies that the missing functions are "isolated". See the description for the <code>isolated</code> directive, above.
pure	Specifies that the missing functions are "pure". See the description for the <code>pure</code> directive, above.
unknown	Specifies that the missing functions are "unknown". See the description for the <code>unknown</code> 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.

Object file directives understood by IPA

IPA recognizes and acts on the following binder object control directives:

- INCLUDE
- LIBRARY
- IMPORT

Some other linkage control statements (such as `NAME`, `RENAME` and `ALIAS`) are accepted and passed through to the linker.

Troubleshooting

It is strongly recommended that you resolve all warnings that occur during the IPA Link step. Resolution of these warnings often removes seemingly unrelated problems.

The following list provides frequently asked questions (Q) and their respective answers (A):

- Q - I am running out of memory while using IPA. Are there any options for reducing its use of memory?

A - IPA reacts to the `NOMEMORY` option, and the code generator will react to the `MAXMEM` option. If this does not give you sufficient memory, consider running IPA from batch where more memory can be accessed. Before switching to batch, verify with your system programmer that you have access to the maximum possible memory (both in batch and in UNIX System Services).

- Q - I am receiving a "partition too large" warning. How do I fix it?
A - Use the IPA Control file to specify a different partition size.
- Q - My IPA compile time is too long. Are there any options?
A - Using a lower IPA compilation level (0 or 1 instead of 2) will reduce the compile time. To minimize the compile time, ensure you are using the IPA(NOOBJECT) option for your IPA compiles. A smaller partition size, specified in the control file, may minimize the amount of time spent in the code generator. Limiting inlining, may improve your compile time, but it will decrease your performance gain significantly and should only be done selectively using the IPA control file. Use the IPA control file to specify little used functions as low frequency so that IPA does not spend too much time trying to optimize them.
- Q - Can I tune the IPA automatic inlining like I can for the regular inliner?
A - Yes. Use the INLINE option for the IPA Link step.
- Q - I am using IPA(PDF1) and my program will not bind. What do I do?
A - Under UNIX System Services, specify -W1,PDF1 when linking with c89 or C++. Under MVS batch, use the CCNPD1B or CCNXP1B PROCS.

Chapter 9. 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 use PDSs, you must use the prelinker for the following:
 - C++ code
 - C code compiled with the LONGNAME, RENT, or DLL options
- For G0FF and XPLINK:
If you have compiled your program with the G0FF, XPLINK, or LP64 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 535.

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 z/OS Release Guide*, where there are several references to PDSEs, and a list of prerequisite APAR fixes.

MTF

MTF does not support PDSEs. If you have to target MTF, you cannot use the binder.

IPA

Object files that are generated by the IPA Compile step using the 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 30 on page 357 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.

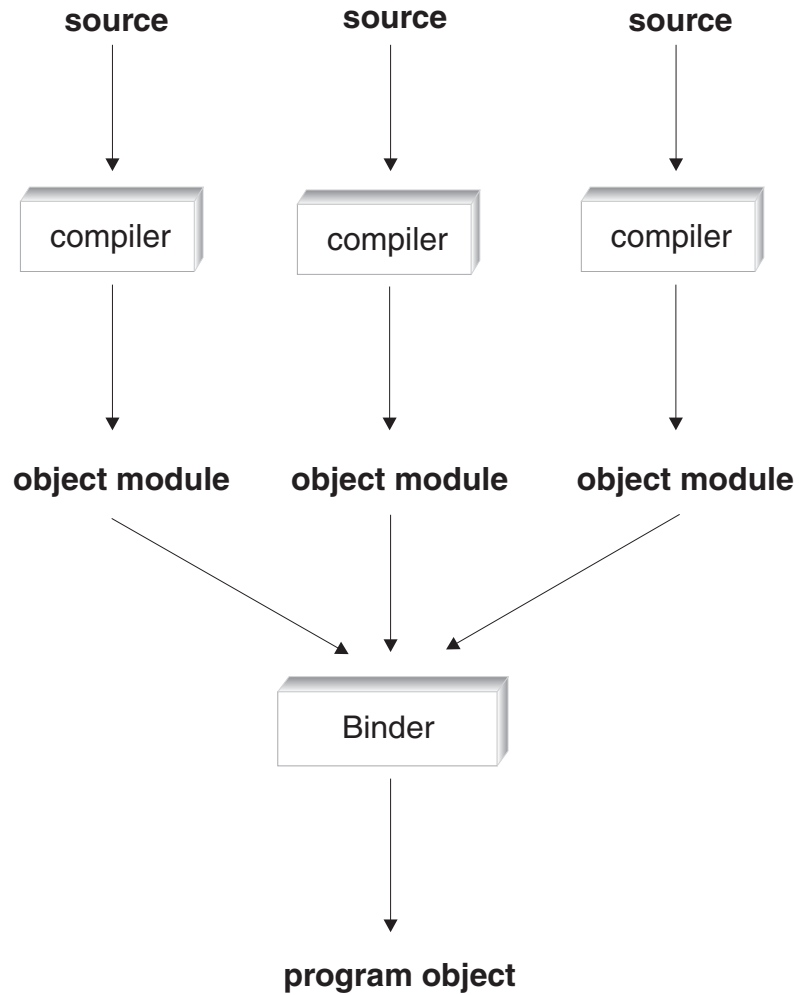


Figure 30. Single final bind

Bind each compile unit

If you have changed the source in a compile unit, you can use the method that is shown in Figure 31 on page 358. 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.

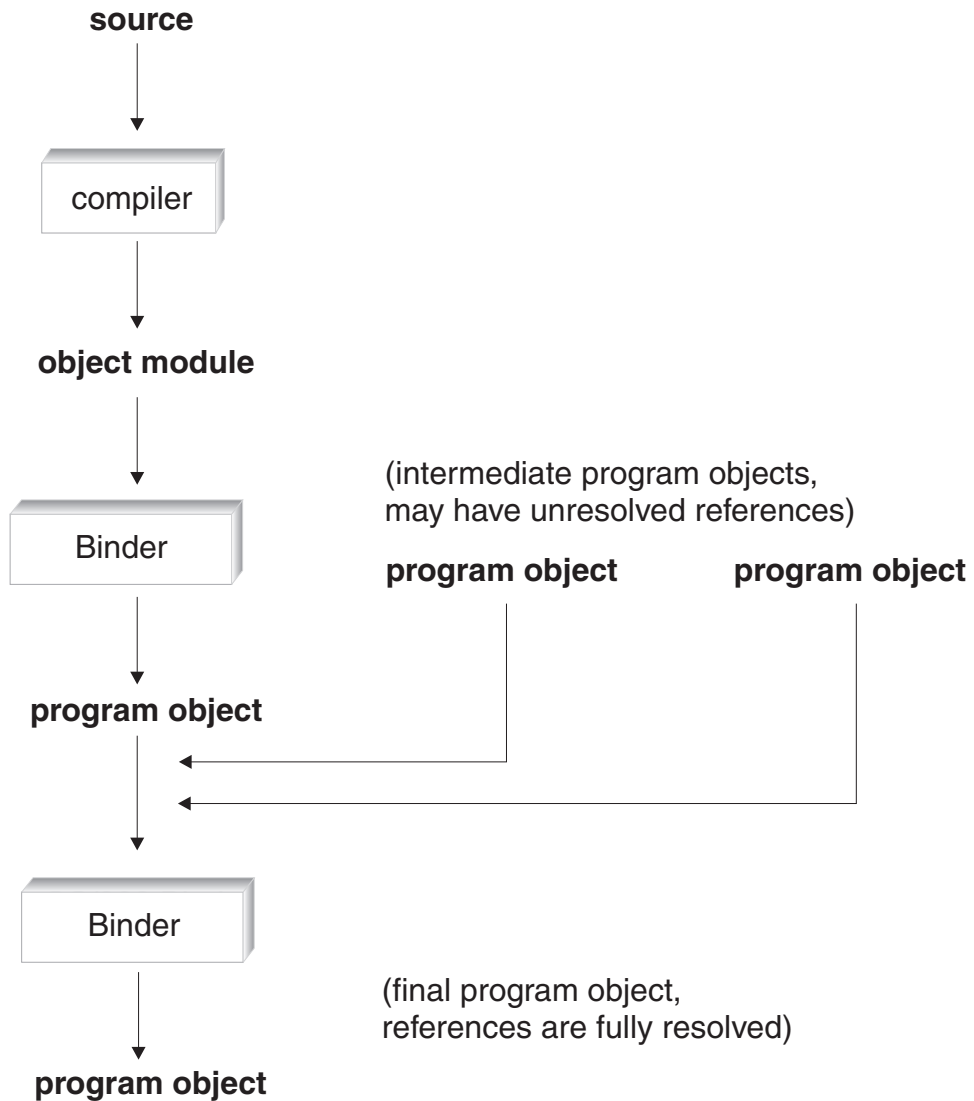


Figure 31. Bind each compile unit

Build and use a DLL

You can use the method that is shown in Figure 32 on page 359 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.

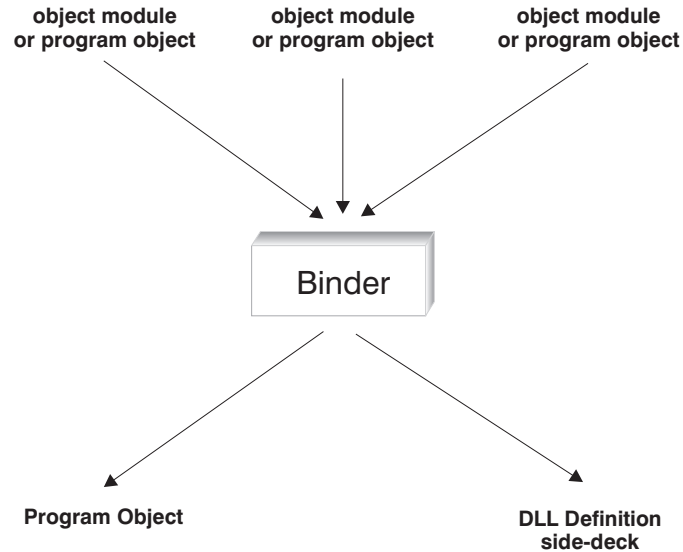


Figure 32. Build a DLL

You can use the method that is shown in Figure 33 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.

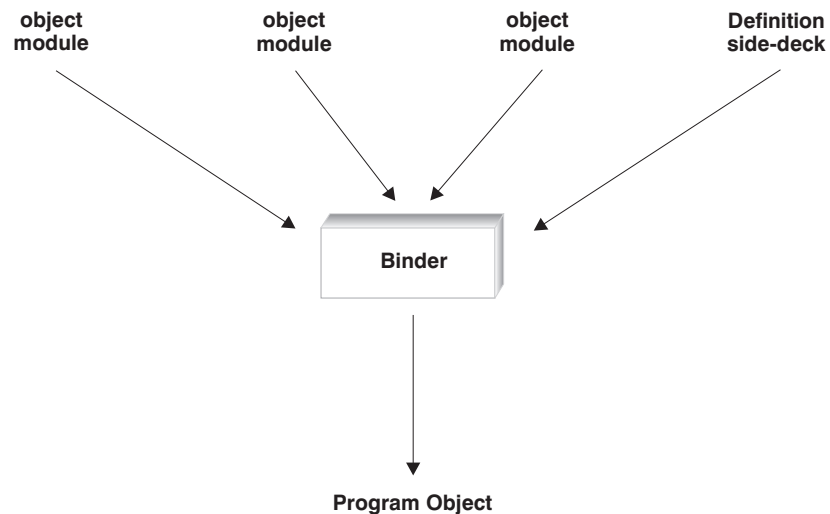


Figure 33. Build an application that uses a DLL

Rebind a changed compile unit

You can use the method shown in Figure 34 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.

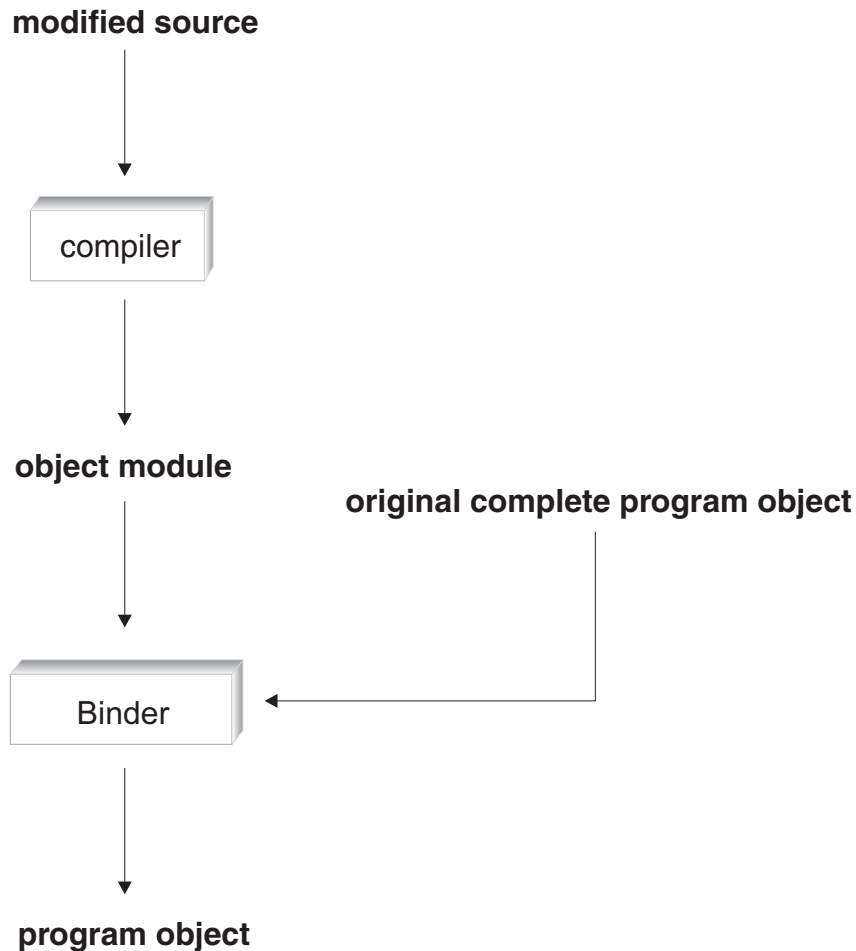


Figure 34. Rebinding a changed compile unit

Binding under z/OS UNIX System Services

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.

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 486.

Typically, you invoke the c89 and c++ utilities from the z/OS shell. For more information on these utilities, see Chapter 18, “c89 — Compiler invocation using host environment variables,” on page 471 or the *z/OS UNIX System Services Command Reference*.

To bind your XPLINK module, specify -WI,xplink on the c89/c++ command.

z/OS UNIX System Services example

The example source files unit0.c, unit1.c, and unit2.c that are shown in Figure 35, 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 35. Example source files

Steps for single final bind using c89

Before you begin: Compile each source file and then perform a single final bind.

Perform the following steps to perform a single final bind using c89:

1. Compile each source file to generate the object modules unit0.o, unit1.o, and unit2.o as follows:

```
c89 -c -W c,"CSECT(myprog)" unit0.c
c89 -c -W c,"CSECT(myprog)" unit1.c
c89 -c -W c,"CSECT(myprog)" unit2.c
```

2. Perform a final single bind to produce the executable program myprog. Use the c89 utility as follows:

```
c89 -o myprog unit0.o unit1.o unit2.o
```

The `-o` option of the `c89` command specifies the name of the output executable. The `c89` utility recognizes from the file extension `.o` that `unit0.o`, `unit1.o` and `unit2.o` are not to be compiled but are to be included in the bind step.

Example: The following is an example of a makefile to perform a similar build:

```
PGM = myprog
SRCS = unit0.c unit1.c unit2.c
OBJJS = $(SRCS:b:+".o")
COPTS = -W c,"CSECT(myprog)"
$(PGM) : ($OBJJS)
        c89 -o $(PGM) $(OBJJS)
%.o : %.c
        c89 -c -o $@ $(COPTS) $<
```

For more information on makefiles, see *z/OS UNIX System Services Programming Tools*.

Advantage

This method is simple, and is consistent with existing methods of building applications, such as makefiles.

Steps for binding each compile unit using `c89`

Before you begin: Compile each source file and also bind it.

Perform the following steps to complete a final bind of all the partially bound units:

1. Compile each source file to its object module (`.tmp`). Bind each object module into a partially bound program object (`.o`), which may have unresolved references. In this example, references to `f1()` and `f4()` in `unit0.o` are unresolved. When the partially bound programs are created, remove the object modules as they are no longer needed. Use `c89` to compile each source file, as follows:

```
c89 -c -W c,"CSECT(myprog)" -o unit0.tmp unit0.c
c89 -r -o unit0.o unit0.tmp
rm unit0.tmp
```

```
c89 -c -W c,"CSECT(myprog)" -o unit1.tmp unit1.c
c89 -r -o unit1.o unit1.tmp
rm unit1.tmp
```

```
c89 -c -W c,"CSECT(myprog)" -o unit2.tmp unit2.c
c89 -r -o unit2.o unit2.tmp
rm unit2.tmp
```

The `-r` option supports rebinding by disabling autocall processing.

2. Perform the final single bind to produce the executable program `myprog` by using `c89`:

```
c89 -o myprog unit0.o unit1.o unit2.o
```

Example: The following is an example of a makefile for performing a similar build:

```

_C89_EXTRA_ARGS=1
_EXPORT : _C89_EXTRA_ARGS      1
PGM = myprog                    2
SRCS = unit0.c unit1.c unit2.c  3
OBJS = $(SRCS:b:+".o")         4
COPTS = -W c,"CSECT(myprog)"
$(PGM) : $(OBJS)               5
    c89 -o $(PGM) $(OBJS)
%.tmp : %.c                    6
    c89 -c -o $@ $(COPTS) $<
%.o : %.tmp                    7
    c89 -r -o $@ $<

```

- 1** Export the environment variable `_C89_EXTRA_ARGS` so `c89` will process files with non-standard extensions. Otherwise `c89` will not recognize `unit0.tmp`, and the makefile will fail
- 2** name of executable
- 3** list of source files
- 4** list of partly bound parts
- 5** executable depends on parts
- 6** make `.tmp` file from `.c`
- 7** make `.o` from `.tmp`

In this example, `make` automatically removes the intermediate `.tmp` files after the makefile completes, since they are not marked as `PRECIOUS`. For more information on makefiles, see *z/OS UNIX System Services Programming Tools*.

Advantage

Binding a set of partially bound program objects into a fully bound program object is faster than binding object modules into a fully bound program object for `NOGOFF` objects. For example, a central build group can create the partially bound program objects. Developers can then use these program objects and their changed object modules to create a development program object.

Steps for building and using a DLL using `c89`

Before you begin: Build `unit1.c` and `unit2.c` into DLL `onetwo`, which exports functions `f1()`, `f2()`, `f3()`, and `f4()`. Then build `unit0.c` into a program which dynamically links to functions `f1()` and `f4()` defined in the DLL.

Perform the following steps to build and use a DLL using `c89`:

1. Compile `unit1.c` and `unit2.c` to generate the object modules `unit1.o` and `unit2.o` which have functions to be exported. Use the `c89` utility as follows:

```

c89 -c -W c,"EXPORTALL,CSECT(myprog)" unit1.c
c89 -c -W c,"EXPORTALL,CSECT(myprog)" unit2.c

```

2. Bind `unit1.o` and `unit2.o` to generate the DLL `onetwo`:

```

c89 -Wl,dll -o onetwo unit1.o unit2.o

```

When you bind code with exported symbols, you should specify the DLL binder option (`-W l,dll`).

In addition to the DLL `onetwo` being generated, the binder writes a list of `IMPORT` control statements to `onetwo.x`. This list is known as the definition side-deck. One `IMPORT` control statement is written for each exported symbol.

These generated control statements will be included later as input to the bind step of an application that uses this DLL, so that it can import the symbols.

3. Compile `unit0.c` with the DLL option `-W c,DLL`, so that it can import unresolved symbols. Bind the object module, with the definition side-deck `onetwo.x` from the DLL build:

```
c89 -c -W c,DLL unit0.c
c89 -o dll12usr unit0.o onetwo.x
```

Advantage

The bind time advantage of using DLLs is that you only need to rebuild the DLL with the changed code in it. You do not need to rebuild all applications that use the DLL in order to use the changed code.

Steps for rebinding a changed compile unit using c89

Before you begin: Rebuild an application after making a change to a single source file.

Perform the following steps to rebind a changed compile unit using c89:

1. Recompile the single changed source file. Use the compile time option `CSECT` to ensure that each section is named for purposes of rebinding. 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 $(?:b+"$@")
```

- 1** allow filenames with non-standard suffixes
- 2** list of source files
- 3** do not delete `myprog` if the make fails
- 4** compile source files newer than the executable, and bind

The attribute `.PRECIOUS` 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 Chapter 18, “`c89` — Compiler invocation using host environment variables,” on page 471. For a description of `make`, see *z/OS UNIX System Services Command Reference* and for a `make` tutorial, see *z/OS UNIX System Services Programming Tools*.

Advantage

Rebinds are fast because most of the program is already bound. Also, none of the intermediate object modules need to be retained because they are available from the program itself.

Using the non-XPLINK version of the Standard C++ Library and `c89`

A non-XPLINK Standard C++ Library DLL is available that provides Standard C++ Library support for CICS and IMS. The CICS and IMS subsystems do not support XPLINK linkage, rendering the XPLINK Standard C++ Library DLL supplied with the compiler inoperable under both subsystems. The non-XPLINK Standard C++ Library DLL allows support for the Standard C++ Library in the CICS and IMS subsystems, as of `z/OS V1R2`. Since CICS and IMS do not support XPLINK linkage, and there are no plans to support XPLINK linkage, a non-XPLINK DLL enables the Standard C++ Library under these subsystems.

To use the non-XPLINK Standard C++ Library DLL, you must first link your object modules with the non-XPLINK system definition side-deck. Use the `{_PSYSIX}` environment variable to pass the non-XPLINK side deck information to `c++/cxx`. The `{_PSYSIX}` environment variable specifies the system definition side-deck list to be used to resolve symbols during the non-XPLINK link-editing phase. The following concatenation should be used:

```
export _CXX_PSYSIX=\
"{_PLIB_PREFIX}.SCEELIB(C128N)":\
"{_CLIB_PREFIX}.SCLBSID(IOSTREAM,COMPLEX)"
```

where `{_PLIB_PREFIX}` and `{_CLIB_PREFIX}` are set to a default (for example, `CEE` and `CBC`, respectively) during custom installation, or using user overrides.

It is only necessary to specify `{_PSYSIX}` in order to use the non-XPLINK side deck with IPA. Corresponding non-XPLINK IPA link step environment variables default to the value of `{_PSYSIX}`. To run a program with the non-XPLINK DLL, ensure that the `SCEERUN` data set containing the non-XPLINK DLL is in the MVS search path; that is, either specified in your `STEPLIB` or already loaded into `LPA`.

Performance

Due to performance differences between XPLINK and non-XPLINK linkages, it is expected that an XPLINK program using the XPLINK Standard C++ Library DLL will outperform a non-XPLINK program using the non-XPLINK Standard C++ Library DLL.

It is possible to use the non-XPLINK DLL with an XPLINK application, although this is not preferred. A call to a function of different linkage than the callee will result in a performance degradation due to the overhead cost required to swap from one stack type to the other.

Binding under z/OS batch

You can use the following procedures, which the z/OS C/C++ compiler supplies, to invoke the binder:

Procedure name	Description
CEEXL	C bind an XPLINK 32-bit program
CEEXLR	C bind and run an XPLINK 32-bit program
EDCCB	C compile and bind a non-XPLINK 32-bit program
EDCCBG	C compile, bind, and run a non-XPLINK 32-bit program
EDCXCB	C compile and bind an XPLINK 32-bit program
EDCXCBG	C compile, bind, and run an XPLINK 32-bit program
EDCXLDEF	Create C Source from a locale, compile, and bind the XPLINK 32-bit program
CBCB	C++ bind a non-XPLINK 32-bit program
CBCBG	C++ bind and run a non-XPLINK 32-bit program
CBCCB	C++ compile and bind a non-XPLINK 32-bit program
CBCCBG	C++ compile, bind, and run a non-XPLINK 32-bit program
CBCXB	C++ bind an XPLINK 32-bit program
CBCXBG	C++ bind and run an XPLINK 32-bit program
CBCXCB	C++ compile and bind an XPLINK 32-bit program
CBCXCBG	C++ compile, bind, and run an XPLINK 32-bit program
CNNPD1B	C or C++ bind an object compiled using the IPA(PDF1) and NOXPLINK options
CNNXP1B	C or C++ bind an object compiled using the IPA(PDF1) and XPLINK options
EDCQB	C bind a 64-bit program
EDCQBG	C bind and run a 64-bit program
EDCQCB	C compile and bind a 64-bit program
EDCQCBG	C compile, bind, and run a 64-bit program
CBCQB	C++ bind a 64-bit program
CBCQBG	C++ bind and run a 64-bit program
CBCQCB	C++ compile and bind a 64-bit program
CBCQCBG	C++ compile and bind a 64-bit program
CBCQCGB	C++ compile, bind and run a 64-bit program
CNNQP1B	C or C++ bind a 64-bit object compiled using the IPA(PDF1) and LP64 options

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 36 on page 367 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 36. Example source files

Steps for single final bind under z/OS batch

Before you begin: Compile each source file.

Perform the following steps to complete a final single bind of everything:

1. Compile each source file to generate the object modules USERID.PLAN9.OBJ(UNIT0), USERID.PLAN9.OBJ(UNIT1), and USERID.PLAN9.OBJ(UNIT2). Use the EDCC procedure as follows:

```

//COMP0 EXEC EDCC,
//      INFILE='USERID.PLAN9.C(UNIT0)',
//      OUTFILE='USERID.PLAN9.OBJ,DISP=SHR',
//      CPARAM='LONG,RENT'
//COMP1 EXEC EDCC,
//      INFILE='USERID.PLAN9.C(UNIT1)',
//      OUTFILE='USERID.PLAN9.OBJ,DISP=SHR',
//      CPARAM='LONG,RENT'
//COMP2 EXEC EDCC,
//      INFILE='USERID.PLAN9.C(UNIT2)',
//      OUTFILE='USERID.PLAN9.OBJ,DISP=SHR',
//      CPARAM='LONG,RENT'

```

2. Perform a final single bind to produce the executable program USERID.PLAN9.LOADE(MYPROG). Use the CBCB procedure as follows:

```

//BIND EXEC CBCB,OUTFILE='USERID.PLAN9.LOADE,DISP=SHR'
//OBJECT DD DSN=USERID.PLAN9.OBJ,DISP=SHR
//SYSIN DD *
INCLUDE OBJECT(UNIT0)
INCLUDE OBJECT(UNIT1)
INCLUDE OBJECT(UNIT2)
NAME MYPROG(R)
/*

```

The OUTFILE parameter along with the NAME control statement specify the name of the output executable to be created.

Advantage

This method is simple, and is consistent with existing methods of building applications, such as makefiles.

Steps for binding each compile unit under z/OS batch

Before you begin: Compile each source file and also bind it.

Perform the following steps to complete a final bind of all the partially bound units:

1. Compile and bind each source file to generate the partially bound program objects USERID.PLAN9.LOADE(UNIT0), USERID.PLAN9.LOADE(UNIT1), and USERID.PLAN9.LOADE(UNIT2), which may have unresolved references. In this example, references to f1() and f4() in USERID.PLAN9.LOADE(UNIT0) are unresolved. Compile and bind each unit by using the EDCCB procedure as follows:

```
//COMP0 EXEC EDCCB,
//      CPARAM='CSECT(MYPROG)',
//      BPARAM='LET,CALL(NO),ALIASES(ALL)',
//      INFILE='USERID.PLAN9.C(UNIT0)',
//      OUTFILE='USERID.PLAN9.LOADE(UNIT0),DISP=SHR'
//COMP1 EXEC EDCCB,
//      CPARAM='CSECT(MYPROG)',
//      BPARAM='LET,CALL(NO),ALIASES(ALL)',
//      INFILE='USERID.PLAN9.C(UNIT1)',
//      OUTFILE='USERID.PLAN9.LOADE(UNIT1),DISP=SHR'
//COMP2 EXEC EDCCB,
//      CPARAM='CSECT(MYPROG)',
//      BPARAM='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.

Steps for building and using a DLL under z/OS batch

Perform the following steps to build USERID.PLAN9.C(UNIT1) and USERID.PLAN9.C(UNIT2) into DLL USERID.PLAN.LOADE(ONETWO), which exports functions f1(), f2(), f3() and f4(). Build USERID.PLAN9.C(UNIT0) into a program which dynamically links to functions f1() and f4() defined in the DLL build and use a DLL under z/OS batch.

1. Compile USERID.PLAN9.C(UNIT1) and USERID.PLAN9.C(UNIT2) to generate the object modules USERID.PLAN9.OBJ(UNIT1) and USERID.PLAN9.OBJ(UNIT2), which define the functions to be exported. Use the EDCC procedure as follows:

```
/* Compile UNIT1
//CC1 EXEC EDCC,
//      CPARM='OPTF(DD:OPTIONS)',
//      INFILE='USERID.PLAN9.C(UNIT1)',
//      OUTFILE='USERID.PLAN9.OBJ(UNIT1),DISP=SHR'
//COMPILE.OPTIONS DD *
        LIST RENT LONGNAME EXPORTALL
*/
/* Compile UNIT2
//CC2 EXEC EDCC,
//      CPARM='OPTF(DD:OPTIONS)',
//      INFILE='USERID.PLAN9.C(UNIT2)',
//      OUTFILE='USERID.PLAN9.OBJ(UNIT2),DISP=SHR'
//COMPILE.OPTIONS DD *
        LIST RENT LONGNAME EXPORTALL
*/
```

2. Bind USERID.PLAN9.OBJ(UNIT1) and USERID.PLAN9.OBJ(UNIT2) to generate the DLL ONETWO:

```

/* Bind the DLL
//BIND1 EXEC CBCB,
//      BPARM='CALL,DYNAM(DLL)',
//      OUTFILE='USERID.PLAN9.LOADE(ONETWO),DISP=SHR'
//INOBJ DD DISP=SHR,DSN=USERID.PLAN9.OBJ
//SYSDEFS 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 SYSDEFS 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 SYSDEFS. One IMPORT control statement is written for each exported symbol. These generated control statements will be included later as input to the bind step of an application that uses this DLL, so that it can import the symbols.

-
3. Compile USERID.PLAN9.C(UNIT0) so that it may import unresolved symbols, and bind with the file of IMPORT control statements from the DLLs build:

```

/* Compile the DLL user
//CC1 EXEC EDCC,
//      CPARM='OPTF(DD:OPTIONS)',
//      INFILE='USERID.PLAN9.C(UNIT0)',
//      OUTFILE='USERID.PLAN9.OBJ(UNIT0),DISP=SHR'
//COMPILE.OPTIONS DD *
        LIST RENT LONGNAME DLL
/*
/* Bind the DLL user with input IMPORT statements from the DLL build
//BIND1 EXEC CBCB,
//      BPARM='CALL,DYNAM(DLL)',
//      OUTFILE='USERID.PLAN9.LOADE,DISP=SHR'
//INOBJ DD DISP=SHR,DSN=USERID.PLAN9.OBJ
//IMP DD DISP=SHR,DSN=USERID.PLAN9.IMP
//SYSLIN DD *
        INCLUDE INOBJ(UNIT0)
        INCLUDE IMP(ONETWO)
        ENTRY CEESTART
        NAME DLL12USR(R)
/*

```

Advantage

The bind time advantage of using DLLs is that you only need to rebuild the DLL with the changed code in it. You do not need to rebuild all applications that use the DLL in order to use the changed code.

| Build and use a 64-bit application under z/OS batch

| Creating a 64-bit application under z/OS Batch is similar to creating a 31-bit application. There are, however, some subtle differences, which the following C++ example demonstrates.

| As of z/OS C/C++ V1R6, new PROCs are available for binding and running with 64-bit applications. There are no new PROCs for a 64-bit compile (without binding

or running) but you can use the previously existing C and C++ PROCs, along with the LP64 compiler option, to create 64-bit object files that can then be used with the new 64-bit enabled PROCs. Then, rather than using the regular binding PROCs (such as CBCB and EDCCBG), you need to use the new 64-bit PROCs for binding; for example, CBCQB and EDCQCBG.

Example: The following example will now show you how to implement the above instructions. In this example, we use the CBCB PROC and the LP64 compiler option for our first 64-bit compile, and the CBCQCBG PROC to compile another source file in 64-bit mode, bind it (along with the first object file we produced), and finally run the resulting load module.

```
#include <iostream>
void lp64_function() {
#ifdef _LP64
    std::cout << "Hello World, z/OS has 64-bit programs now!" << std::endl;
#else
    std::cout << "Uh oh, someone didn't compile this file with LP64" << std::endl;
#endif
}

HELLO2.C
void lp64_function();
int main() {
    lp64_function();
}

//USERID   JOB (641A,2317),'Programmer Name',REGION=128M,
//         CLASS=B,MSGCLASS=S,NOTIFY=&SYSUID;,MSGLEVEL=(1,1)
//ORDER JCLLIB ORDER=(CBC.SCCNPRC)
/*-----
/* C++ Compile using LP64 compiler option
/*-----
//COMPILE EXEC CBCB,
//         INFIL='USERID.LP64.SOURCE(HELLO1)',
//         OUTFILE='USERID.LP64.OBJECT(HELLO1),DISP=SHR',
//         CPARM='OPTFILE(DD:OPTIONS)'
//OPTIONS DD DATA,DLM='>'
//         LP64
//>
/*-----
/* C++ 64-bit Compile, Bind, and Go Step
/*-----
//COBINDGO EXEC CBCQCBG,
//         INFIL='USERID.LP64.SOURCE(HELLO2)',
//         OUTFILE='USERID.LP64.LOAD(HELLO),DISP=SHR'
//BIND.SYSIN DD DATA,DLM='>'
//         INCLUDE OBJECT(HELLO1)
//>
//OBJECT   DD DSN=USERID.LP64.OBJECT,DISP=SHR
```

Build and use a 64-bit application with IPA under z/OS batch

Example: This example shows you how to IPA Compile both a C source file and a C++ source file in 64-bit mode, then IPA Link them, bind them (in 64-bit mode), and run the resulting load module.

This example also shows that when you want to create an IPA optimized program that makes use of calls to standard library functions, you need to explicitly let IPA know where to find the libraries that it will link with. The location of the standard library functions is not included by default in the IPA Link PROCs because if you do not actually ever call a standard library function, IPA will spend time analyzing the unused libraries before realizing your program does not need them, thereby unnecessarily slowing down your compilation time. If you are building a C++ program and do not tell IPA where to find the libraries it needs at IPA Link time, the

IPA Linker will complain about the unresolved symbols it cannot find. You can tell IPA where the standard libraries are by adding the following lines to the CBCXI or EDCXI job steps in your JCL:

```
//SYSIN DD DATA,DLM='>'
INCLUDE OBJECT(HELLO)
INCLUDE SYSLIB(C64,IOSX64)
INCLUDE SYSLIB(CELQSCPP,CELQS003)
/>
//SYSLIB DD DSN=CEE.SCEEEND2,DISP=SHR
// DD DSN=CEE.SCEEELIB,DISP=SHR
// DD DSN=CBC.SCLBSID,DISP=SHR
//OBJECT DD DSN=USER.TEST.OBJECT,DISP=SHR
```

Note: The USER.TEST.OBJECT data set and the HELLO PDS member are meant to represent the object file(s) for your application, which you should have created using a previous IPA Compile step.

Example: The following example will now show you how to implement the above instructions.

```
//USERID JOB (641A,2317),'Programmer Name',REGION=128M,
// CLASS=B,MSGCLASS=S,NOTIFY=&SYSUID;,MSGLEVEL=(1,1)
//ORDER JCLLIB ORDER=(CBC.SCCNPRC)
/*-----
/* 64-bit C IPA Compile
/*-----
//IPACOMP1 EXEC EDCC,
// OUTFILE='USERID.IPA.LP64.OBJECT(OBJECT1),DISP=SHR',
// CPARM='OPTFILE(DD:OPTIONS)'
//SYSIN DD DATA,DLM='>'

#include <time.h>
#include <string.h>
int get_time_of_day(char* output) {

    time_t    time_val;
    struct tm* time_struct;
    char*     time_string;

    if ( -1 != time(&time_val;) ) {

        time_struct = localtime(&time_val;);

        if ( NULL != time_struct ) {

            time_string = asctime(time_struct);

            if ( NULL != time_string ) {

                strcpy(output, time_string);
                output[strlen(output) - 1] = 0;

                return 0;
            }
        }
    }

    return 1;
}
/>
//OPTIONS DD DATA,DLM='>'
IPA(NOOBJECT,NOLINK) LP64 LONGNAME OPT
/>
/*-----
/* 64-bit C++ IPA Compile with very high optimization
/*-----
```

```

//IPACOMP2 EXEC CBCC,
//          OUTFILE='USERID.IPA.LP64.OBJECT(OBJECT2),DISP=SHR',
//          CPARM='OPTFILE(DD:OPTIONS)'
//SYSIN    DD DATA,DLM='>'

#include <iostream>
#include <string>

using std::cout;
using std::endl;
using std::string;

extern "C" int get_time_of_day(char*);

int main() {

    char* tod;

    tod = new char[100];

    if ( 0 == get_time_of_day(tod) ) {

        cout << "The current time is: " << tod << endl;

    } else {

        cout << "Error: Could not determine the time" << endl;

    }

    delete tod;

    return 0;
}
/>
//OPTIONS DD DATA,DLM='>'
IPA(NOOBJECT,NOLINK) LP64 OPT(3)
/>
/*-----
/* 64-bit C++ IPA Link
/*-----
//IPALINK EXEC CBCXI,
//          OUTFILE='USERID.IPALINK.LP64.OBJECT(IPAOBJ),DISP=SHR',
//          IPARM='IPA(LEVEL(2),MAP) LONGNAME'
//SYSIN    DD DATA,DLM='>'
INCLUDE OBJECT(OBJECT1)
INCLUDE OBJECT(OBJECT2)
INCLUDE SYSLIB(C64,IOSX64)
INCLUDE SYSLIB(CELQSCPP,CELQS003)
/>
//SYSLIB DD DSN=CEE.SCEEEND2,DISP=SHR
//          DD DSN=CEE.SCEELIB,DISP=SHR
//          DD DSN=CBC.SCLBSID,DISP=SHR
//OBJECT DD DSN=USERID.IPA.LP64.OBJECT,DISP=SHR
/*-----
/* C++ 64-bit Bind and Go Step
/*-----
//BINDGO EXEC CBCQBG,
//          INFILE='USERID.IPALINK.LP64.OBJECT(IPAOBJ)',
//          OUTFILE='USERID.LP64.LOAD(FINALEXE),DISP=SHR'
//SYSIN    DD DATA,DLM='>'
INCLUDE OBJECT(IPAOBJ)
/>
//OBJECT DD DSN=USERID.IPA.LP64.OBJECT,DISP=SHR

```

Using the non-XPLINK version of the Standard C++ Library and z/OS batch

A non-XPLINK Standard C++ Library DLL is available that provides Standard C++ Library support for CICS and IMS. The CICS and IMS subsystems do not support XPLINK linkage, rendering the XPLINK Standard C++ Library DLL supplied with the compiler inoperable under both subsystems. The non-XPLINK Standard C++ Library DLL allows support for the Standard C++ Library in the CICS and IMS subsystems, as of z/OS V1R2. Since CICS and IMS do not support XPLINK linkage, and there are no plans to support XPLINK linkage, a non-XPLINK DLL enables the Standard C++ Library under these subsystems.

All non-XPLINK C++ PROCs containing bind and pre-link steps need to be overridden in order to use the non-XPLINK Standard C++ Library DLL. These PROCs are: CBCB, CBCBG, CBCCB, CBCCBG, CBCCL, CBCCLG, CBCL, CBCLG and CCNPD1B.

The appropriate DD statements in these PROCs must be overridden:

- For a bind step, the non-XPLINK side deck must override the XPLINK side-deck or the SYSLIN concatenation.
- For a pre-link step, the non-XPLINK side deck must override the XPLINK side deck or the SYSIN concatenation.

The following concatenations added to the calling JCL will override the appropriate DD statement of the corresponding CBC PROC:

CBCB, CBCBG

```
//SYSLIN DD
// DD DSN=&LIBPREFIX..SCEELIB;(C128N),DISP=SHR
```

CBCCB, CBCCBG

```
//BIND.SYSLIN DD
// DD DSN=&LIBPREFIX..SCEELIB;(C128N),DISP=SHR
```

CBCL, CBCLG

```
//SYSLIN DD
// DD DSN=&LIBPREFIX..SCEELIB;(C128N),DISP=SHR
```

CBCCL, CBCCLG

```
//PLKED.SYSLIN DD
// DD DSN=&LIBPREFIX..SCEELIB;(C128N),DISP=SHR
```

The following concatenation added to the calling JCL will override the appropriate DD statement of the corresponding CCN PROC. Note that CICS does not support PDF.

CCNPD1B

```
//SYSLIN DD
// DD DSN=&LIBPREFIX..SCEELIB;(C128N),DISP=SHR
```

Restrictions concerning use of non-XPLINK Standard C++ Library DLL

The following is a list of restrictions:

- **No Enhanced ASCII Functionality Support:**
The non-XPLINK Standard C++ Library DLL does not provide enhanced ASCII functionality support as ASCII run-time functions require XPLINK linkage. Classes and functions sensitive to character encoding are provided in EBCDIC alone in the non-XPLINK DLL.

- No PDF PROC Support for CICS:

CICS does not support Profile Directed Feedback (PDF). The non-XPLINK PDF PROC, CCNPD1B, cannot be used with CICS. The XPLINK PDF CCNXP1B PROC and the 64-bit PDF CCNQPD1B PROC cannot be used with CICS as well.

Steps for rebinding a changed compile unit under z/OS batch

Before you begin: Make a change to a single source file and rebuild the application.

Perform the following steps to recompile the single changed source file and make a replacement of its binder sections in the program:

1. Recompile the single changed source file. Use the compile time option CSECT to ensure that each section is named for purposes of rebinding. For example, assume that you have made a change to USERID.PLAN9.C(UNIT1). Recompile the source file using the EDCC procedure as follows:

```

/* Compile UNIT1 user
//CC EXEC EDCC,
//          CPARM='OPTF(DD:OPTIONS)',
//          INFILE='USERID.PLAN9.C(UNIT1)',
//          OUTFILE='USERID.PLAN9.OBJ(UNIT1),DISP=SHR'
//COMPILE.OPTIONS DD *
//          LIST RENT LONGNAME DLL CSECT(MYPROG)
/*

```

2. Rebind only the changed compile unit into the executable program, which replaces its corresponding binder sections in the program object:

```

//BIND EXEC CBCB,
//          OUTFILE='USERID.PLAN9.LOADE,DISP=SHR'
//OLDPGM DD DSN=USERID.PLAN9.LOADE,DISP=SHR
//NEWOBJ DD DSN=USERID.PLAN9.OBJ,DISP=SHR
//SYSIN DD *
//          INCLUDE NEWOBJ(UNIT1)
//          INCLUDE OLDPGM(MYPROG)
//          NAME NEWPGM(R)
/*

```

Advantage

Rebinds are fast because most of the program is already bound, and none of the intermediate object modules are retained.

Writing JCL for the binder

You can use cataloged procedures rather than supply all the JCL required for a job step. However, you can use JCL statements to override the statements of the cataloged procedure.

Use the EXEC statement in your JCL to invoke the binder. The EXEC statement to invoke the binder is:

```
//BIND EXEC PGM=IEWL
```

Use PARM parameter for the EXEC statement to select one or more of the optional facilities that the binder provides.

Example: You can specify the OPTIONS option on the PARM parameter to read binder options from the ddname OPTS, as follows:

```
//BIND1 EXEC PGM=IEWL,PARM='OPTIONS=OPTS'
//OPTS DD *
        AMODE=31,MAP
        RENT,DYNAM=DLL
        CASE=MIXED,COMPAT=CURR
/*
//SYSLIB DD DISP=SHR,DSN=CEE.SCEELKEX
// DD DISP=SHR,DSN=CEE.SCEELKED
// DD DISP=SHR,DSN=CEE.SCEECPP
//SYSLIN DD DISP=SHR,DSN=USERID.PLAN9.OBJ(P1)
// DD DISP=SHR,DSN=CBC.SCLBSID(IOSTREAM)
//SYSLMOD DD DISP=SHR,DSN=USERID.PLAN9.LOADE(PROG1)
//SYSPRINT DD SYSOUT=*
```

In the example above, object module P1, which was compiled NOXPLINK, is bound using the IOSTREAM DLL definition side-deck. The Language Environment non-XPLINK run-time libraries SCEELKED, SCEELKEX, and SCEECPP are statically bound to produce the program object PROG1.

Example: If the object module P1 was compiled XPLINK, then the JCL would be:

```
//BIND1 EXEC PGM=IEWL,PARM='OPTIONS=OPTS'
//OPTS DD *
AMODE=31,MAP
RENT,DYNAM=DLL
CASE=MIXED,COMPAT=CURR
LIST=NOIMP
/*
//SYSLIB DD DSN=CEE.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=*
```

Example: If the object module P1 was compiled LP64, then the JCL would be:

```
//BIND1 EXEC PGM=IEWL,PARM='OPTIONS=OPTS'
//OPTS DD *
AMODE=64,MAP
RENT,DYNAM=DLL
CASE=MIXED,COMPAT=CURR
LIST=NOIMP
/*
//SYSLIB DD DSN=CEE.SCEEBIND,DISP=SHR
//SYSLIN DD DSN=USRID.PLAN9.OBJ(P1),DISP=SHR
// DD DSN=CEE.SCEELIB(CELQHSCPP),DISP=SHR
// DD DSN=CEE.SCEELIB(CELQHS003),DISP=SHR
// DD DSN=CBC.SCLBSID(IOSX64),DISP=SHR
// DD DISP=SHR,DSN=CBC.SCLBSID(IOSTREAM)
//SYSLMOD DD DISP=SHR,DSN=USERID.PLAN9.LOADE(PROG1)
//SYSPRINT DD SYSOUT=*
```

For more information on the files given, please refer to “LP64 libraries” on page 387.

The binder always requires three standard data sets. You must define these data sets on DD statements with the ddnames SYSLIN, SYSLMOD, and SYSPRINT.

Example: A typical sequence of job control statements for binding an object module into a program object is shown below. In the following non-XPLINK example, the binder control statement NAME puts the program object into the PDSE USER.LOADE with the member name PROGRAM1.

```
//BIND EXEC PGM=IEWL,PARM='MAP'
//SYSPRINT DD * << out: binder listing
//SYSDEFSD DD DUMMY << out: generated IMPORTs
//SYSLMOD DD DISP=SHR,DSN=USERID.PLAN9.LOADE << out: PDSE of executables
//SYSLIB DD DISP=SHR,DSN=CEE.SCEELKED << in: autocal 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.

Note: This REXX EXEC does not support 64-bit binding. You must use the PROCs or c89, cc, c++, or cxx commands under UNIX System Services to perform 64-bit binding.

If you specify a data set name in an option, and the high-level qualifier of the data set is not the same as your user prefix, you must use the fully qualified name of the data set and place single quotation marks around the entire name.

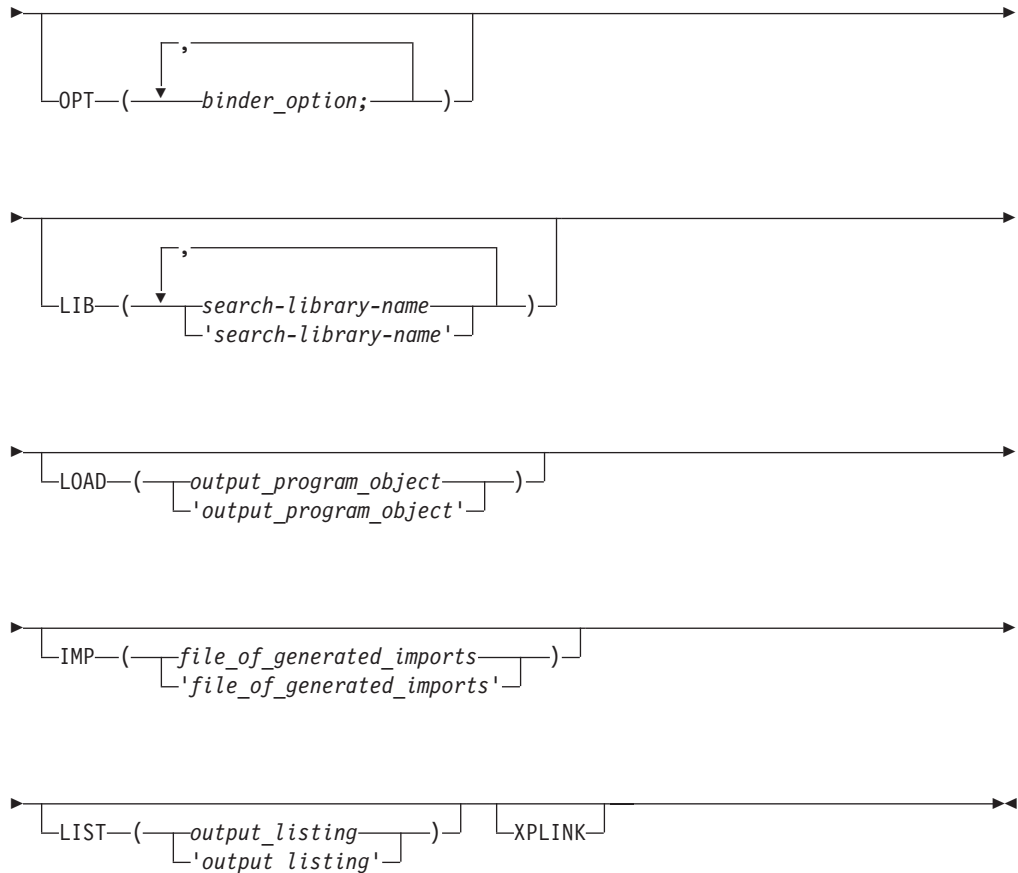
If you specify an HFS filename in an option, it must be an absolute filename; it must begin with a slash (/). You can include commas and special characters in filenames, but you must enclose filenames that contain special characters or commas in single quotes. If a single quote is part of the filename, you must specify the quote twice.

The syntax for the CXXBIND EXEC is:

```

CXXBIND OBJ ( ( [ , ] [ input-object ] [ 'input-object' ] ) )

```



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 ddnameSYSLIB 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 37 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 37. Example Source Files

Steps for single final bind under TSO

Before you begin: Compile each source file.

Perform the following steps to complete a single final bind of everything:

1. Compile each unit to generate the object modules PLAN9.OBJ(UNIT0), PLAN9.OBJ(UNIT1), and PLAN9.OBJ(UNIT2). Use the CC REXX exec as follows:

```
CC PLAN9.C(UNIT0) OBJECT(PLAN9.OBJ) CSECT(MYPROG)
CC PLAN9.C(UNIT1) OBJECT(PLAN9.OBJ) CSECT(MYPROG)
CC PLAN9.C(UNIT2) OBJECT(PLAN9.OBJ) CSECT(MYPROG)
```

2. Perform a final single bind to produce the executable program PLAN9.LOADE(MYPROG). Use the CXXBIND REXX exec as follows:

```
CXXBIND OBJ(PLAN9.OBJ(UNIT0),PLAN9.OBJ(UNIT1),PLAN9.OBJ(UNIT2))
LOAD(PLAN9.LOADE(MYPROG))
```

Advantage

This method is simple, and is consistent with existing methods of building applications, such as makefiles.

Steps for binding each compile unit under TSO

Before you begin: Compile and bind each source file.

Perform the following steps to complete a final bind of all the partially bound units:

1. Compile and bind each source file to generate the partially bound program objects PLAN9.LOADE(UNIT0), PLAN9.LOADE(UNIT1), and PLAN9.LOADE(UNIT2), which may have unresolved references. In this example, references to f1() and f4() in PLAN9.LOADE(UNIT0) are unresolved. Compile and bind each unit by using the CC and CXXBIND REXX execs as follows:

```
CC PLAN9.C(UNIT0) OBJECT(PLAN9.OBJ) CSECT(MYPROG)
CXXBIND OBJ(PLAN9.OBJ(UNIT0)) OPT('LET,CALL(NO)')
LOAD(PLAN9.LOADE(UNIT0))
```

```
CC PLAN9.C(UNIT1) OBJECT(PLAN9.OBJ) CSECT(MYPROG)
CXXBIND OBJ(PLAN9.OBJ(UNIT1)) OPT('LET,CALL(NO)')
LOAD(PLAN9.LOADE(UNIT1))
```

```
CC PLAN9.C(UNIT2) OBJECT(PLAN9.OBJ) CSECT(MYPROG)
CXXBIND OBJ(PLAN9.OBJ(UNIT2)) OPT('LET,CALL(NO)')
LOAD(PLAN9.LOADE(UNIT1))
```

The CALL(NO) option prevents autocall processing.

-
2. Perform the final single bind to produce the executable program MYPROG by using the CXXBIND REXX exec:

```
CXXBIND OBJ(PLAN9.LOADE(UNIT0),PLAN9.LOADE(UNIT1),PLAN9.LOADE(UNIT2))
LOAD(PLAN9.LOADE(MYPROG))
```

Advantage

Binding a set of partially bound program objects into a fully bound program object is faster than binding object modules into a fully bound program object. For example, a central build group can create the partially bound program objects. Developers can then use these program objects and their changed object modules to create a development program object.

Steps for building and using a DLL under TSO

Perform the following steps to build PLAN9.C(UNIT1) and PLAN9.C(UNIT2) into DLL PLAN9.LOADE(ONETWO) which exports functions f1(), f2(), f3() and f4(). Then build PLAN9.C(UNIT0) into a program which dynamically links to functions f1() and f4() defined in the DLL.

1. Compile PLAN9.C(UNIT1) and PLAN9.C(UNIT2) to generate the object modules PLAN9.OBJ(UNIT1) and PLAN9.OBJ(UNIT2) which have functions to be exported. Use the CC REXX exec as follows:

```
CC PLAN9.C(UNIT1) OBJECT(PLAN9.OBJ) EXPORTALL, LONGNAME, DLL, CSECT(MYPROG)
CC PLAN9.C(UNIT2) OBJECT(PLAN9.OBJ) EXPORTALL, LONGNAME, DLL, CSECT(MYPROG)
```

2. Bind PLAN9.OBJ(UNIT1) and PLAN9.OBJ(UNIT2) to generate the DLL PLAN9.LOADE(ONETWO):

```
CXXBIND OBJ(PLAN9.LOADE(UNIT0), PLAN9.LOADE(UNIT1)) IMP (PLAN9.IMP(ONETWO))
LOAD(PLAN9.LOADE(ONETWO))
```

When you bind code with exported symbols, you must specify the binder option DYNAM(DLL). You must also use the CXXBIND IMP option to define the definition side-deck where the IMPORT control statements are to be written.

3. Compile PLAN9.C(UNIT0) so that it may import unresolved symbols, and bind with PLAN9.IMP(ONETWO), which is the definition side-deck containing IMPORT control statements from the DLL build:

```
CC PLAN9.C(UNIT0) OBJECT(PLAN9.OBJ) CSECT(MYPROG), DLL
CXXBIND OBJ(PLAN9.LOADE(UNIT0), PLAN9.IMP(ONETWO)) LOAD(PLAN9.LOADE(DLL12USR))
```

Advantage

The bind time advantage of using DLLs is that you only need to rebuild the DLL with the changed code in it. You do not need to rebuild all applications that use the DLL in order to use the changed code.

Steps for rebinding a changed compile unit under TSO

Before you begin: Make a change to a single source file and rebuild the application.

Perform the following steps to recompile the single changed source file and make a replacement of its binder sections in the program:

1. Recompile the single changed source file. Use the 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 10. Binder processing

You can bind any z/OS C/C++ object module or program object.

Object files with long name symbols, reentrant writable static symbols, and DLL-style function calls require additional processing to build global data for the application. You can always rebind if you don't require this additional processing. You can also re-bind if you used the binder for this additional processing and produced a program object (in other words, you didn't use the prelinker). If you used the prelinker and performed this additional processing, you cannot later rebind. If you have done additional processing and output it to a PDS, you cannot rebind it. For further information, refer to "About prelinking, linking, and binding" on page 10.

Various limits have been increased from the linkage-editor. For example, the V1R6 binder supports variable and function names up to 32767 characters long.

For the Writable Static Area (WSA), the binder assigns relative offsets to objects in the Writable Static Area and manages initialization information for objects in the Writable Static Area. The Writable Static Area is not loaded with the code. Language Environment 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 run-time environment only and are not required by any user code. When these symbols are generated, if the binder option is `DYNAM=DLL` and the definition side-deck is not defined for the binder, the binder issues a message indicating the condition. If you are not building a DLL, you can use `DYNAM=NO` or you can ignore the message; or you can define a dummy side deck for the binder and then ignore the generated side deck.

Note: When using the C++ shell utility, use `-W1,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 `G0FF` option is active, then your `CSECTs` will automatically be named. See "CSECT | NOCSECT" on page 84.

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.

I For C and C++, the default attribute is based on the XPLINK, NOXPLINK, LP64 and
I ILP32 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 408 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 360 and “Rebindability” on page 401.

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 *z/OS MVS Program Management: User's Guide and Reference* for further information on the AUTOCALL control statement. After processing the AUTOCALL statement, if new unresolved symbols are found that cannot be resolved from within the library being processed, the library will not be searched again. To search the library again, another AUTOCALL statement or SYSLIB must indicate the same library.

Final autocall processing (SYSLIB)

The binder performs final autocall processing of DD SYSLIB in addition to incremental autocall. It performs this processing after it completes the processing of DD SYSLIN.

DD SYSLIB defines the libraries of object modules, load modules, or program objects that the binder will search after it processes primary and secondary input.

The binder searches each library (PDS or PDSE) in the DD SYSLIB concatenation in order. The rules for searching for a symbol definition in a PDS or PDSE are as follows:

- If the library contains a C370LIB directory (@@DC370\$ or @@DC390\$) that was created using the C/C++ Object Library Utility, and the directory points to a member containing the definition for the symbol, that member is read.
- If the library has a member or alias with the same name as the symbol that is being searched, that member of the library is read.

You can use the LIBRARY control statement to suppress the search of SYSLIB for certain symbols, or to search an alternate library.

Non-XPLINK libraries

The libraries described here are to be used only for binding non-XPLINK program modules.

For C and C++, you should include CEE.SCEELKEX and CEE.SCEELKED in your DD SYSLIB concatenation when binding your program. Those libraries contain the Language Environment resident routines, which include those for callable services, initialization, and termination. CEE.SCEELKED has the uppercase (NOLONGNAME),

8-byte-or-less versions of the standard C library routines, for example PRINTF and @@PT@C. CEE.SCEELKEX has the equivalent case-sensitive long-named routines; for example, printf, pthread_create.

For C++, you should also include the C++ base library in data set CEE.SCEECPP in your DD SYSLIB concatenation when binding your program. It contains the C++ base routines such as global operator new.

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 CEE.SCEELIB side decks reside in the CEE.SCEELIB data set.

The Language Environment routine definitions for callable services are contained in the CELHS001 member of the data set CEE.SCEELIB. For example, CEEGTST is contained here.

The C run-time library routine definitions for 32-bit programs are contained in the CELHS003 member of the data set CEE.SCEELIB, which contains NOLONGNAME and case-sensitive long-named routines (for example, printf, PRINTF, and pthread_create are contained here). It also contains the C run-time library global variables; for example, environ.

For 32-bit C++ programs, you should also include the C++ base library side deck (member CELHSCPP in data set CEE.SCEELIB). It contains the C++ base routines such as global operator new.

LP64 libraries

The libraries described below are to be used only for binding LP64 program modules. LP64 is built upon the XPLINK linkage so the basic elements mentioned above still apply:

- As described above, in the simple XPLINK case, you must include CEE.SCEEBND2 in your DD SYSLIB concatenation when binding your programs.
- The 64-bit C++ libraries are packaged as DLLs, so INCLUDE statements must be used to resolve C and C++ runtime references.
- The 64-bit side decks are in the CEE.SCEELIB dataset.
- For 64-bit modules, the C run-time library definitions are contained in the CELQS003 member of the CEE.SCEELIB data set.
- The C++ base library side deck member for 64-bit is the CELQSCPP member of the CEE.SCEELIB data set.
- The C++ class libraries are contained in the C64 member of the CEE.SCEELIB data set.
- There is no 64-bit equivalent for the CELHS001 member.

Rename processing

Rename processing is performed at the end of the first pass of final autocall processing of DD SYSLIB, when all possible references have been resolved with the names as they were on input. The binder renaming logic permits the conversion of unresolved non-DLL external function references and drives the final autocall process again.

The binder maps names according to the following hierarchy:

1. If the name has ever been mapped due to a pragma map in C++ code, the name is not renamed.
2. If the name has ever been mapped due to a pragma map in C code that was compiled with the LONGNAME option, the name is not renamed.
3. If a valid RENAME control statement was read for an unresolved function name, new-name specified on the applied RENAME statement is chosen, provided that old-name did not already appear on an applied RENAME statement as either a new or old name. Syntactically correct RENAME control statements that are not applied are ignored. See *z/OS MVS Program Management: User's Guide and Reference* for more information on RENAME control statements.
4. If the name corresponds to a Language Environment function, the binder may map the name according to C/C++ run-time library rules.
5. If the UPCASE(YES) option is in effect and the name is 8 bytes or less, and not otherwise renamed by any of the previous rules, the name chosen is the same name but with all alphabetic characters mapped to uppercase, and '_' mapped to '@'. The binder maps names with the initial characters IBM, CEE, or PLI to initial characters of IB\$, CE\$, and PL\$, respectively. All names that are different only in case will map to the same name.

If renamed, the original name is replaced. The original name and the generated new name appear in the rename table of the binder listing. See "Renamed Symbol Cross Reference" on page 394.

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 *z/OS MVS Program Management: User's Guide and Reference* for more information on the DYNAM(DLL) binder option. You must also specify CASE(MIXED) in order to preserve the case sensitivity of symbols on IMPORT control statements.

If you are building an application that imports symbol definitions from a DLL, you must include an IMPORT control statement for each symbol to which your application expects to dynamically link. Typically, the input to your bind step for your application should include the definition side-deck of IMPORT control statements that the binder generated when the DLL was built. For compatibility, the binder accepts definition side-decks of IMPORT control statements that the Language Environment Prelinker generated. To use the definition side-decks that are distributed with IBM Class libraries, you must specify the binder option CASE(MIXED).

After final autocall processing of DD SYSLIB is complete, all DLL-type references that are not statically resolved are compared to IMPORT control statements. Symbols on IMPORT control statements are treated as definitions, and cause a matching unresolved symbol to be considered dynamically rather than statically resolved. A dynamically resolved symbol causes an entry in the binder class B_IMPEXP to be created. If the symbol is unresolved at the end of DLL processing, it is not accessible at run time.

Addresses of statically bound symbols are known at application load time, but addresses of dynamically bound symbols are not. Instead, the run-time library that loads the DLL that exports those symbols finds their addresses at application run time. The run-time library also fixes up the linkage blocks (descriptors) for the importer in C_WSA during program execution.

The binder builds tables of imported and exported symbols in the class B_IMPEXP, section IEWBCIE. This element contains the necessary information about imported and exported symbols to support run-time library dynamic linking and loading.

Statically bound functions

For each DLL-referenced function, the binder will generate a function linkage block (descriptor) of the same name as a part in the class C_WSA.

Some of the linkage descriptors for XPLINK code are generated by the compiler rather than the binder. Compiler-generated descriptors are not visible as named entities at bind time. For XPLINK:

- Functions, which are referenced exclusively in the compilation unit, have descriptors which are generated by the compiler and have no visible names.
- Functions, which are possibly referenced outside of the compilation unit (either by function pointer, or because they are exported), have descriptors which are generated by Language Environment when the DLL is loaded. They are not part of C_WSA. There will be a pointer to the function descriptor in C_WSA.

- For all other DLL-referenced functions, function descriptors are generated by the binder as a part with the same name in the class C_WSA (with the exception that for NORENT compiles, the descriptor will be in B_DESCR rather than C_WSA).

All C++ code and XPLINK code generate DLL references. C code generates DLL references if you used the DLL compiler option. If a DLL reference to an external function is resolved at the end of final autocall processing, the binder generates a function linkage block of the same name in the Writable Static Area, and initializes it to point to the resolved function. If the DLL reference is to a static function, the binder generates a function linkage block with a private name, which is initialized to point to the resolved static function.

Imported variables

For each DLL-referenced external variable in C_WSA that is unresolved at the end of final autocall processing (DD SYSLIB), if a matching IMPORT control statement was read in, the variable is considered to be resolved via dynamic linking from the DLL named on the IMPORT control statement. The binder will generate a variable linkage block (descriptor) of the same name, as a part in the class C_WSA.

Imported functions

For each DLL-referenced external function that is unresolved at the end of final autocall processing, if a matching IMPORT control statement was read in, the function is considered to be resolved via dynamic linking from the DLL named on the IMPORT control statement. The binder will generate a function linkage block (descriptor) of the same name, as a part in the class C_WSA.

Output program object

The DD SYSLMOD defines where the binder stores its output program object. You can store the output program object in one of the following:

- A PDSE member, where the binder stores a single program object
- A PDSE where the binder stores its output program objects (one program object for each NAME control statement)
- 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 39 on page 392. Figure 38 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 38. 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 39. 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:

```

| z/OS V1 R6 BINDER          09:08:20 WEDNESDAY NOVEMBER 11, 2003
| 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 *z/OS MVS Program Management: User's Guide and Reference* for more information on the LIST option.

```

IEW2278I B352 INVOCATION PARAMETERS - AMODE=31,MAP,RENT,DYNAM=DLL,CASE=MIXED,
COMPAT=CURR,ALIASES=ALL,LIST(ALL),MAP,XREF
IEW2322I 1220 1 INCLUDE INOBJ(CU1)
IEW2308I 1112 SECTION CEESTART HAS BEEN MERGED.
IEW2308I 1112 SECTION PROJ9#CU1#C HAS BEEN MERGED.
IEW2308I 1112 SECTION ALongNamedThingVVWhichIsExported HAS BEEN MERGED.
IEW2308I 1112 SECTION Ax HAS BEEN MERGED.
IEW2308I 1112 SECTION PROJ9#CU1#S HAS BEEN MERGED.
IEW2308I 1112 SECTION CEEMAIN HAS BEEN MERGED.
IEW2308I 1112 SECTION PROJ9#CU1#T HAS BEEN MERGED.
IEW2322I 1220 2 INCLUDE '/u/userid/plan9/cu2.o'
IEW2308I 1112 SECTION PROJ9#cu2.C#C HAS BEEN MERGED.
IEW2308I 1112 SECTION PROJ9#cu2.C#S HAS BEEN MERGED.
IEW2308I 1112 SECTION PROJ9#cu2.C#T HAS BEEN MERGED.
IEW2322I 1220 3 IMPORT CODE 'DLLC' 'c1'
IEW2322I 1220 4 IMPORT CODE 'DLLC' 'c2'
IEW2322I 1220 5 IMPORT CODE 'DLLC' 'c3__Fv'
IEW2322I 1220 6 RENAME 'call_a3' 'A3'
IEW2322I 1220 7 RENAME 'call_b1_in_cu2' 'b1'
IEW2322I 1220 8 ENTRY CEESTART
IEW2322I 1220 9 NAME CU1CU2(R)
:
:

```

Module Map

The Module Map is printed only if you specify the binder MAP option. It displays the attributes of each loadable binder class, along with the storage layout of the parts in that class.

For C/C++ programmers who use constructed reentrancy, two classes are of special interest: C_CODE and C_WSA. For LP64, the class names are C_CODE64 and C_WSA64. The C_CODE class exists if C++ code is encountered or if C code is compiled with LONGNAME or RENT. The C_WSA class exists if any defined writable static objects are encountered.

*** M O D U L E M A P ***

 CLASS C_CODE LENGTH = 5E4 ATTRIBUTES = CAT, LOAD, RMODE=ANY

SECTION OFFSET	CLASS OFFSET	NAME	TYPE	LENGTH	----- DDNAME	SOURCE 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 388. 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.

CROSS - REFERENCE TABLE

TEXT CLASS = C_CODE

REFERENCE				TARGET		
CLASS	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 their 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 398.

In the example below, you can see that c2 and c3 are to be dynamically linked from a DLL named DLLC. Also, this program exports variables Ax and ALongNamedThingVVWhichIsExported, and functions A3 and ALongNamedThingFFWhichIsExported.

```

*** I M P O R T E D   A N D   E X P O R T E D   S Y M B O L S   ***

IMPORT/EXPORT      TYPE      NAME              MEMBER
-----
IMPORT             CODE      c2                DLLC
IMPORT             CODE      c3()              DLLC

EXPORT             DATA     Ax
EXPORT             CODE     ALongName-ported
EXPORT             DATA     ALongName#000001
EXPORT             CODE     A3

*** END OF IMPORT/EXPORT ***

```

Mangled to Demangled Symbol Cross Reference

The mangled to demangled 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

__javC1s1-ension           :=
__javC1s18_java/awt/Dimension
  $$DEMANGLED$$           ==    java.awt.Dimension

__javC1s1-nuItem          :=
__javC1s17_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		00000000	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

***** END OF REPORT *****

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 ddnames, and do not appear in this table. The following is the format of the DDname vs Pathname Cross Reference Table.

```

+++++
| D D N A M E   V S   P A T H N A M E   C R O S S   R E F E R E N C E |
+++++

      DDNAME      PATHNAME
      -----
      /0000001   /u/userid/plan9/cu2.o

```

*** END OF DDNAME VS PATHNAME ***

Message Summary Report

The binder generates a message summary report at the conclusion of each bind operation. The summary contains information on the types and severity of the messages that were issued during the bind process. You can search other parts of the listing to find where the messages were issued.

```

-----
MESSAGE SUMMARY REPORT
-----
SEVERE MESSAGES      (SEVERITY = 12)
NONE

ERROR MESSAGES      (SEVERITY = 08)
NONE

WARNING MESSAGES    (SEVERITY = 04)
NONE

INFORMATIONAL MESSAGES (SEVERITY = 00)
2008 2278 2308 2322

**** END OF MESSAGE SUMMARY REPORT ****

```

Binder processing of C/C++ object to program object

The binder recognizes C/C++ object modules and performs special processing for them.

C/C++ categorizes reentrant programs as natural or constructed. The binder supports both natural reentrancy and C/C++ constructed reentrancy. However, programs that contain constructed reentrancy need additional run-time library for support while executing.

C code is naturally reentrant if it contains no data in the Writable Static Area. Modifiable data can be one of the following:

- External variables
- Static variables
- Writable strings
- DLL linkage blocks (descriptors) for variables
- DLL linkage blocks (descriptors) for functions

C++ code always has DLL type references for all function references that require a function descriptor in C_WSA. This means that all C++ programs are made reentrant via constructed reentrancy.

Programs with constructed reentrancy have two areas:

- A modifiable area that contains modifiable objects, seen in the binder class C_WSA
- A constant or reentrant area that contains executable code and constant data, seen in the binder classes B_TEXT or C_CODE.

Each user running the program receives a private copy of the C_WSA demand load class, which is mapped by the binder and is loaded by the run-time library. Multiple spaces or sessions can share the second part only if it is installed in the link pack area (LPA) or extended link pack area (ELPA). You must install PDSEs dynamically in the LPA.

To generate reentrant C/C++ code, follow these steps:

1. Compile your source files to generate code with constructed reentrancy as follows:
 - Compile your C source files with the RENT compiler option to generate code with constructed reentrancy.

- Compile your C++ source files with whatever options you require. The compiler will generate C++ code with constructed reentrancy.
2. Use the binder to combine all input object modules into a single output program object.

Each compile unit maps to a number of sections, which belong to the C_CODE, C_WSA, or B_TEXT binder classes. Named binder sections may be replaced and make the code potentially rebinding. You can name your C/C++ sections with either the CSECT compiler option, or with the use of the #pragma csect directive. The name of a section should not be the same as one of your functions or variables, as this will cause duplicate symbols.

Each section owns one or more parts. The names of the parts are the names that resolve references. The names of functions appear as labels, which also resolve references. Some parts that are owned by a section may be unnamed. Each part belongs to a binder class.

Each externally named object in the Writable Static Area appears as a part that is owned by a section of the same name in the program object. Such parts belong to the C_WSA binder class. The binder section that owns an object also owns the initialization information for the object in the Writable Static Area. A rebind replaces this initialization information.

The code parts belong to the binder class of C_CODE or B_TEXT. The code parts consist of assembly instructions, constants and literals, and potentially read only variables that are not in the Writable Static Area. The following example will produce two sections, i and CODE1:

```
#pragma code(csect,"CODE1")
int i=10;
int foo(void) { return i; }
```

- The section named i is in class C_WSA, and has associated with it the initialization information to initialize 'i' to 10.
- The section named CODE1 is in class C_CODE, and has associated with it the entry point for function foo() and the machine instructions for the function.

When rebound, both sections i and CODE1 are replaced along with any information that is associated with them.

The names in the C_WSA class and in the C_CODE class are in the same namespace. A variable and a function cannot have the same name.

C++ constructor calls and destructor calls that need to be collected across compile units are collected in the class C_@@STINIT.

DLL initialization information, which needs to be collected across compile units, is collected in the class C_@@DLLI.

Note: The information in this section is applicable to GOFF object modules and is not applicable to XOBJ.

Rebindability

If the binder processes duplicate sections, it keeps **only the first one**. This feature is particularly important when rebinding. You must include the changed parts first and the old program object second. This is how you replace the changed sections.

The binder can process each object module separately so that you only need to recompile and rebind the modules that you have modified. You do not need to recompile or include the object module for any unchanged modules.

When the binder replaces a named section, it also replaces all of its parts (named or unnamed). If a section does not have the name you desire, you can change it with the `#pragma csect` directive or with the `CSECT` compiler option. Unnamed parts typically come from the following:

- Unnamed modifiable static parts in `C_WSA` (static variables, strings)
- Unnamed static parts in `C_CODE` that may not be modifiable (static variables, strings)
- Unnamed code, static, or test part in `C_CODE`

You should name all sections if you want to rebind. If a section is unnamed (has a private name) and you attempt to replace it on a rebind, the unnamed section is not replaced by the updated but corresponding unnamed section. Instead, the binder keeps both the old and new unnamed sections, causing the program module to grow in size. All references to functions that are defined by both the old section and the new section are resolved first to functions in the new section. The program may run correctly, but you will get warnings about duplicate function definitions at bind time. These duplicates will never go away on future rebinds because you cannot replace or delete unnamed sections. You will also accumulate dead code in the duplicate functions which can never be accessed. This is why it is important to name all sections if you want to rebind your code.

Example: Suppose that our DLL consists of two compile units, `cu3.c` and `cu4.c`, that are bound using the JCL in Figure 40:

```
/* 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 40. 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,
//      BPARAM='LIST(ALL),CALL,XREF,LET,MAP,DYNAM(DLL)',
//      OUTFILE='USERID.PLAN9.LOADE,DISP=SHR'
//INOBJ DD DISP=SHR,DSN=USERID.PLAN9.OBJ
//INPOBJ DD DISP=SHR,DSN=USERID.PLAN9.LOADE
//SYSLIN DD *
INCLUDE INOBJ(CU3)
INCLUDE SYSLMOD(BADEXE)
ENTRY CEESTART
NAME GOODEXE(R)
/*

```

The input event log in the binder listing shows:

```

IEW2322I 1220 1 INCLUDE INOBJ(CU3)
IEW2308I 1112 SECTION CODE3 HAS BEEN MERGED.
IEW2308I 1112 SECTION int3 HAS BEEN MERGED.
IEW2322I 1220 2 INCLUDE INPOBJ(BADEXE)
IEW2308I 1112 SECTION CODE4 HAS BEEN MERGED.
IEW2308I 1112 SECTION int4 HAS BEEN MERGED.
IEW2308I 1112 SECTION CEESTART HAS BEEN MERGED.
IEW2308I 1112 SECTION CEESG003 HAS BEEN MERGED.
IEW2308I 1112 SECTION CEEBETBL HAS BEEN MERGED.
IEW2308I 1112 SECTION CEEBPUBT HAS BEEN MERGED.
IEW2308I 1112 SECTION CEEBTRM HAS BEEN MERGED.
IEW2308I 1112 SECTION CEEBLLST HAS BEEN MERGED.
IEW2308I 1112 SECTION CEEBINT HAS BEEN MERGED.
IEW2308I 1112 SECTION CEETGTFN HAS BEEN MERGED.
IEW2308I 1112 SECTION CEETLOC HAS BEEN MERGED.
IEW2322I 1220 3 ENTRY CEESTART
IEW2322I 1220 4 NAME GOODEXE(R)

```

BADEXE defines sections int3, CODE3, int4, and CODE4. If the binder sees duplicate sections, it uses the first one that it reads. Since CU3 defines sections CODE3 and int3, and is included before BADEXE, both sections are replaced by the newer ones in CU3 when program object GOODEXE is created.

DLL considerations

Any IMPORT control statements used in the original bind must also be input to the re-bind, unless the dynamic resolution information is available via an INCLUDE statement.

Error recovery

This 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 388. Such inconsistencies arise from inconsistent usage of the `LONGNAME` and `NOLONGNAME` compiler options, and from multi-language programs that make symbol names uppercase.

Example: Compile the file `main.c` with the options `LONG`, `NORENT`, and `other.c` with the options `NOLONG`, `RENT`:

```
/* file: main.c */
/* compile with LONG, NORENT */
extern int I2;
extern int func2(void);
main() {
    int i;
    i = i2 + func2();
    return i;
}
/* file: other.c */
/* compile with NOLONG,RENT */
int I2 = 2;
int func2(void) { return 2; }
```

When you bind the object modules together, the following errors will occur:

- An inconsistent use of the `RENT` | `NORENT` C compiler option causes symbol `I2` to be unresolved. The definition of `I2` from `other.c` is a writable static object because of the `RENT` option. But a writable static object cannot resolve the reference to `I2` from `main.c` because it is a `NORENT` reference. The binder messages show:

```
IEW2308I 1112 SECTION I2 HAS BEEN MERGED.
```

```
IEW2456E 9207 SYMBOL I2 UNRESOLVED.
```

- An inconsistent use of the `LONG` | `NOLONG` C compiler option causes the symbol `func2` to be unresolved. The function definition in `other.c` is in uppercase because of the `NOLONG` option. But the reference to `func2` from `main.c` is in lowercase because of the `LONG` option. The binder listing shows that '`FUNC2`' is a LABEL, that is a defined entry point; yet the binder messages show:

```
IEW2456E 9207 SYMBOL func2 UNRESOLVED.
```

Significance of library search order

The order in which the libraries in `SYSLIB` are concatenated is significant.

Example: Suppose that functions `f1()` and `f4()` are resolved from `SYSLIB`:

```
/* file: unit0.c */
extern int f1(void); /* from member UNIT1 of library LIB1 */
extern int f4(void); /* from member UNIT2 of library LIB2 */
int main() {
    int rc1, rc4;
    rc1 = f1();
    rc4 = f4();
    if (rc1 != 1) printf("fail rc1 is %d-n", rc1);
    if (rc4 != 40) printf("fail rc1 is %d-n", rc4);
    return 0;
}
```

`SYSLIB` defines the libraries `USERID.LIB1` with members `UNIT1` and `UNIT2`, and `USERID.LIB2` with members of the same name but different contents.

The library members are compiled from the following:

```
/* member UNIT1 of library LIB1 */
int f1(void) { return 1; }

/* member UNIT2 of library LIB1 */
int f2(void) { return 2; }

/* member UNIT1 of library LIB2 */
int f1(void) { return 10; }

/* member UNIT2 of library LIB2 */
int f2(void) { return 20; }
int f3(void) { return 30; }
int f4(void) { return f2()*2; /* 40 */ }
```

When bound with ALIASES(ALL), or when the EDCALIAS utility is used, all defined symbols are seen in a library directory as aliases that indicate the library member that contains their definition.

There are two definitions of f1(), but library search of SYSLIB for f1 searches library LIB1 first, and finds alias f1 of member UNIT1. It reads in that member, and the call to f1() returns 1. Library search of SYSLIB for f4 searches LIB1 first, and does not find a definition. It then searches LIB2, and finds alias f4 of member UNIT2 of library LIB2. So UNIT2 of library LIB2 is read in resolving not only f4, but also f2 and f3, and the call to f4() returns 40. UNIT2 of library LIB1 is not read by mistake because an alias indicates not only the member name, but also the library in which that member resides.

If the order of LIB1 and LIB2 is reversed, LIB2 is searched first, and f1() is obtained from LIB2 instead.

If changing the library search order cannot work for you, use the LIBRARY control statement. See *z/OS MVS Program Management: User's Guide and Reference* for further information on the LIBRARY control statement.

Duplicates

If the binder processes duplicate sections, it keeps the first one and ignores subsequent ones, without giving a warning. This feature is used to replace named sections when rebinding by replacing only changed sections.

If the binder processes functions that have duplicate names, it keeps all definitions, but all references resolve to the first one. An exception is in the case of C++ template instantiation. The binder takes the first user-defined function (if any) of the same signature rather than the first compiler-generated definition via template instantiation.

Example: Compile the following source files doit1.c and doit2.c:

```
#include <stdio.h>
/* file: doit1.c */
int int1 = 1;
#pragma csect(code,"D01")
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,"D02")
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 D01.

```

It is easier to find the object code with the duplicate if you use multiple INCLUDE statements rather than DD concatenation.

Example: If you use:

```

//INOBJ DD DISP=SHR,DSN=USERID.PLAN9.OBJ
//SYSLIN DD *
INCLUDE INOBJ(DOIT1)
INCLUDE INOBJ(DOIT2)
ENTRY CEESTART
/*

```

The members in the binder listing are separated logically. The messages in the binder listing are:

```

:
:
IEW2322I 1220 1 INCLUDE INOBJ(DOIT1)
IEW2308I 1112 SECTION CEESTART HAS BEEN MERGED.
IEW2308I 1112 SECTION D01 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 D01.
IEW2308I 1112 SECTION D02 HAS BEEN MERGED.

```

From the informational messages, it is clear that section DO1 is from INOBJ(DOIT1), and that DO2 is from INOBJ(DOIT2).

Example: But if you use DD concatenation as follows:

```

//INOBJ DD DISP=SHR,DSN=USERID.PLAN9.OBJ
//SYSLIN DD DISP=SHR,DSN=USERID.PLAN9.OBJ(DOIT1)
// DD DISP=SHR,DSN=USERID.PLAN9.OBJ(DOIT2)
// DD *
ENTRY CEESTART
/*
:
:

```

Now the messages are:


```

IEW2308I 1112 SECTION CEESTART HAS BEEN MERGED.
IEW2308I 1112 SECTION D01 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 D01.
IEW2308I 1112 SECTION D02 HAS BEEN MERGED.

```

It is no longer clear which input file defines which section, and this makes tracking down duplicates to the originating compile unit more difficult.

Duplicate functions from autocall

If a library member that is expected to contain the definition of a symbol is read, it may resolve the expected symbol. It may also resolve other symbols because the library member may define multiple functions. These unexpected definitions that are pulled in through library search may cause duplicates. Since you cannot always be sure which one of the duplicate symbols you will resolve with, you should remedy the situation that is causing the duplicate symbols.

Hunting down references to unresolved symbols

Unresolved requests generate error or warning messages in the binder listing. If a function or variable is unresolved at the end of binder processing, it can be resolved at a later rebind.

Example: If you did not expect a symbol to remain unresolved, you can look at the binder listing to see which parts reference the symbol. If your DD SYSLIN has a large concatenation, the input is logically concatenated before the binder processes it. Since the compile units are not logically separated, it is hard to tell which compile unit defines the part that has the reference; for example:

```

//SYSLIN DD DISP=SHR,DSN=USERID.PLAN9.OBJ(MEM1)
//      DD DISP=SHR,DSN=USERID.PLAN9.OBJ(MEM2)
//      DD DISP=SHR,DSN=USERID.PLAN9.OBJ(MEM3)

```

Example: You should consider using multiple INCLUDE control statements, which will logically separate the compile units for the binder informational messages in the listing. You can then find the compile unit with the unresolved reference (similar to finding duplicate function definitions); for example:

```

//INOBJ DD DISP=SHR,DSN=USERID.PLAN9.OBJ
//SYSLIN DD *
INCLUDE INOBJ(DOIT1)
INCLUDE INOBJ(DOIT2)
ENTRY CEESTART
/*

```

Incompatible linkage attributes

The binder will check that a statically bound symbol reference and symbol definition have compatible attributes. If a mismatch is detected, the binder will issue a diagnostic message. This attribute information is contained within the binder input files, such as object files, program objects, and load modules.

For C and C++, the default attribute is based on the XPLINK and NOXPLINK options. Individual symbols can have a different attribute than the default by using the #pragma OS_UPSTACK, #pragma OS_DOWNSTACK, and #pragma OS_NOSTACK.

The attributes can also be set for assembly language. Refer to the *HLASM Language Reference*, SC26-4940 for further information.

Non-reentrant DLL problems

If you bind a DLL with the 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 11. 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 run-time 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. As of z/OS V1R2, 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 411.
- On the GPARM parameter of the cataloged procedures that are supplied by IBM. Refer to “Using cataloged procedures” on page 411.
- 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 on the assembler user exit, 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 287.

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 CEELOPT macro and linking this csect into the program module.
- Specifying run-time options in the EXEC or GPARM statements

Example: The following example shows you how to run your program under z/OS batch. Partitioned data set member `MEDICAL.ILLNESS.LOAD(SYMPTOMS)` contains your z/OS 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 41. Running your program under z/OS batch

Specifying run-time options in the EXEC statement

Example: You can specify run-time options in the PARM parameter of the EXEC statement as follows:

```
//[stepname] EXEC PGM=program_name,
//          PARM='[runtime options/][program parameters]'
```

Example: If you want to generate a storage report and run-time options report for the application `PROGRAM1`, specify the run-time option `RPTOPTS(ON)` as follows:

```
//G01 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
EDXCBCG	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 591.

Example: If you are using an IBM-supplied cataloged procedure, you must specify the run-time options on the GPARM parameter of the EXEC statement. Ensure that the EXECOPS run-time option is in effect.

```
//STEP EXEC EDCCBG,INFILE='...',
//      GPARM='STACK(10K)'
```

Example: You can also use the GPARM parameter to pass arguments to the z/OS C/C++ main() function. Place the argument, preceded by a slash, after the run-time options; for example:

```
//GO EXEC EDCCBG,INFILE=...,
//      GPARM='STACK(10K)/ARGUMENT'
```

Example: If you want to pass an argument without specifying run-time options and EXECOPS is in effect (this is the default), precede it with a slash; for example:

```
//GO EXEC EDCCBG,...GPARM='/ARGUMENT'
//GO EXEC EDCCBG,...GPARM='/HFS file:/u/mike/cloudy.C'
```

Example: If you want to pass parameters which contain slashes, and you are not providing run-time options, you must precede the parameters with a slash, as follows:

```
//GO EXEC EDCCBG,...GPARM='/HFS file:/u/mike/cloudy.C'
```

See also “Specifying run-time options” on page 287.

Running an application under TSO

Before you run your program under TSO, you must have access to the run-time library CEE.SCEERUN. To ensure that you have access to the run-time library, do one of the following:

- If you are running under ISPF in the foreground, concatenate the libraries to ISPLLIB.
- Have your system programmer add the libraries to the LPALST or LPA.
- Have your system programmer add the libraries to the LNKLST.
- Have your system programmer change the LOGON PROC so the libraries are added to the STEPLIB for the TSO session.
- If your application was compiled using the XPLINK compiler option, you must have the Language Environment Library SCEERUN2 available before you try to run your application under TSO.

The TSO CALL command runs a load module under TSO. If *data-set-name* is the partitioned data set member that holds the load module, the command to load and run a specified load module is:

```
CALL 'data-set-name' ['parameter-string'];
```

For example, if the load module is stored in partitioned data set member SAMPLE.CPGM.LOAD(TRICKS), and the default run-time options are in effect, run your program as follows:

```
CALL 'SAMPLE.CPGM.LOAD(TRICKS)'
```

If you specify the unqualified name of the data set, the system assumes the descriptive qualifier LOAD. If you do not specify a member name, the system assumes the name TEMPNAME.

You do not need to use the CALL command if the STEPLIB ddname includes the data set that contains your program. For example, you could call a program PROG1 with two required parameters PARM1 and PARM2 from the command line:

```
PROG1 PARM1 PARM2
```

See the appropriate 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 287.

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 73 or the description in *z/OS 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 33. 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 System Services

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 System Services. If your application was compiled using the XPLINK compiler option you must have the Language Environment Library SCEERUN2 available before you try to run your application under z/OS UNIX System Services. If your application was bound with the DLL Class Libraries, you must supply SCLBDLL and/or SCLBDLL2 at run time. The OS/390 V2R10 version of the DLL library is in CBC.SCLBDLL. As of z/OS V1R2, the version of the DLL library is in CBC.SCLBDLL2.

z/OS UNIX System Services Application environments

You can run your z/OS UNIX System Services 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 System Services

When invoking a program from the z/OS shell, slash-separated run-time options arguments syntax is not used. All the arguments always go to the `main()` routine. Specify run-time options by using the exported environment variable `_CEE_RUNOPTS`. The run-time will only use `_CEE_RUNOPTS` if the `EXECOPS` option is in effect.

Restriction on using 24-bit AMODE programs

You cannot run a 24-bit AMODE z/OS C/C++ application program that resides in an HFS file. Any programs you intend to run from the file system must be 31-bit or 64-bit AMODE, problem program state, PSW key 8 programs. If you plan to run a 24-bit AMODE z/OS 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 or 64-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 `0PUTX` 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 `0GETX` 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 Chapter 18, “c89 — Compiler invocation using host environment variables,” on page 471. 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 the shell. Just precede the `CALL` with `tso` on the command line and press the `ENTER` key.

When the program completes, the shell session is restored.

Running z/OS UNIX System Services applications under z/OS batch

Using the BPXBATCH utility

Use the IBM-supplied `BPXBATCH` program to run a 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 17, “BPXBATCH Utility,” on page 467.

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 42 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 42. 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, write error messages to the file /myap/std/my.err, and code the JCL statements as follows:

```
//jobname JOB ...
//stepname EXEC PGM=BPXBATCH,PARM='PGM /u/myu/myap/base_comp'
//STDOUT DD PATH='/u/myu/myap/std/my.out',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU
//STDERR DD PATH='/u/myu/myap/std/my.err',
// PATHOPTS=(OWRONLY,OCREAT,OTRUNC),PATHMODE=SIRWXU
```

Submitting a non-HFS z/OS UNIX System Services 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 12, “Object Library Utility,” on page 421
- Chapter 13, “Filter Utility,” on page 433
- Chapter 14, “DSECT Conversion Utility,” on page 439
- Chapter 15, “Coded Character Set and Locale Utilities,” on page 453

Chapter 12. 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 (also called Object Library Utility directories) provide convenient packaging of object modules in MVS data sets, in much the same way as the UNIX System Services (USS) utility archives packaged object modules that reside in HFS files. Using the Object Library Utility, you can create libraries that contain object modules compiled with various combinations of compiler options, such as LONGNAME/NOLONGNAME, XPLINK/NOXPLINK, IPA/NOIPA and LP64/ILP32.

The Object Library Utility keeps track of the attributes of each of its members in two special members of the library, which are the Basic Directory Member (@@DC370\$) and the Enhanced Directory Member (@@DC390\$). The Basic Directory Member is used to maintain backwards compatibility with Object Library Utility directories that were created with older versions of the Object Library Utility (pre-z/OS V1R2). The Enhanced Directory Member was introduced to support object modules that were compiled with the IPA, XPLINK, or LP64 compiler options, as well as provide more detailed listing information. If you do have older Object Library Utility directories at your site, you should consider upgrading them to include the Enhanced Directory Member by using the DIR command (described later in this chapter).

Commands for this utility allow you to add and delete object modules from a library, rebuild the Basic and Enhanced Directory Members, and to create a listing of all the contents in a Object Library Utility directory.

You can create an object library under z/OS batch and TSO, but not from under USS.

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 595.

To compile the z/OS C source file WALTER.SOURCE(SUB1) with the LONGNAME compiler option, and then add it to the preallocated PDS (or PDSE) data set WALTER.SOURCE.LIB, use the following JCL. If this is the first time the Object Library Utility has been used to add an object module to WALTER.SOURCE.LIB, then the Basic and Enhanced Directory members will be created in this data set. If they already exist in this data set, then they will be updated to include the information for the object module created during the compilation.

```
//COMPILE EXEC EDCLIB,INFILE='WALTER.SOURCE(SUB1)',CPARM='LO',  
//          LIBRARY='WALTER.SOURCE.LIB',MEMBER='SUB1'
```

If you request a map for the library WALTER.SOURCE.LIB, use the following:

```
//OBJLIB EXEC EDCLIB,OPARM='MAP',LIBRARY='WALTER.SOURCE.LIB'
```

For z/OS C++, use the EDCLIB cataloged procedure. You can specify commands for the Object Library Utility step on the OPARM parameter. You can specify options for the Object Library Utility step. These options can generate a library directory, add members or delete members of a directory, or generate a map of library members and defined external symbols. This section shows you how to specify these options under z/OS batch.

Example: The following example creates a new Object Library Utility directory. If the directory already exists, it is updated.

```
//DIRDIR EXEC EDCLIB,
//      LIBRARY='LUCKY13.CXX.OBJMATH',
//      OPARM='DIR'
```

Example: To create a listing of all the object files (members) in an Object Library Utility directory:

```
//MAPDIR EXEC EDCLIB,
//      LIBRARY='LUCKY13.CXX.OBJMATH',
//      OPARM='MAP'
```

To add new members to an object library, use the ADD option to update the directory.

Example: To add a new member named MA191:

```
//ADDIR EXEC EDCLIB,
//      LIBRARY='LUCKY13.CXX.OBJMATH',
//      OPARM='ADD MA191',
//      OBJECT='DSNAME=LUCKY13.CXX.OBJ(OBJ191),DISP=SHR'
```

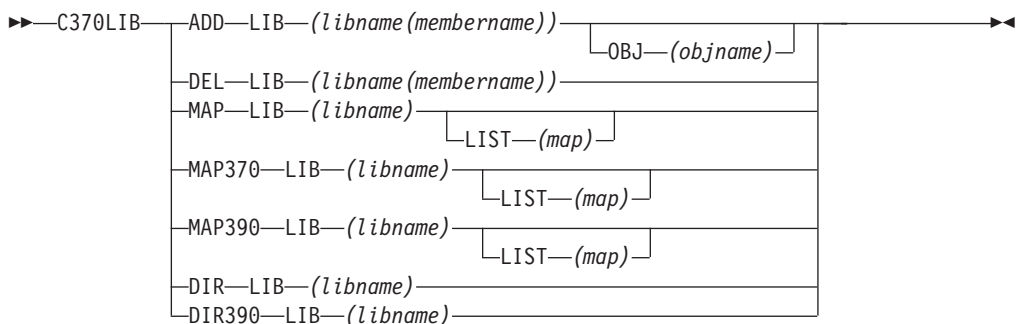
To delete a member from an object library, use the DEL option to keep the directory up to date.

Example: To delete a member named OLDMEM:

```
//DELDIR EXEC EDCLIB,
//      LIBRARY='LUCKY13.CXX.OBJMATH',
//      OPARM='DEL OLDMEM'
```

Creating an object library under TSO

The Object Library Utility has the following syntax:



where:

ADD Adds (or replaces) an object module to an object library.

If you use ADD to insert an object module to a member of a library that already exists, the previous member is deleted prior to the insert. If the source

data set is the same as the target data set, ADD does not delete the member, and only updates the Object Library Utility directory.

DEL	Deletes an object module from an object library.
MAP	Lists the names (entry points) of object library members in the Enhanced Directory Member if it is available; otherwise in the Basic Directory Member. You will only see object library members that were compiled with the options IPA(NOOBJECT), XPLINK or LP64 in the listing if the Enhanced Directory Member is available.
MAP370	Lists the names (entry points) of all object library members in the Basic Directory Member.
MAP390	Lists the names (entry points) of all object library members in the Enhanced Directory Member.
DIR	Builds the Object Library Utility directory member. The Object Library Utility directory contains the names (entry points) of library members. The DIR function is only necessary if object modules were previously added or deleted from the library without using the Object Library Utility.
DIR390	As of z/OS V1R2, the DIR and DIR390 commands are aliases of each other, and can be used interchangeably.
LIB(<i>libname(membername)</i>)	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(<i>objname</i>)	Specifies the source data set that contains the object module that is to be added to the library. If you do not specify a data set name, the Object Library Utility uses the target data set that you specified in LIB(<i>libname(membername)</i>) as the source.
LIB(<i>libname</i>)	Specifies the object library for which a map is to be produced or for which an Object Library Utility directory is to be built.
LIST(<i>map</i>)	Specifies the data set that is to contain the Object Library Utility listing. If you specified an asterisk (*), the listing is directed to your terminal. If you do not specify a data set name, a name is generated using the library name and the qualifier MAP. If TEST.OBJ is the input library data set, and your user prefix is FRANK, the data set name for the listing is FRANK.TEST.OBJ.MAP.

Under TSO, for z/OS C you can use either the C370LIB REXX EXEC or the CC REXX EXEC with the parameter C370LIB. The C370LIB parameter of the CC REXX EXEC specifies that, if the object module from the compile is directed to a PDS member, the Object Library Utility directory is to be updated. This step is the

equivalent to a compile and C370LIB ADD step. If the C370LIB parameter is specified, and the object module is not directed to a member of a PDS, the C370LIB parameter is ignored.

Object Library Utility Map

The Object Library Utility produces a listing for a given library when you specify the MAP, MAP370, or MAP390 command. MAP370 displays the listing using only the information in the Basic Directory Member. It assumes that all the extended attributes are set as zero, which provides backward compatibility with earlier versions of the Object Library Utility. MAP390 displays the listing using only the information in the Enhanced Directory Member. MAP is the preferred way of getting a listing. It generates a listing based on the Enhanced Directory Member if it's available, otherwise it generates a listing based on the Basic Directory Member. It provides additional attribute information on symbols when the information is available.

Example: The example that follows is produced by the Object Library Utility for a given library when you specify the MAP or MAP390 command. The listing contains information on each member of the library.

```

=====
1                               Object Library Utility Map
C370LIB:5694A01 V1 R6 M0 IBM Language Environment 2004/01/14 14:27:41
=====

```

Library Name: USERID.PROJECT.LIB

```

-----*
* 2 Member Name:   CGOFF           (P) 2004/01/14 13:26:41 *
*                                     5694A01   V1 R06   *
*-----*

```

3

User Comment:
AGGRCOPY(NOOVERLAP) NOALIAS ANSIALIAS ARCH(5) ARGPARSE NOASCII
BITFIELD(UNSIGNED) CHARS(UNSIGNED) NOCOMPACT NOCOMPRESS NOCONVLIT
CSECT() NODEBUG NODLL(NOCALLBACKANY) EXECOPS NOEXPORTALL FLOAT(HEX,
FOLD, AFP) GOFF NOGONUMBER NOIGNERRNO ILP32 NOINITAUTO NOINLINE NOIPA
LANGLVL(*EXTENDED) NOLIBANSI NOLOCALE LONGNAME MAXMEM(2097152)
NOOPTIMIZE PLIST(HOST) REDIR NORENT NOROCONST ROSTRING NOSERVICE
SPILL(128) START STRICT NOSTRICT_INDUCTION TARGET(LE, zOSV1R6) NOTEST
TUNE(5) UNROLL(AUTO) NOUPCONV NOXPLINK COMPILED_ON_MVS

```

4 ( L) Function Name: @InStream@#C
  ( L) Function Name: foo
  ( WL) External Name: @InStream@#S
  ( WL) External Name: @InStream@#T
  ( WL) External Name: this_int_is_in_writable_static_and_will_wrap_b
                        ecause_it_is_too_long

```

```

-----*
* Member Name: CPPIPANO           (P) 2004/01/14 14:27:44 *
*                                     5694A01   V1 R06   *
*-----*

```

User Comment:
AGGRCOPY(NOOVERLAP) ANSIALIAS ARCH(5) ARGPARSE NOASCII
BITFIELD(UNSIGNED) CHARS(UNSIGNED) NOCOMPACT NOCOMPRESS NOCONVLIT
NOCSECT CVFT NODEBUG DLL(NOCALLBACKANY) EXECOPS NOEXPORTALL FLOAT(HEX
FOLD,AFP) NOGOFF NOGONUMBER NOIGNERRNO NOINITAUTO NOINLINE(NOAUTO
NOREPORT,100,1000) IPA(NOLINK, NOOBJECT, NOOBJONLY, OPTIMIZE, NOGONUM
NOPDF1 NOPDF2 NOATTRIBUTE NOXREF) LANGLVL(ANONSTRUCT, ANONUNION,
ANSIFOR, DBCS, NODOLLARINAMES, EMPTYSTRUCT, ILLPTOM, IMPLICITINT,
LIBEXT, LONGLONG, OFFSETNONPOD, NOOLDDIGRAPH, OLDFRIEND, NOOLDMATH,
OLDTEMPACC, NOOLDTMPLALIGN, OLDTMPLSPEC, TRAILENUM, TYPEDEFCLASS, NOUCS
ZEROEXTARRAY) NOLIBANSI NOLOCALE LONGNAME ILP32 MAXMEM(2097152)
OBJECTMODEL(COMPAT) NOOPTIMIZE PLIST(HOST) REDIR ROCONST ROSTRING
ROUND(Z) NORTTI NOSERVICE SPILL(128) START STRICT NOSTRICT_INDUCTION
TARGET(LE, zOSV1R6) TEMPLATERECOMPILE NOTEMPLATEREGISTRY TMLPARSE(NO)
NOTEST(HOOK) TUNE(5) UNROLL(AUTO) NOXPLINK(NOBACKCHAIN,NOCALLBACK
NOGUARD,OSCALL(UPSTACK),NOSTOREARGS) COMPILED_ON_MVS

```

( I L) Function Name: myclass::myclass()
( I L) External Name: another_global
( I L) Function Name: myclass::foo(float,double)
( I L) Function Name: some_function(char)
( I L) External Name: some_global

```

```

*-----*
* Member Name: CPPNOIPA (P) 2004/01/14 14:27:47 *
* 5694A01 V1 R06 *
*-----*

```

User Comment:

```

AGGRCOPY(NOOVERLAP) ANSIALIAS ARCH(5) ARGPARSE NOASCII
BITFIELD(UNSIGNED) CHARS(UNSIGNED) NOCOMPACT NOCOMPRESS NOCONVLIT
NOCSECT CVFT NODEBUG DLL(NOCALLBACKANY) EXECOPS NOEXPORTALL FLOAT(HEX
FOLD,AFP) NOG OFF NOGONUMBER NOIGNERRNO NOINITAUTO NOINLINE(NOAUTO
NOREPORT,100,1000) NOIPA LANGLVL(ANONSTRUCT, ANONUNION, ANSIFOR, DBCS,
NODOLLARINNAMES, EMPTYSTRUCT, ILLPTOM, IMPLICITINT, LIBEXT, LONGLONG,
OFFSETNONPOD, NOOLDDIGRAPH, OLDFRIEND, NOOLDMATH, OLDTEMPACC,
NOOLDTMPLALIGN, OLDTMPLSPEC, TRAILENUM, TYPEDEFCLASS, NOUCS,
ZEROEXTARRAY) NOLIBANSI NOLOCALE LONGNAME ILP32 MAXMEM(2097152)
OBJECTMODEL(COMPAT) NOOPTIMIZE PLIST(HOST) REDIR ROCONST ROSTRING
ROUND(Z) NORTTI NOSERVICE SPILL(128) START STRICT NOSTRICT_INDUCTION
TARGET(LE, zOSV1R6) TEMPLATERE_COMPILE NOTEMPLATEREGISTRY TEMPLPARSE(NO)
NOTEST(HOOK) TUNE(5) UNROLL(AUTO) NOXPLINK(NOBACKCHAIN,NOCALLBACK
NOGUARD,OSCALL(UPSTACK),NOSTOREARGS) COMPILED_ON_MVS

```

```

( L) Function Name: myclass::myclass()
( L) Function Name: myclass::foo(float,double)
( L) Function Name: some_function(char)
( WL) External Name: another_global
( WL) External Name: some_global

```

```

*-----*
* Member Name: CPPLP64 (P) 2004/01/14 13:26:49 *
* 5694A01 V1 R06 *
*-----*

```

User Comment:

```

AGGRCOPY(NOOVERLAP) ANSIALIAS ARCH(5) ARGPARSE NOASCII
BITFIELD(UNSIGNED) CHARS(UNSIGNED) NOCOMPACT NOCOMPRESS NOCONVLIT
NOCSECT CVFT NODEBUG DLL(NOCALLBACKANY) EXECOPS NOEXPORTALL FLOAT(HEX
FOLD,AFP) GOFF NOGONUMBER NOIGNERRNO NOINITAUTO NOINLINE(NOAUTO
NOREPORT,100,1000) NOIPA LANGLVL(ANONSTRUCT, ANONUNION, ANSIFOR, DBCS,
NODOLLARINNAMES, EMPTYSTRUCT, ILLPTOM, IMPLICITINT, LIBEXT, LONGLONG,
OFFSETNONPOD, NOOLDDIGRAPH, OLDFRIEND, NOOLDMATH, OLDTEMPACC,
NOOLDTMPLALIGN, OLDTMPLSPEC, TRAILENUM, TYPEDEFCLASS, NOUCS,
ZEROEXTARRAY) NOLIBANSI NOLOCALE LONGNAME LP64 MAXMEM(2097152)
OBJECTMODEL(IBM) NOOPTIMIZE PLIST(HOST) REDIR ROCONST ROSTRING ROUND(Z)
NORTTI NOSERVICE SPILL(128) START STRICT NOSTRICT_INDUCTION TARGET(LE,
zOSV1R6) TEMPLATERE_COMPILE NOTEMPLATEREGISTRY TEMPLPARSE(NO)
NOTEST(HOOK) TUNE(5) UNROLL(AUTO) XPLINK(NOBACKCHAIN,NOCALLBACK,GUARD
OSCALL(UPSTACK),NOSTOREARGS) COMPILED_ON_MVS

```

```

(6 X L) Function Name: myclass::myclass()
(6 X L) Function Name: myclass::foo(float,double)
(6 X L) Function Name: some_function(char)
(6 XWL) External Name: another_global
(6 XWL) External Name: some_global

```

```

*-----*
* Member Name:  CIPA64                      (P) 2004/01/14 14:27:51 *
*                                           5694A01  V1 R06  *
*-----*

```

```

User Comment:
AGGRCOPY(NOOVERLAP) NOALIAS ANSIALIAS ARCH(5) ARGPARSE NOASCII
BITFIELD(UNSIGNED) CHARS(UNSIGNED) NOCOMPACT NOCOMPRESS NOCONVLIT
CSECT() NODEBUG NODLL(NOCALLBACKANY) EXECOPS NOEXPORTALL FLOAT(HEX,
FOLD, AFP) GOFF NOGONUMBER NOIGNERRNO NOINITAUTO NOINLINE IPA(NOLINK,
NOOBJ, COM, OPT, NOGONUM) LANGLVL(*EXTENDED) NOLIBANSI NOLOCALE
LONGNAME LP64 MAXMEM(2097152) NOOPTIMIZE PLIST(HOST) REDIR RENT
NOROCONST ROSTRING NOSERVICE SPILL(128) START STRICT NOSTRICT_INDUCTION
TARGET(LE, zOSV1R6) NOTEST TUNE(5) UNROLL(AUTO) NOUPCONV
XPLINK(NOBACKCHAIN, NOSTOREARGS, NOCALLBACK, GUARD, OSCALL(NOSTACK))
COMPILED_ON_MVS

```

```

(6IX L) Function Name: foo
(6IX L) External Name: this_int_is_in_writable_static_and_will_wrap_b
                       ecause_it_is_too_long

```

```

=====
|                               Symbol Definition Map                               |
=====

```

```

*-----*
| 5 Symbol Name: @InStream@#C |
*-----*

```

```

6 From member:  CGOFF Type: Function ( L)

```

```

*-----*
| Symbol Name: this_int_is_in_writable_static_and_will_wrap_because_i |
|                               t_is_too_long                               |
*-----*

```

```

From member:  CGOFF Type: External ( WL)
From member:  CIPA64 Type: External (6IX L)

```

```

*-----*
| Symbol Name: foo |
*-----*

```

```

From member:  CGOFF Type: Function ( L)
From member:  CIPA64 Type: Function (6IX L)

```

```

*-----*
| Symbol Name: @InStream@#T |
*-----*

```

```

From member:  CGOFF Type: External ( WL)

```

```

*-----*
| Symbol Name: @InStream@#S |
*-----*

```

```

From member:  CGOFF Type: External ( WL)

```

```

*-----*
| Symbol Name: some_function(char) |
*-----*

```

```

From member: CPPNOIPA Type: Function ( L)
From member: CPPLP64 Type: Function (6 X L)
From member: CPPIPANO Type: Function ( I L)

```

```

*-----*
| Symbol Name: myclass::myclass() |
*-----*

From member: CPPNOIPA Type: Function ( L)
From member: CPPIPANO Type: Function ( I L)

*-----*
| Symbol Name: myclass::foo(float,double) |
*-----*

From member: CPPNOIPA Type: Function ( L)
From member: CPPIPANO Type: Function ( I L)

*-----*
| Symbol Name: some_global |
*-----*

From member: CPPNOIPA Type: External ( WL)
From member: CPPLP64 Type: External (6 XWL)
From member: CPPIPANO Type: External ( I L)

*-----*
| Symbol Name: another_global |
*-----*

From member: CPPNOIPA Type: External ( WL)
From member: CPPLP64 Type: External (6 XWL)
From member: CPPIPANO Type: External ( I L)

*-----*
| Symbol Name: myclass::myclass() |
*-----*

From member: CPPLP64 Type: Function (6 X L)

*-----*
| Symbol Name: myclass::foo(float,double) |
*-----*

From member: CPPLP64 Type: Function (6 X L)

===== E N D O F O B J E C T L I B R A R Y M A P =====

```

| The Object Library Utility produces a listing for a given library when the MAP370
| command is specified. The listing produced by MAP370 will only contain information
| from the Object Library Utility directory members that are in the XOBJ object file
| format. In other words, files compiled with the GOFF compiler option (which includes
| all XPLINK and LP64 compiled object files) will not appear in the MAP370 listing. Also,
| IPA(NOOBJECT) compiled files will not appear in the MAP370 listing either.

```

=====
1                               Object Library Utility Map
C370LIB:5694A01 V1 R6 M0 IBM Language Environment 2004/01/14 14:27:41
=====

```

```

2 Library Name: USERID.PROJECT.LIB

```

```

-----*
* 3 Member Name: CGOFF (P) 2004/01/14 13:26:41 *
*-----*

```

```

-----*
* Member Name: CPPIPANO (P) 2004/01/14 14:27:44 *
* 5694A01 V1 R06 *
*-----*

```

```

User Comment:
AGGRCOPY(NOOVERLAP) ANSIALIAS ARCH(5) ARGPARSE NOASCII
BITFIELD(UNSIGNED) CHARS(UNSIGNED) NOCOMPACT NOCOMPRESS NOCONVLIT
NOCSECT CVFT NODEBUG DLL(NOCALLBACKANY) EXECOPS NOEXPORTALL FLOAT(HEX
FOLD,AFP) NOGOFF NOGONUMBER NOIGNERRNO NOINITAUTO NOINLINE(NOAUTO
NOREPORT,100,1000) IPA(NOLINK, NOOBJECT, NOOBJONLY, OPTIMIZE, NOGONUM
NOPDF1 NOPDF2 NOATTRIBUTE NOXREF) LANGLVL(ANONSTRUCT, ANONUNION,
ANSIFOR, DBCS, NODOLLARINAMES, EMPTYSTRUCT, ILLPTOM, IMPLICITINT,
LIBEXT, LONGLONG, OFFSETNONPOD, NOOLDDIGRAPH, OLDFRIEND, NOOLDMATH,
OLDTEMPACC, NOOLDTMPLALIGN, OLDTMPLSPEC, TRAILENUM, TYPEDEFCLASS, NOUCS
ZEROEXTARRAY) NOLIBANSI NOLOCALE LONGNAME ILP32 MAXMEM(2097152)
OBJECTMODEL(COMPAT) NOOPTIMIZE PLIST(HOST) REDIR ROCONST ROSTRING
ROUND(Z) NORTTI NOSERVICE SPILL(128) START STRICT NOSTRICT_INDUCION
TARGET(LE, zOSV1R6) TEMPLATERECOMPILE NOTEMPLATEREGISTRY TMLPARSE(NO)
NOTEST(HOOK) TUNE(5) UNROLL(AUTO) NOXPLINK(NOBACKCHAIN, NOCALLBACK
NOGUARD, OSCALL(UPSTACK), NOSTOREARGS) COMPILED_ON_MVS

```

```

4 ( L) External Name: another_global
( L) External Name: some_global
( L) Function Name: myclass::myclass()
( L) Function Name: myclass::foo(float,double)
( L) Function Name: some_function(char)

```

```

-----*
* Member Name: CPPNOIPA (P) 2004/01/14 14:27:47 *
* 5694A01 V1 R06 *
-----*

```

```

User Comment:
AGGRCOPY(NOOVERLAP) ANSIALIAS ARCH(5) ARGPARSE NOASCII
BITFIELD(UNSIGNED) CHARS(UNSIGNED) NOCOMPACT NOCOMPRESS NOCONVLIT
NOCSECT CVFT NODEBUG DLL(NOCALLBACKANY) EXECOPS NOEXPORTALL FLOAT(HEX
FOLD,AFP) NOGOFF NOGONUMBER NOIGNERRNO NOINITAUTO NOINLINE(NOAUTO
NOREPORT,100,1000) NOIPA LANGLVL(ANONSTRUCT, ANONUNION, ANSIFOR, DBCS,
NODOLLARINAMES, EMPTYSTRUCT, ILLPTOM, IMPLICITINT, LIBEXT, LONGLONG,
OFFSETNONPOD, NOOLDDIGRAPH, OLDFRIEND, NOOLDMATH, OLDTEMPACC,
NOOLDTMPLALIGN, OLDTMPLSPEC, TRAILENUM, TYPEDEFCLASS, NOUCS,
ZEROEXTARRAY) NOLIBANSI NOLOCALE LONGNAME ILP32 MAXMEM(2097152)
OBJECTMODEL(COMPAT) NOOPTIMIZE PLIST(HOST) REDIR ROCONST ROSTRING
ROUND(Z) NORTTI NOSERVICE SPILL(128) START STRICT NOSTRICT_INDUCTION
TARGET(LE, zOSV1R6) TEMPLATERECOMPILE NOTEMPLATEREGISTRY TMLPARSE(NO)
NOTEST(HOOK) TUNE(5) UNROLL(AUTO) NOXPLINK(NOBACKCHAIN, NOCALLBACK
NOGUARD, OSCALL(UPSTACK), NOSTOREARGS) COMPILED_ON_MVS

```

```

( L) Function Name: myclass::myclass()
( L) Function Name: myclass::foo(float,double)
( L) Function Name: some_function(char)
( WL) External Name: another_global
( WL) External Name: some_global

```

```

-----*
* Member Name: CPPLP64 (P) 2004/01/14 13:26:49 *
-----*

```

```

-----*
* Member Name: CIPA64 (P) 2004/01/14 14:27:51 *
-----*

```

```

=====
| 5 Symbol Definition Map |
=====

```

```

-----*
| 6 Symbol Name: some_function(char) |
-----*

```

```

From member: CPPIPAN0 Type: Function ( L)
From member: CPPNOIPA Type: Function ( L)

```



```

*-----*
| Symbol Name: myclass::myclass() |
*-----*

From member: CPPIPANO Type: Function ( L)
From member: CPPNOIPA Type: Function ( L)

*-----*
| Symbol Name: myclass::foo(float,double) |
*-----*

From member: CPPIPANO Type: Function ( L)
From member: CPPNOIPA Type: Function ( L)

*-----*
| Symbol Name: some_global |
*-----*

From member: CPPIPANO Type: External ( L)
From member: CPPNOIPA Type: External ( WL)

*-----*
| Symbol Name: another_global |
*-----*

From member: CPPIPANO Type: External ( L)
From member: CPPNOIPA Type: External ( WL)

===== E N D O F O B J E C T L I B R A R Y M A P =====

```

1 Map Heading

The heading contains the product number, the library version and release number, and the date and the time the Object Library Utility step began. The name of the library immediately follows the heading. To the right of the library name is the start time of the last Object Library Utility step that updated the Object Library Utility directory.

2 Member Heading

The name of the object module member is immediately followed by the Timestamp field presented in *yyyy/mm/dd* format. The meaning of the timestamp is enclosed in parentheses. The Object Library Utility retains a timestamp for each member and selects the time according to the following hierarchy:

- (P) indicates that the compile timestamp is extracted from the object module.
- (D) indicates that the timestamp is based on the time that the Object Library Utility DIR command was last issued.
- (T) indicates that the timestamp is the time that the ADD command was issued for the member.

The next line contains the ID of the processor that produced the object module. If the processor ID is not present, the Processor ID field is not listed.

3 User Comments

Displays any comments that were specified in the object module with the `#pragma` comment directive. It is possible to manually add such comments to the END records of an object member and have them displayed in the listing. These comments are extracted from the END record. The compile time options are stored in the same area as user comments and are displayed here as well.

4 Symbol Information

Immediately following Member Heading and user comments is a list of the defined objects that the member contains. Each symbol is prefixed by Type information that is enclosed in parentheses and either External Name or Function Name. Function Name will appear, provided the object module was compiled with the LONGNAME option and the symbol is the name of a defined external function. In all other cases, External Name is displayed. The Type field gives the following additional information on each symbol:

- 6** indicates that the object was compiled with LP64
- I** indicates that the name is compiled IPA(NOOBJECT).
- L** indicates that the name is a long name. A long name is an external C++ name in an object module or an external non-C++ name in an object module produced by compiling with the LONGNAME option.
- S** indicates that the name is a short name. A short name is an external non-C++ name in an object module produced by compiling with the NOLONGNAME option. Such a name is up to 8 characters long and single case.
- W** indicates that this is a writable static object. If it is not present, then this is not a writable static object.
- X** indicates that the name was compiled with the XPLINK option.

5 Symbol Definition Map

This section of the listing has an entry for each unique symbol name that appeared in the previous half of the listing. Any duplicate symbol names that appear in the entire Object Library Utility directory are grouped here for cross-reference purposes. This allows you to quickly determine which attributes a particular symbol name possesses within this Object Library Utility directory.

6 Symbol Source List

Displays the object module(s) found by the given symbol. Symbol attributes (described under "Symbol Information" above) immediately follow the names of the source objects.

Chapter 13. Filter Utility

This chapter describes how to use the CXXFILT utility to convert C++ mangled names to demangled names, which are human-readable.

When z/OS C++ compiles one of your source files, it does not place the function and class names appearing in that file verbatim in the object file, as would occur if you were compiling a z/OS C program. Instead, it "mangles" them, which means it encodes your function names with their type and scoping information. This process is required to support the more advanced features of C++ such as inheritance and function overloading. Mangled names are used to 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.

Example: The mangled name `__1s__7ostreamFPCc` is demangled as `ostream::operator<<(const char*)`.

class names

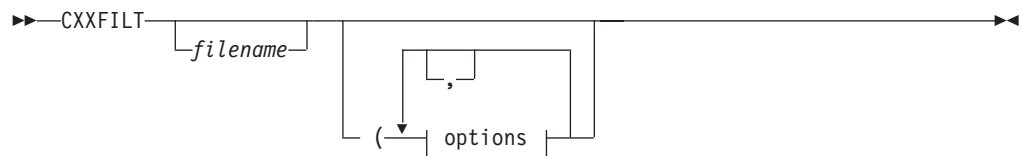
Includes stand-alone class names that do not appear within the context of a function name or a member variable.

Example: For example, the stand-alone class name `Q2_1X1Y` is demangled as `X::Y`

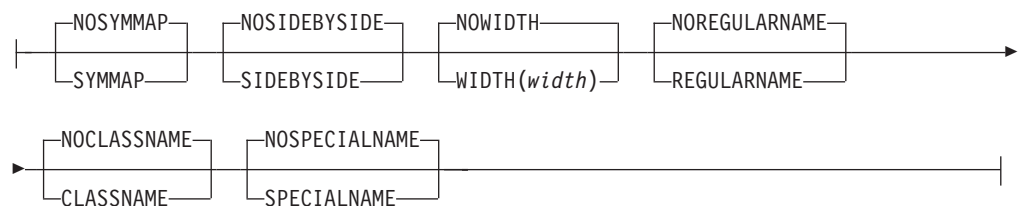
special names

Special compiler-generated class objects.

Example: For example, the compiler-generated symbol name `__vft1X` is demangled as `X::virtual-fn-table-ptr`.



options:



The *filename* refers to the files that contain the mangled names to be demangled. You may specify more than one file name, which can be a sequential file or a PDS member. If you do not specify a file name, CXXFILT reads its input from stdin.

The following 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__3F00F, then the SIDEBYSIDE option would produce this result:

```
F00::pr()      pr__3F00Fvxxxx
```

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__3F00Fv to F00:pr().

The mangled name that is supplied to CXXFILT is treated as a regular name by default. Specifying the NOREGULARNAME option will turn the default off. For example, specifying the CLASSNAME option without the NOREGULARNAME option will cause CXXFILT to treat the mangled name as either a regular name or stand-alone class name.

CLASSNAME | NOCLASSNAME

Default: NOCLASSNAME

This option demangles stand-alone class names such as Q2_1X1Y to X::Y.

To request that the mangled names be treated as stand-alone class names only, and never as a regular name, use both CLASSNAME and NOREGULARNAME.

SPECIALNAME | NOSPECIALNAME

Default: NOSPECIALNAME

This option demangles special names, such as compiler-generated symbol names; for example, __vft1X is demangled to X::virtual-fn-table-ptr.

To request that the mangled names be treated as special names only, and never as regular names, use CXXFILT (SPECIALNAME NOREGULARNAME).

Unknown type of name

If you cannot specify the type of name, use CXXFILT (SPECIALNAME CLASSNAME). This causes CXXFILT to attempt to demangle the name in the following order:

1. Regular name
2. Stand-alone class name
3. Special name

Under z/OS batch

The CXXFILT utility accepts input by two methods: from stdin or from a file.

Example: The following example uses the CXXFILT cataloged procedure, from data set CBC.SCBCPRC. CXXFILT reads from stdin (sysin), treats mangled names as regular names, produces a symbol mapping, and uses a field width 15 characters. The JCL follows:

```
//RUN EXEC CXXFILT,CXXPARM='(SYMMAP WIDTH(15) '  
:  
//SYSIN DD *  
pr__3F00Fvxxxx  
__1s__7ostreamFPCc  
__vft1X  
/*
```

The output is:

```
F00::pr()      xxxx  
ostream::operator<<(const char*)  
__vft1X
```

C++ Symbol Mapping

demangled	mangled
-----	-----
F00::pr()	pr__3F00Fv
ostream::operator<<(const char*)	__1s__7ostreamFPCs

Notes:

1. Because the trailing characters xxxx in the input name pr__3F00Fvxxxx are not part of a valid mangled name, and the SIDEBYSIDE option is not on, the trailing characters are not demangled.

Note: In the symbol mappings, the trailing characters xxxx are *not* displayed.

2. The `__vft1X` input is not demangled and does not appear in the symbol mapping because it is a special name, and the `SPECIALNAME` option was not specified.

The second method of giving input to `CXXFILT` is to supply it in one or more files. Fixed and variable file record formats are supported. Each line of a file can have one or more names separated by space. In the example below, mangled names are treated either as regular names or as special names (the special names are compiler-generated symbol names). Demangled names are printed in fields 35 characters wide, and output is in side-by-side format.

The output contains the following two mangled names:

```
pr__3F00Fv
__vft1X
```

You can use the following JCL:

```
//RUN EXEC CXXFILT,CXXPARAM='FILE1 (SPECIALNAME WIDTH(35) SIDEBYSIDE'
```

The `CXXFILT` utility terminates when it reads the end-of-file character.

Under TSO

The `CXXFILT` utility accepts input by two methods: from `stdin` or from a file.

With the first method, enter names after invoking `CXXFILT`. You can specify one or more names on one or more lines. The output is displayed after you press Enter. Names that are successfully demangled, as well as those which are not demangled, are displayed in the same order as they were entered. To indicate end of input, enter `/*`.

Example: In the following example, `CXXFILT` treats mangled names as regular names, produces a symbol mapping, and uses a field width 15 characters wide.

```
user> CXXFILT (SYMMAP WIDTH(15)
user> pr__3F00Fvxxxx
reply< F00::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> F00::pr()                pr__3F00Fv
reply> ostream::operator<<(const char*)  __ls__7ostreamFPCs
```

Notes:

1. Because the trailing characters xxxx in the input name `pr__3F00Fvxxxx` are not part of a valid mangled name, and the `SIDEBYSIDE` option is not on, the trailing characters are not demangled.

In the symbol mappings, the trailing characters xxxx are *not* displayed.

2. The `__vft1X` input is not demangled and does not appear in the symbol mapping because it is a special name, and the `SPECIALNAME` option was not specified.

3. The symbol mapping is displayed only after /* requests CXXFILT termination

The second method of giving input to CXXFILT is to supply it in one or more files. CXXFILT supports fixed and variable file record formats. Each line of a file can have one or more names separated by space. In the example below, mangled names are treated either as regular names or as special names (the special names are compiler-generated symbol names). Demangled names are printed in fields 35 characters wide, and output is in side-by-side format.

The output contains the following two mangled names:

```
pr__3F00Fv  
__vft1X
```

Example: Enter the following command:

```
cxxfilt FILE1 (SPECIALNAME WIDTH(35) SIDEBYSIDE
```

This will produce the output:

```
F00::pr()                pr__3F00Fv  
X::virtual-fn-table-ptr  __vft1X
```

CXXFILT terminates when it reads the end-of-file character.

Chapter 14. 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 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 34. DSECT Utility options, abbreviations, and IBM-supplied defaults

DSECT Utility Option	Abbreviated Name	IBM Supplied Default
SECT[(<i>name</i> ,...)]	None	SECT(ALL)
BITF0XL NOBITF0XL	BITF NOBITF	NOBITF0XL
COMMENT[(<i>delim</i> ,...)] NOCOMMENT	COM NOCOM	COMMENT
DECIMAL NODECIMAL	None	NODECIMAL
DEFSUB NODEFSUB	DEF NODEF	DEFSUB
EQUATE[(<i>suboptions</i> ,...)] NOEQUATE	EQU NOEQU	NOEQUATE
HDRSKIP[(<i>length</i>)] NOHDRSKIP	HDR(<i>length</i>) NOHDR	NOHDRSKIP
LOCALE(<i>name</i>) NOLOCALE	LOC NOLOC	NOLOCALE
INDENT[(<i>count</i>)] NOINDENT	IN(<i>count</i>) NOIN	INDENT(2)
LOWERCASE NOLOWERCASE	LC NOLC	LOWERCASE
OPTFILE(<i>filename</i>) NOOPTFILE	OPTF NOOPTF	NOOPTFILE
PPCOND[(<i>switch</i>)] NOPPCOND	PP(<i>switch</i>) NOPP	NOPPCOND
SEQUENCE NOSEQUENCE	SEQ NOSEQ	NOSEQUENCE
UNIQUE NOUNIQUE	None	NOUNIQUE
UNNAMED NOUNNAMED	UNN NOUNN	NOUNNAMED
OUTPUT[(<i>filename</i>)]	OUT[(<i>filename</i>)]	OUTPUT(DD:EDCDSECT)
RECFM[(<i>rcfm</i>)]	None	C/C++ Library defaults
LRECL[(<i>lrec</i>)]	None	C/C++ Library defaults
BLKSIZE[(<i>blksize</i>)]	None	C/C++ Library defaults
LP64	None	NOLP64

SECT

DEFAULT: SECT(ALL)

The SECT option specifies the section names for which structures are produced. The section names can be either CSECT or DSECT names. They must exist in the

SYSADATA file that is produced by the assembler. If you do not specify the SECT option or if you specify SECT(ALL), structures are produced for all CSECTs and DSECTs defined in the SYSADATA file, except for private code and unnamed DSECTs.

If the High Level Assembler is run with the BATCH option, only the section names defined within the first program can be specified on the SECT option. If you specify SECT(ALL) (or select it by default), only the sections from the first program are selected.

BITF0XL | NOBITF0XL

DEFAULT: NOBITF0XL

Specify the BITF0XL option when the bit fields are mapped into a flag byte as in the following example:

```
FLAGFLD DS F
          ORG FLAGFLD+0
B1FLG1  DC 0XL(B'10000000')'00'  Definition for bit 0 of 1st byte
B1FLG2  DC 0XL(B'01000000')'00'  Definition for bit 1 of 1st byte
B1FLG3  DC 0XL(B'00100000')'00'  Definition for bit 2 of 1st byte
B1FLG4  DC 0XL(B'00010000')'00'  Definition for bit 3 of 1st byte
B1FLG5  DC 0XL(B'00001000')'00'  Definition for bit 4 of 1st byte
B1FLG6  DC 0XL(B'00000100')'00'  Definition for bit 5 of 1st byte
B1FLG7  DC 0XL(B'00000010')'00'  Definition for bit 6 of 1st byte
B1FLG8  DC 0XL(B'00000001')'00'  Definition for bit 7 of 1st byte
          ORG FLAGFLD+1
B2FLG1  DC 0XL(B'10000000')'00'  Definition for bit 0 of 2nd byte
B2FLG2  DC 0XL(B'01000000')'00'  Definition for bit 1 of 2nd byte
B2FLG3  DC 0XL(B'00100000')'00'  Definition for bit 2 of 2nd byte
B2FLG4  DC 0XL(B'00010000')'00'  Definition for bit 3 of 2nd byte
```

When the bit fields are mapped as shown in 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 SYSADATA 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, $(\text{byte_size} * 2) - 1$]. 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 destructors, 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
#endif

```

This code will force the inclusion of the necessary header files, depending on whether the C or C++ compiler is used. It will also force the `_dec__var` and `_dec_uvar` types, which are outputted by the DSECT utility, to be mapped to the appropriate C or C++ decimal type. The definition of the macro `__decimal_found` is used to guard against the redefinition of macros if several DSECT utility output files are compiled together.

If the default `NODECIMAL` option is used then the DSECT utility will convert all P type DC/DS SYSATADA records to character arrays of the same byte size as the P type data, as is the existing behavior; for example, 171 (a value of PL3) will map to an unsigned `char[3]`.

DEFSUB | NODEFSUB

DEFAULT: DEFSUB

The `DEFSUB` option specifies whether `#define` directives will be built for fields that are part of a union or substructure.

Example: If the `DEFSUB` option is in effect, fields within a substructure or union have the field names prefixed by an underscore. A `#define` directive is written at the end of the structure to allow the field name to be specified directly as in the following example.

```

struct dsect_name {
    int field1;
    struct {
        int _subfld1;
        short int _subfld2;
        unsigned char _subfld3[4];
    } field2;
}
#define subfld1 field2._subfld1
#define subfld2 field2._subfld2
#define subfld3 field2._subfld3

```

If the DEFSUB option is in effect, the fields that are prefixed by an underscore may match the name of another field within the structure. No warning is issued.

EQUATE | NOEQUATE

DEFAULT: NOEQUATE

The EQUATE option specifies whether the EQU statements following a field are to be used to define bit fields, to generate `#define` directives, or are to be ignored.

The suboptions specify how the EQU statement is used. You can specify one or more of the suboptions, separating them by spaces or commas. If you specify more than one suboption, the EQU statements that follow a field are checked to see if they are valid for the first suboption. If so, they are formatted according to that option. Otherwise, the subsequent suboptions are checked to see if they are applicable.

If you specify the EQUATE option without suboptions, EQUATE(BIT) is used. If you specify NOEQUATE (or select it by default), the EQU statements that follow a field are ignored.

You can specify the following suboptions for the EQUATE option:

BIT Indicates that the value for an EQU statement is used to define the bits for a field where the field conforms to the following rules:

- The field does not have a duplication factor of zero.
- The field has a length between 1 and 4 bytes and has a bit length that is a multiple of 8.
- The field does not have more than one nominal value.

and the EQU statements that follow the field conform to the following rules:

- The value for the EQU statements that follow the field mask consecutive bits (for example, X'80' followed by X'40').
- The value for an EQU statement masks one bit or consecutive bits for example, B'10000000' or B'11000000', but not B'10100000'.
- Where the length of the field is greater than 1 byte, the bits for the remaining bytes can be defined by providing the EQU statements for the second byte after the EQU statement for the first byte.
- The value for the EQU statement is not a relocatable value.

Example: When you specify EQUATE(BIT), the EQU statements are converted as in the following example:

```
FLAGFLD DS H
FLAG21 EQU X'80'
FLAG22 EQU X'40'
FLAG23 EQU X'20'
FLAG24 EQU X'10'
FLAG25 EQU X'08'
FLAG26 EQU X'04'
FLAG27 EQU X'02'
FLAG28 EQU X'01'
FLAG2A EQU X'80'
FLAG2B EQU X'40'
struct dsect_name {
    unsigned int flag21 : 1,
                  flag22 : 1,
                  flag23 : 1,
                  flag24 : 1,
                  flag25 : 1,
                  flag26 : 1,
                  flag27 : 1,
                  flag28 : 1,
```

```

        flag2a   : 1,
        flag2b   : 1,
                : 6;
    }

```

BITL Indicates that the length attribute for an EQU statement is used to define the bits for a field where the field conforms to the following rules:

- The field does not have a duplication factor of zero.
- The field has a length between 1 and 4 bytes and has a bit length that is a multiple of 8.
- The field does not have more than one nominal value.

and the EQU statements that follow the field conform to the following rules:

- The value that is specified for the EQU statement has the same or overlapping offset as the field.
- The length attribute for the EQU statement is between 1 and 255.
- The length attribute for the EQU statement masks one bit or consecutive bits, for example, B'10000000' or B'11000000', but not B'10100000'.
- The value for the EQU statement is a relocatable value.

Example: When you specify EQUATE(BITL), the EQU statements are converted as in the following example:

```

BYTEFLD DS F
B1FLG1 EQU BYTEFLD+0,B'10000000'
B1FLG2 EQU BYTEFLD+0,B'01000000'
B1FLG3 EQU BYTEFLD+0,B'00100000'
B1FLG4 EQU BYTEFLD+0,B'00010000'
B1FLG5 EQU BYTEFLD+0,B'00001000'
B1FLG6 EQU BYTEFLD+0,B'00000100'
B1FLG7 EQU BYTEFLD+0,B'00000010'
B1FLG8 EQU BYTEFLD+0,B'00000001'
B2FLG1 EQU BYTEFLD+1,B'10000000'
B2FLG2 EQU BYTEFLD+1,B'01000000'
B2FLG3 EQU BYTEFLD+1,B'00100000'
B2FLG4 EQU BYTEFLD+1,B'00010000'
struct dsect_name {
    unsigned int b1flg1 : 1,
                 b1flg2 : 1,
                 b1flg3 : 1,
                 b1flg4 : 1,
                 b1flg5 : 1,
                 b1flg6 : 1,
                 b1flg7 : 1,
                 b1flg8 : 1,
                 b2flg1 : 1,
                 b2flg2 : 1,
                 b2flg3 : 1,
                 b2flg4 : 1,
                 : 20;
}

```

DEF Indicates that the EQU statements following a field are used to build #define directives to define the possible values for a field. The #define directives are placed after the end of the structure. The EQU statements should not specify a relocatable value.

Example: When you specify EQUATE(DEF), the EQU statements are converted as in the following example:

```

FLAGBYTE DS X
FLAG1 EQU X'80'
FLAG2 EQU X'20'
FLAG3 EQU X'10'
FLAG4 EQU X'08'
FLAG5 EQU X'06'

```

```

FLAG6 EQU X'01'
struct dsect_name {
    unsigned char flagbyte;
}
/* Values for flagbyte field */
#define flag1 0x80
#define flag2 0x20
#define flag3 0x10
#define flag4 0x08
#define flag5 0x06
#define flag6 0x01

```

HDRSKIP | NOHDRSKIP

DEFAULT: NOHDRSKIP

The HDRSKIP option specifies that the fields within the specified number of bytes from the start of the section are to be skipped. Use this option where a section has a header that is not required in the structure produced.

The value that is specified on the HDRSKIP option indicates the number of bytes at the start of the section that are to be skipped. HDRSKIP(0) is equivalent to NOHDRSKIP.

Example: In the following example, if you specify HDRSKIP(8), the first two fields are skipped and only the remaining two fields are built into the structure.

```

SECTNAME DSECT
PREFIX1 DS CL4
PREFIX2 DS CL4
FIELD1 DS CL4
FIELD2 DS CL4
struct sectname {
    unsigned char field1[4];
    unsigned char field2[4];
}

```

If the value specified for the HDRSKIP option is greater than the length of the section, the structure is not be produced for that section.

INDENT | NOINDENT

DEFAULT: INDENT(2)

The INDENT option specifies the number of character positions that the fields, unions, and substructures are indented. Turn off indentation by specifying INDENT(0) or NOINDENT. The maximum value that you can specify for the INDENT option is 32767.

LOCALE | NOLOCALE

The LOCALE(name) specifies the name of a locale to be passed to the setlocale() function. Specifying LOCALE without the *name* parameter is equivalent to passing the NULL string to the setlocale() function.

The structure produced contains the left and right brace, and left and right square bracket, backslash, and number sign which have different code point values for the different code pages. When the LOCALE option is specified, and these characters are written to the output file, the code point from the LC_SYNTAX category for the specified locale is used.

The default is NOLOCALE.

You can abbreviate the option to `LOC(name)` or `NOLOC`.

LOWERCASE | NOLOWERCASE

DEFAULT: LOWERCASE

The LOWERCASE option specifies whether the field names within the C structure are to be converted to lowercase or left as entered. If you specify LOWERCASE, all the field names are converted to lowercase. If you specify NOLOWERCASE, the field names are built into the structure in the case in which they were entered in the assembler section.

LP64 | NOLP64

DEFAULT: NOLP64

The equivalent of NOLP64 for the compiler is the option ILP32, which means 32-bit integer, long, and pointer type. This is the default in the compiler as well. LP64 means 64-bit long and pointer type. LP64 and ILP32 specify the data model for the programming language.

The LP64 option instructs the DSECT utility to generate structures for use by the programs compiled with the LP64 option. When this option is enabled, address fields are mapped to C pointer types (64 bits), and 64-bit integer fields are mapped to long data types. C/C++ also supports a `__ptr32` qualifier for declaring pointers that are 32-bit in size, which means that if a field is explicitly specified with a 31-bit address, it is mapped to a `__ptr32` qualified pointer.

OPTFILE | NOOPTFILE

The OPTFILE(*filename*) option specifies the filename that contains the records that specify the options to be used for processing the sections. The records must be as follows:

- The lines must begin with the SECT option, and only one section name must be specified. The options following determine how the structure is produced for the specified section. The section name must only be specified once.
- The lines may contain the options BITFOXL, 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.


```

#ifdef 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 `#ifdef` and `#endif` directives are placed around all structures that are produced. If you do not specify a switch, the `#ifdef` and `#endif` directives are placed around each structure produced.

SEQUENCE | NOSEQUENCE

DEFAULT: NOSEQUENCE

The SEQUENCE option specifies whether sequence numbers will be placed in columns 73 to 80 of the output record. If you specify the SEQUENCE option, the structure is built into columns 1 to 72 of the output record, and sequence numbers are placed in columns 73 to 80. If you specify NOSEQUENCE (or select it by default), sequence numbers are not generated, and the structure is built within all available columns in the output record.

If the record length for the output file is less than 80 characters, the SEQUENCE option is ignored.

UNIQUE | NOUNIQUE

DEFAULT: NOUNIQUE

The UNIQUE option tells the DSECT utility to consider the given unique string as not occurring in any field names in the input SYSADATA. This is necessary because it is a guarantee from the user that if the DSECT utility were to use the unique string to map national characters, no conflict would occur with any other field name. Given this guarantee the DSECT utility maps national characters as follows:

```

# = unique string + 'n' + unique string
@ = unique string + 'a' + unique string
$ = unique string + 'd' + unique string

```

Example: If the default `"_"` unique string was used then the national characters would be mapped as:

```

# = _n_
@ = _a_
$ = _d_

```

If the default NOUNIQUE option is enabled, the DSECT utility converts all national characters to a single underscore, even if the resulting label names conflict (as is the existing behavior).

Note: If the DSECT utility detects a field name that has a length that exceeds the maximum allowed, a message is displayed and the name is truncated in the output. This can happen due to the substitution characters in the UNIQUE option. That is, the field name as specified by the user is within the maximum limit, but due to the presence of national characters and the mapping done by UNIQUE, the resulting field name can exceed the limit. The DSECT utility then ends the output field name with `"..."` to make it easy to find.

The user should check and fix the field name either by changing the UNIQUE option, or by shortening the original field name, or both.

UNNAMED | NOUNNAMED

DEFAULT: NOUNNAMED

The 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 80 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

Example: You can use the IBM-supplied cataloged procedure EDCDSECT to execute the DSECT utility as in the following example.

```

KNOWN:  - The assembler source name is FRED.SOURCE(TESTASM).
        - The structure is to be written to FRED.INCLUDE(TESTASM).
        - The required DSECT Utility options are EQU(BIT).

USE THE FOLLOWING JCL:
//DSECT EXEC PROC=EDCDSECT,
//      INFILE='FRED.SOURCE(TESTASM)',
//      OUTFILE='FRED.INCLUDE(TESTASM)',
//      DPARM='EQU(BIT)'
```

Figure 43. 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.

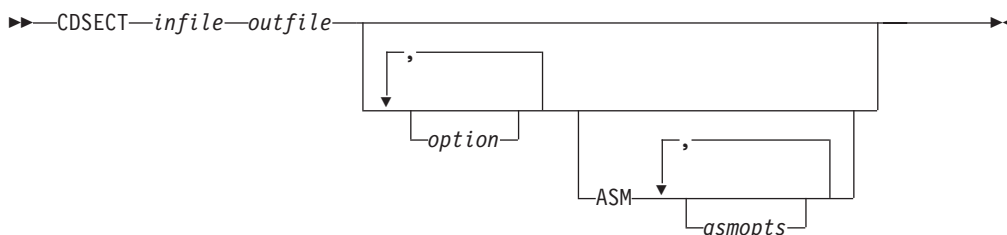
The parameters to the EDCDSECT procedure are:

Table 35. 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 44. 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 15. 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:

iconv Converts a file from one coded character set encoding to another. You can use `iconv` to convert C source code before compilation or to convert input files. For more information, refer to *z/OS UNIX System Services Command Reference*.

uconvdef

Reads the input source file and creates a binary conversion table. The input source file defines a mapping between UCS-2 and multibyte code sets. For more information, refer to *z/OS UNIX System Services Command Reference*.

genxlt Generates a translate table that the `iconv` utility and the `iconv` family of functions can use to convert coded character sets. It can be used to build code set converters for code pages that are not supplied with z/OS 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 from applications running 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 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.

There are three standard library functions that can be used by any application to change the character set of data. These functions are `iconv_open()`, `iconv()`, and `iconv_close()`. For more information on the `iconv` utility, see *z/OS 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

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, `i conv` will fail. An output data set with a fixed record format may only be used if all the records created by the `i conv` 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 `i conv` utility will fail.

For more information on the `i conv` utility, refer to *z/OS C/C++ Programming Guide*.

Under TSO

TSO CLIST `ICONV` invokes the `i conv` utility to copy the input data set to the output data set and convert the characters from the input code page to the output code page.

The parameters of the `ICONV` CLIST are as follows:

```
▶▶—ICONV—infile—outfile—FROMCODE(—fromcode—)—TOCODE(—tocode—)————▶▶
```

Where:

<i>infile</i>	The input data set name.
<i>outfile</i>	The output data set name.
<i>fromcode</i>	The name of the code set in which the input data is encoded.
<i>tocode</i>	The name of the code set to which the output data is to be converted.

Example:

```
ICONV INPUT.FILE OUTPUT.FILE FROMCODE(IBM-037) TOCODE(IBM-1047)
```

The output data set must be pre-allocated. If the data set does not exist, `i conv` will fail. An output data set with a fixed record format may only be used if all the records created by the `i conv` 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 `i conv` utility will fail.

For more information on `i conv`, refer to *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.

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 and XPLINK versions of the GENXLT table should always be generated. If any XPLINK applications will require one of these translation tables, then the XPLINK version should also be generated.

Under TSO, you specify the options on the command line. Under z/OS batch, the options are specified on the EXEC PARM, and may be separated by spaces or commas. If you specify the same option more than once, `genxlt` uses the last specification.

- DBCS | NODBCS** Specifies whether `genxlt` will convert the DBCS characters within shift-out and shift-in characters. You should only specify the DBCS option when you are converting an EBCDIC code page to a different EBCDIC code page.

If the DBCS option is specified, when a shift-out character is encountered in the input, the characters up to the shift-in character are copied to the output, and not converted. There must be an even number of characters between the shift-out and shift-in characters, and the characters must be valid DBCS characters.

If you specify the NODBCS option, `genxlt` treats all the characters as a single SBCS character, and does not perform a check of DBCS characters.

For more information on the `genxlt` utility, refer to *z/OS 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:

<code>INFILE</code>	The data set name for the file that contains the character conversion information.
<code>OUTFILE</code>	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 <code>EDCU</code> as the first four characters. The XPLINK version of this table should have <code>CEHU</code> as the first four characters.
<code>GOPT</code>	Options for the <code>genxlt</code> utility.

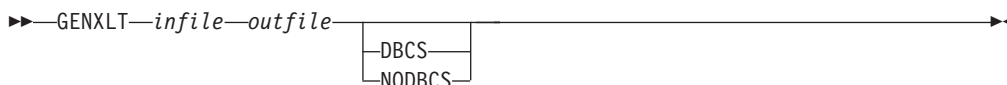
Example:

```
//GENXLT EXEC PROC=GENXLT,
//      INFILE='FRED.GENXLT.SOURCE(EDCUEAEY)',
//      OUTFILE='FRED.GENXLT.LOADLIB(EDCUEAEY)',
//      GOPT='DBCS'
```

Under TSO

TSO CLIST `GENXLT` invokes the `genxlt` utility to read the character conversion information and produce the conversion table. It then invokes the system Linkage Editor to build the load module.

The general parameters for `GENXLT CLIST` are as follows:



Where:

- infile* The file name for the file that contains the character conversion information.
- outfile* The file name for the output file that is to contain the link-edited conversion table. The non-XPLINK version of the table should have `EDCU` as the first four characters. The XPLINK version of this table should have `CEHU` as the first four characters.

For example:

```
GENXLT GENXLT.SOURCE(EDCUEAEY) GENXLT.LOADLIB(EDCUEAEY) DBCS
```

localedef Utility

A *locale* is a collection of data that defines language and cultural conventions. Locales consist of various categories, that are identified by name, that characterize specific aspects of your cultural environment.

The `localedef` utility generates locales according to the rules that are defined in the locale definition file. A user can create his own customized locale definition file.

The `localedef` utility creates locale objects, which are used by the `setlocale()` service of the run-time library. This service can be called from both non-XPLINK and XPLINK applications. The non-XPLINK, XPLINK, and 64-bit locale object versions have different names. Also, `localedef` can generate the locale objects into a PDS or PDSE under BATCH or TSO, or into the HFS under the z/OS shell. The non-XPLINK, XPLINK, and 64-bit versions of the locale object should always be generated. If any XPLINK applications will use the locale then the XPLINK version should also be generated.

The utility reads the locale definition file and produces a locale object that the locale-specific library functions can use. You invoke `localedef` using either a JCL procedure or a TSO CLIST, or by specifying the `localedef` command under z/OS UNIX System Services. To activate a locale during your application's execution, you call the run-time function `setlocale()`.

Note: TSO and z/OS batch are not supported for building 64-bit locales. You must use the `localedef` command under z/OS UNIX System Services to build 64-bit locales.

The options for the `localedef` utility in TSO or z/OS batch are as follows. Spaces or commas can separate the options. If you specify the same option more than once, `localedef` uses the last option that you specified.

<code>CHARMAP</code> (<i>name</i>)	Specifies the member name of the file that contains the definition of the encoded character set. If you do not specify this option, the <code>localedef</code> utility assumes the encoded character set IBM-1047. The name that is specified for the <code>CHARMAP</code> is the member name within a partitioned data set, with the - (dash) sign converted to an @ (at) sign.
<code>FLAG</code> (<u>W</u> <u>E</u>)	The <code>FLAG</code> option controls whether <code>localedef</code> issues warning messages. If you specify <code>FLAG(W)</code> , <code>localedef</code> issues warning and error messages. If you specify <code>FLAG(E)</code> , <code>localedef</code> issues only the error messages.
<code>BLDERR</code> <u><code>NOBLDERR</code></u>	If you specify the <code>BLDERR</code> option, <code>localedef</code> generates the locale even if it detects errors. If you specify the <code>NOBLDERR</code> option, <code>localedef</code> 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 definition files, codeset mapping files (CHARMAPs), method files, and locale object names, refer to *z/OS C/C++ Programming Guide*. For information on using the `localedef` utility under z/OS UNIX System Services, refer to *z/OS UNIX System Services Command Reference*.

Under z/OS batch

Note: To build XPLINK optimized locales, use EDCXLDEF.

Under z/OS batch, JCL procedure EDCLDEF invokes the `localedef` utility. It does the following:

1. Invokes the EDCLDEF module to read the locale definition data set and produces the C code to build the locale
2. Invokes the z/OS 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 name that is chosen determines the locale that is built (for further information, see <i>z/OS C/C++ Programming Guide</i>). The XPLINK version should have CEH\$ or CEH@ as the first four characters of the member name.
LOPT	The options for the <code>localedef</code> utility

Example:

```
//LOCALDEF EXEC PROC=EDCLDEF,  
//          INFILE='FRED.LOCALE.SOURCE(EDC$EUEY)',  
//          OUTFILE='FRED.LOCALE.LODLIB(EDC$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:

```
▶▶—LOCALDEF—infile—outfile—┬──LOPT(—loptions)──┬──XPLINK──┬──▶▶
```

where:

<i>infile</i>	The data set name for the data set that contains the locale definition information
<i>outfile</i>	For non-XPLINK, it is the data set name for the output partitioned data set and member that is to contain the link-edited locale object.

For XPLINK, it is the data set name for the output PDSE and member that is to contain the bound locale object. The non-XPLINK version of the locale object should have EDC\$ or EDC@ as the first four characters of the member name. The XPLINK version should have CEH\$ or CEH@ as the first four characters of the member name.

loptions The options for the `localedef` utility.

XPLINK Indicates that the locale to be built is an XPLINK locale.

Example: In the following example, the input source is `LOCALE.SOURCE(EDC$EUEY)`, the output library is `LOCALE.LOADLIB(EDC$EUEM)` for `en_us.IBM-297`, and options are `CHARMAP(IBM-297)`:

```
LOCALEDEF LOCALE.SOURCE(EDC$EUEY) LOCALE.LOADLIB(EDC$EUEM) LOPT(CHARMAP(IBM-297))
```

Under TSO, you specify the options on the command line.

Under the z/OS Shell

Under z/OS UNIX System Services, use the `localedef` command to invoke the `localedef` utility. The following is the invocation syntax for the `localedef` command:

```
localedef [-c] [-w] [-X] [-A][-f charmap] [-i sourcefile] [-m] [-L binderoptions]  
name
```

Options:

- A** Causes `localedef` to generate an ASCII locale object. ASCII locales invoke ASCII methods, so they must be generated using ASCII *charmaps*. An ASCII *charmap* maps symbolic character names into ASCII code points, but even ASCII *charmap* specifications are written in EBCDIC code page IBM-1047. Users must ensure that the *charmap* specified, when they invoke the `localedef` utility, is an ASCII *charmap*. Note: When **-A** is specified, **-X** is assumed because ASCII locales are only supported as XPLINK locales.
- c** Creates permanent output even if there were warning messages. Normally, `localedef` does not create permanent output when it has issued warning messages.
- f *charmap*** Specifies a *charmap* file that contains a mapping of character symbols and collating element symbols to actual character encodings.
- i *sourcefile*** Specifies the file that contains the source definitions. If there is no **-i**, `localedef` reads the source definitions from the standard input.
- m *MethodFile*** Specifies the names of a method file that identifies the methods to be overridden when constructing a locale object. The `localedef` utility reads a method file and uses indicated entry points when constructing a locale object. Method files are used to replace IBM-supplied method functions with user-written method functions. For each replaced method, the method file supplies the user-written method function name and optionally indicates where the method function code is to be found (.o file, archive library or DLL). Method files typically replace the *charmap* related methods. When this is done, the end result is the creation of a locale, which supports a blended code page. The user-written method functions are used

both by the locale-sensitive APIs they represent, and also by `localedef` itself while generating the method-file based ASCII locale object. This second use by `localedef` itself causes a temporary DLL to be created, while processing the *charmap* file supplied on the `-f` parameter. The name of the file containing method objects or side deck information is passed by `localedef` as a parameter on the c89 command line, so the standard archive/object/side deck suffix naming conventions apply (in other words, `.a`, `.o`, `.x`).

Note: Method files may only be used when constructing ASCII locale objects (that is, when the `-A` option is also specified). If the `-A` option is not specified along with the `-m` option, then a severe error message will be issued and processing will be terminated.

- `-w` Instructs `localedef` to issue a warning message when a duplicate character definition is found. This is mainly intended for debugging character map specifications. It can help to ensure that a code point value is not accidentally assigned to the wrong symbolic character name.
- `-X` Causes `localedef` to generate an XPLINK AMODE 31 locale object (DLL).
- `-L binderoptions` Instructs `localedef` to pass additional binder options (mostly for diagnostic purposes).
- `-6` Instructs `localedef` to an XPLINK AMODE 64 locale object (DLL). The `-X` option is implied when this option is specified.
- 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 *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 *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 can be invoked under z/OS batch and TSO, and is shipped with the z/OS C/C++ compiler.
- The other can be invoked 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 System Services utilities

This part contains information about the z/OS UNIX System Services utilities.

- Chapter 16, “Archive and Make Utilities,” on page 465
- Chapter 17, “BPXBATCH Utility,” on page 467
- Chapter 18, “c89 — Compiler invocation using host environment variables,” on page 471
- Chapter 19, “xlc — Compiler invocation using a customizable configuration file,” on page 513

Chapter 16. 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 *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 files.

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 with object files in data sets. For more information, see Chapter 12, “Object Library Utility,” on page 421.

By default, the c89 and c++ utilities require that archive libraries end in the suffix .a, as in file.a. For example; source file dirsum.c is in your src subdirectory in your working directory, and the archive library symb.a is in your working directory. To compile dirsum.c and resolve external symbols from symb.a, and create the executable in exfils/dirsum enter:

```
c89 -o exfils/dirsum src/dirsum.c symb.a
```

Creating archive libraries

To create the archive library, use the ar -r option.

Example: To create an archive library that is named bin/libbrobompgm.a from your working directory, and add the member jkeyadd.o to it, specify:

```
ar -rc ./bin/libbrobompgm.a jkeyadd.o
```

ar creates the archive library file libbrobompgm.a in the bin subdirectory of your HFS working directory. The -c option tells ar to suppress the message that it normally sends when it creates an archive library file.

Example: For control purposes, when working interactively, you can use the -v option to generate a message as each member is added to the archive:

```
ar -rv ./bin/libbrobompgm.a jkeyadd.o
```

Example: To display the object files that are archived in the bin/libbrobompgm.a library from your working directory, specify:

```
ar -t ./bin/libbrobompgm.a
```

For a detailed discussion of the `ar` utility, see *z/OS UNIX System Services Command Reference*.

Creating makefiles

The `make` utility maintains all the parts of and dependencies for your application program. It uses a *makefile*, to keep your application parts (listed in it) up to date with one another. If one part changes, `make` updates all the other files that depend on the changed part.

A makefile is a 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 *z/OS UNIX System Services Programming Tools*, and *z/OS UNIX System Services Command Reference* for a detailed discussion of the shell `make` utility.

Makedepend Utility

The `makedepend` utility can also be used to create a makefile 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 *z/OS UNIX System Services Command Reference* for a detailed discussion of the `makedepend` utility.

Chapter 17. 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 forked/executed. Formerly, with BPXBATCH, this could not be done since BPXBATCH and the requested program shared the same environment variables. BPXBATSL is provided as an alternative to BPXBATCH. It will force the running of the target program into the same address space as the job itself is initiated in, so that all resources for the job can be used by the target program; for example, DD allocations. In all other respects, it is identical to BPXBATCH.

For information on c89 commands, see Chapter 18, “c89 — Compiler invocation using host environment variables,” on page 471.

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 the program that is being invoked. This can be useful when not using the shell, such as when using the `PGM` parameter.

Example: For JCL jobs, specify `PATH` keyword options on DD statements; for example:

```

//jobname JOB ...

//stepname EXEC PGM=BPXBATCH,PARM='PGM program-name parm1 parm2'

//STDIN DD PATH='/stdin-file-pathname',PATHOPTS=(ORDONLY)
//STDOUT DD PATH='/stdout-file-pathname',PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=SIRWXU
//STDERR DD PATH='/stderr-file-pathname',PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
// PATHMODE=SIRWXU
:

```

You can also allocate the standard files dynamically through use of SVC 99.

For TSO/E, you specify PATH keyword options on the ALLOCATE command. For example:

```

ALLOCATE FILE(STDIN) PATH('/stdin-file-pathname') PATHOPTS(ORDONLY)
ALLOCATE FILE(STDOUT) PATH('/stdout-file-pathname')
PATHOPTS(OWRONLY,OCREAT,OTRUNC) PATHMODE(SIRWXU)
ALLOCATE FILE(STDERR) PATH('/stderr-file-pathname')
PATHOPTS(OWRONLY,OCREAT,OTRUNC) PATHMODE(SIRWXU)

```

```
BPXBATCH SH program-name
```

You must always allocate stdin as read. You must always allocate stdout and stderr as write.

Parameter

BPXBATCH accepts one parameter string as input. At least one blank character must separate the parts of the parameter string. When BPXBATCH is run from a batch job, the total length of the parameter string must not exceed 100 characters. When BPXBATCH is run from TSO, the parameter string can be up to 500 characters. If neither SH nor PGM is specified as part of the parameter string, BPXBATCH assumes that it must start the shell to run the shell script allocated by stdin.

SH | PGM

Specifies whether BPXBATCH is to run a shell script or command or a z/OS 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

You should be aware of the following:

1. BPXBATCH is an alias for the program BPXMBATC, which resides in the SYS1.LINKLIB data set.
2. BPXBATCH must be invoked from a user address space running with a program status word (PSW) key of 8.
3. BPXBATCH does not perform any character translation on the supplied parameter information. You should supply parameter information, including HFS pathnames, using only the POSIX portable character set.
4. A program that is run by BPXBATCH cannot use allocations for any files other than stdin, stdout, or stderr.
5. BPXBATCH does not close file descriptors except for 0, 1, and 2. Other file descriptors that are open and not defined as "marked to be closed" remain open when you call BPXBATCH. BPXBATCH runs the specified script or executable file.
6. BPXBATCH uses write-to-operator (WTO) routing code 11 to write error messages to either the JCL job log or your TSO/E terminal. Your TSO/E user profile must specify WTPMSG so that BPXBATCH can display messages to the terminal.

Files

The following list describes the files:

- SYS1.LINKLIB(BPXMBATC) is the BPXBATCH program location.
- The stdin default is /dev/null.
- The stdout default is /dev/null.
- The stdev default is /dev/null.
- The stderr default is the value of stdout. If all defaults are accepted, stderr is /dev/null.

Chapter 18. c89 — Compiler invocation using host environment variables

Format

```
c89 | cc | c++ | cxx [-+CcEfgOpqrsVv0123]
    [-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*.
- cxx/c++ must be used when compiling C++ programs. Prior to z/OS V1R2, the C++ compiler supported the *Draft Proposal International Standard for Information Systems — Programming Language C++ (X3J16)*. As of z/OS 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 name of the document that describes the program you use for that step and the document that describes any messages issued by that program, and prefixes to those messages:

Table 36. c89, cc, and c++ programs and reference documentation

Step Name	Document Describing Options and How to Call Program	Document Containing Messages Issued by Program	Prefix of Messages Issued by Program
ASSEMBLE	<i>HLASM Programmer's Guide</i>	<i>HLASM Programmer's Guide</i>	ASMA
COMPILE, IPACOMP, TEMPINC, IPATEMP, IPALINK	<i>z/OS C/C++ User's Guide</i>	<i>z/OS C/C++ Messages for z/OS V1R5 and later releases</i>	CBC for OS/390 V2R10 and z/OS V1R1; CCN for z/OS V1R2 and later releases

Table 36. c89, cc, and c++ programs and reference documentation (continued)

Step Name	Document Describing Options and How to Call Program	Document Containing Messages Issued by Program	Prefix of Messages Issued by Program
PRELINK	<i>z/OS Language Environment Programming Guide</i> and <i>z/OS C/C++ User's Guide</i>	<i>z/OS Language Environment Debugging Guide</i>	EDC
LINKEDIT (Program Management Binder)	<i>z/OS DFSMS Program Management</i>	<i>z/OS MVS System Messages, Vol 8 (IEF-IGD)</i>	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 *z/OS Introduction and Release Guide* for further details. Also see {_CLIB_PREFIX} and {_PLIB_PREFIX} in “Environment variables” on page 486 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 486 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 the information on "Using TEMPINC or NOTEMPINC" in *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 481.

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 506. Also see Usage Note 13 on page 507.
- `-E` Specifies that output of the compiler preprocessor phase be copied to `stdout`. Object files are not created, and no link-editing is performed.
- `-e function`
Specifies the name of the function to be used as the entry point of the program. This can be useful when creating a fetchable program, or a non-C or non-C++ main, such as a COBOL program. Non-C++ linkage symbols of up to 1024 characters in length may be specified. You can specify an S-name by preceding the function name with double slash (`//`). (For more information about S-names, see Usage Note 23 on page 510.)

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 ". For more information about the Program Management Binder and the ORDER control statement, see *z/OS DFSMS Program Management*.

This option may be required when building products which are intended to be installed using the IBM SMP/E product. When installing ++MOD elements with SMP/E, binder control statements should be provided in the JCLIN created to install the product instead of being embedded in the elements themselves.

-F Ignored by cc. Provided for compatibility with historical implementations of cc. Flagged as an error by c89 and c++.

-f Ignored by cc. Provided for compatibility with historical implementations of cc. Flagged as an error by c89 and c++.

Historical implementations of C/C++ used this option to enable floating-point support. Floating-point is automatically included in z/OS C/C++. However, in z/OS C/C++, two types of floating-point support are available:

HEXADECIMAL

Base 16 zSeries hexadecimal format. The zSeries hexadecimal format is referred to as the hexadecimal floating-point format, and is unique to zSeries hardware. This is the default.

IEEE754

Base 2 IEEE-754 binary format. The IEEE-754 binary format is referred to as binary floating-point format. The IEEE-754 binary format is the more common floating point format used on other platforms.

If you are porting an application from another platform, transmitting floating-point numbers between other platforms or workstations, or your application requires the larger exponent range provided by IEEE-754 binary format, then you should consider using IEEE floating-point.

Example: The following is an example of compiling with *IEEE-754* binary floating point format:

```
c89 -o outfile -Wc,'float(ieee)' file.c
```

-g Specifies that a side file that contains symbolic information is emitted and the executable is to be loaded into read/write storage, which is required for source-level debugging with dbx, and other debuggers.

For 32-bit compiles, if the `_DEBUG_FORMAT=ISD` environment variable is exported, then `-g` specifies that the output file (executable) is to contain symbolic information and is to be loaded into read/write storage, which is required for source-level debugging with dbx, and other debuggers.

When specified for the compilation phase, the compiler produces symbolic information for source-level debugging.

When specified for the link-editing phase, the executable file is marked as being serially reusable and will always be loaded into read/write storage.

dbx requires that all the executables comprising the process be loaded into read/write storage so that it can set break points in these executables. When dbx is attached to a running process, this cannot be guaranteed because the process was already running and some executables were already loaded. There are two techniques that will guarantee that all the executables comprising the process are loaded into read-write storage:

1. Specify the `-g` option for the link-editing phase of each executable. After this is done, the executable is always loaded into read/write storage. Because the executable is marked as being serially reusable, this technique works except in cases where the executable must be marked as being reentrant. For example:

- If the executable is to be used by multiple processes in the same user space.
- If the executable is a DLL that is used on more than one thread in a multithreaded program.

In these cases, use the following technique instead:

2. Do not specify the `-g` option during the link-editing phase so that the executable will be marked as being reentrant. Before invoking the program, export the environment variable `_BPX_PTRACE_ATTACH` with a value of YES. After you do this, then executables will be loaded into read/write storage regardless of their reusability attribute.

If you compile an MVS data set source using the `-g` option, you can use `dbx` to perform source-level debugging for the executable file. You must first issue the `dbx use` subcommand to specify a path of double slash (`//`), causing `dbx` to recognize that the symbolic name of the primary source file is an MVS data set. For information on the `dbx` command and its use subcommand, see *z/OS UNIX System Services Command Reference*.

For more information on using `dbx`, see *z/OS UNIX System Services Programming Tools*.

The *z/OS UNIX System Services* web page also has more information about `dbx`. Go to <http://www.ibm.com/servers/eserver/zseries/zos/unix/>.

For more information on the `_BPX_PTRACE_ATTACH` environment variable, see *z/OS UNIX System Services Programming: Assembler Callable Services Reference*.

The `GONUMBER` option is automatically turned on by the `-g` option, but can also be turned on independently. There is no execution path overhead incurred for turning on this option, only some additional space for the saved line number tables.

For 31-bit compiles and In Storage Debug (ISD) information, the `GONUMBER` option generates tables that correspond to the input source file line numbers. These tables make it possible for Debug Tools and for error trace back information in CEE dumps to display the source line numbers. Having source line numbers in CEE dumps improves serviceability costs of applications in production.

Example: The following is an example of compiling with the `GONUMBER` compiler option:

```
c89 -o outfile -Wc,'GONUM' file.c
```

Note: 64-bit compiles do not support `GONUMBER` and line number information will not be available within 64-bit compiled objects.

-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 505 for a description of the usual places.)

You can specify an MVS data set name as an include file search directory. Also, MVS data set names can explicitly be specified on `#include` directives. You can indicate both by specifying a leading double slash (//).

Example: To include the include file DEF that is a member of the MVS PDS ABC.HDRS, code your C or C++ source as follows:

```
#include </'abc.hdrs(def) '>
```

MVS data set include files are handled according to z/OS C/C++ compiler conversion rules (see Usage Note 4 on page 505). When specifying an `#include` directive with a leading double slash (in a format other than `#include</'dsname'>` and `#include</'dd:dsname'>`), 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 506 for a description of the usual places.

`-O`, `-O (-1)`, `-2`, `-3`

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 `-3` gives the highest level of optimization. The default is `-O` (level zero), no optimization and no inlining, when not using IPA (Interprocedural Analysis).

When optimization is specified, the default is ANSIALIAS. The ANSIALIAS default specifies whether type-based aliasing is to be used during optimization. That is, the optimizer assumes that pointers can only be used to access objects of the same type. Type-based aliasing improves optimization. Applications that use pointers that point to objects of a different type will need to specify NOANSIALIAS when the optimization compiler option is specified. If your application works when compiled with

no optimization and fails when compiled with optimization, then try compiling your application with both optimization and NOANSIALIAS compiler options.

Example: The following is an example of a compile with the highest level of optimization and no type-based aliasing:

```
c89 -o outfile -3 -Wc,NOANSIALIAS file.c
```

When optimization is specified, you may want to obtain a report on the amount of inlining performed and increase or decrease the level of inlining. More inlining will improve application performance and increase application memory usage.

Example: The following is an example of a compile with optimization with no report generated, a threshold of 500 abstract code units, and a limit of 2500 abstract code units:

```
c89 -o outfile -2 -Wc,'inline(auto,noreport,500,2500)' file.c
```

When using IPA, the default is -0 (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 481.

If compiling with PDF, the same optimization level must be used in the PDF1 and PDF2 steps.

If you compile your program to take advantage of dbx source-level debugging and specify -g (see the -g option on page 474), 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 “ROSTRING | NOROSTRING” on page 181.

-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 483 for information on the default name of the output file.

Otherwise the -o option specifies the name of the executable file produced during the link-editing phase. The default output file is **a.out**.

- p Ignored by cc. Provided for compatibility with historical implementations of cc. Flagged as an error by c89 and c++.
- q Ignored by cc. Provided for compatibility with historical implementations of cc. Flagged as an error by c89 and c++.
- r Specifies that c89/cc/c++ is to save relocation information about the object files which are processed. When the output file (as specified on -o) is

created, it is not made an executable file. Instead, this output file can later be used as input to c89/cc/c++. This can be used as an alternative to an archive library.

IPA Usage Note:

When using `-r` and link-editing IPA compiled object files, you must link-edit with IPA (see the description of IPA under the `-w` option). However, the `-r` option is typically not useful when creating an IPA optimized program. This is because link-editing with IPA requires that all of the program information is available to the link editor (that is, all of the object files). It is not acceptable to have unresolved symbols, especially the program entry point symbol (which is usually *main*). The `-r` option is normally used when you wish to combine object files incrementally. You would specify some object files during the initial link-edit that uses `-r`. Later, you would specify the output of the initial link-edit, along with the remaining object files in a final link-edit that is done without using `-r`. In such situations where you wish to combine IPA compiled object files, there is an alternative which does not involve the link editor. That alternative is to concatenate the object files into one larger file. This larger file can later be used in a final link-edit, when the remainder of the object files are also made available. (This concatenation can easily be done using the `cp` or `cat` utilities.)

- `-s` Specifies that the compilation phase is to produce a *file.o* file that does *not* include symbolic information, and that the link-editing phase produce an executable that is marked reentrant. This is the default behavior for c89/cc/c++.
- `-U name` Undefines a C or C++ macro specified with *name*. This option affects only macros defined by the `-D` option, including those automatically specified by c89/cc/c++. For information about macros that c89/cc/c++ automatically define, see Usage Note 5 on page 506. Also see Usage Note 13 on page 507.
- `-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 510.) 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 `-0`, `-2`, or `-3` options are specified and cause c89/cc/c++ to use the compiler `INLINE` option, then the inline report is also produced with the compiler listing. Error output continues to be directed to `stderr`. Because this option causes c89/cc/c++ to change the options passed to the steps producing these listings so that they produce more information, it may also result in additional messages being directed to `stderr`. In the case of the compile step, it may also result in the return code of the compiler changing from 0 to 4.
- `-v` This verbose option causes pseudo-JCL to be written to `stdout` before the compiler, assembler, IPA linker, prelinker, and link editor programs are run.

Example: It also causes phaseid information to be emitted in stderr:

```
FSUM0000I Utility(c89) Level(UQ99999)
```

It provides information about exactly which compiler, prelinker, and link editor options are being passed, and also which data sets are being used. If you want to obtain this information without actually invoking the underlying programs, specify the `-v` option more than once on the `c89/cc/c++` command string. For more information about the programs which are executed, see Usage Note 14 on page 508.

`-W phase, option[,option]...`

Specifies options to be passed to the steps associated with the compile, assemble, or link-editing phases of `c89/cc/c++`. The valid phase codes are:

- 0** Specifies the compile phase (used for both non-IPA and IPA compilation).
- a** Specifies the assemble phase.
- c** Same as phase code 0.
- I** Enables IPA (Interprocedural Analysis) optimization.

Unlike other phase codes, the IPA phase code `I` does not require that any additional options be specified, but it does allow them. In order to pass IPA suboptions, specify those suboptions using the IPA phase code.

Example: To specify that an IPA compile should save source line number information, without writing a listing file, specify:

```
c89 -c -W I,list file.c
```

Example: To specify that an IPA link-edit should write the map file to `stdout`, specify:

```
c89 -W I,map file.o
```

l Specifies the link-editing phase.

- To pass options to the prelinker, the first link-editing phase option must be `p` or `P`. All the following options are then prelink options.

Example: To write the prelink map to `stdout`, specify:

```
c89 -W l,p,map file.c
```

Note: The prelinker is no longer used in the link-editing phase in most circumstances. If it is not used, any options passed are accepted but ignored. See the environment variable `{_STEPS}` under “Environment variables” on page 486 for more information about the link-editing phase prelink step.

- To pass options to the IPA linker, the first link-editing phase option must be `i` or `I`. All the following options are then IPA link options.

Example: To specify the size of the SPILL area to be used during an IPA link-edit, you could specify:

```
c89 -W l,I,"spill(256)" file.o
```

- To link-edit a DLL (Dynamic Link Library) and produce a side deck, the link-editing phase option `DLL` must be specified.

Example: To accomplish this task, you could specify:

```
c89 -o outdll -W l,dll file.o
```

Most z/OS 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 485 for information on where the definition side-deck is written. In order for the program to refer to symbols exported by the DLL, it must be compiled with the DLL compiler option.

Example: To compile and link a program into a DLL, you could specify:

```
c89 -o outdll -W c,exportall -W l,dll file.c
```

To subsequently use *file.x* definition side-decks, specify them along with any other *file.o* object files specified for c89/cc/c++ link-editing phase.

Example: To accomplish this task, you could specify:

```
c89 -o myappl -W c,dll myappl.c outdll.x
```

In order to run an application which is link-edited with a definition side-deck, the DLL must be made available (the definition side-deck created along with the DLL is not needed at execution time). When the DLL resides in the 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.

Note: For non-DLL C++ compiles, a dummy definition side file will be allocated to prevent the binder from issuing a warning message. If you do want the binder to issue a warning message when an exported symbol is encountered, specify the `DLL=NO` option for the link-editing phase; for example:

```
c++ -o outfile -W l,dll=no file.C
```

IPA (interprocedural analysis)

IPA optimization is independent from and can be used in addition to the c89/cc/c++ optimization level options (such as -O). IPA optimization can also improve the execution time of your application. IPA is a mechanism for performing optimizations across function boundaries, even across compilation units. It also performs optimizations not otherwise available with the 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 1) option is i or I, then an additional IPA link step is performed prior to the prelink and link-edit steps.

With conventional compilation and link-editing, the object code generation takes place during the compilation phase. With IPA compilation and link-editing, the object code generation takes place during the link-editing phase. Therefore, you might need to request listing information about the program (such as with the -V option) during the link-editing phase.

Unlike the other phase codes, phase code I does not require that any additional options be specified. If they are, they should be specified for both the compilation and link-editing phases.

No additional preparation needs to be done in order to use IPA.

Example: To create the executable myIPApgm using c89 with some existing source program mypgm.c, you could specify:

```
c89 -W I -o myIPApgm mypgm.c
```

When IPA is used with c++, and automatic template generation is being used, phase code I will control whether the automatic template generation compiles are done using IPA. If you do not specify phase code I, then regular compiles will be done. Specifying I as the first option of the link-editing phase option (that is, -W 1,I), will cause the IPA linker to be used, but will not cause the IPA compiler to be used for automatic template generation unless phase code I (that is, -W I) is also specified.

The IPA Profile-Directed Feedback (PDF) option tunes optimizations, where results from sample program execution are used to improve optimization near conditional branches and in frequently executed code sections. The profiling information is placed in the file specified by the PDFNAME(filename) suboption. If PDFNAME(filename) is not specified, the default name of the file containing profile information is PDF.

LP64 The LP64 option instructs the compiler to generate AMODE 64 code utilizing the z/Architecture 64-bit instructions.

To compile 64-bit code, specify the z/OS C/C++ LP64 compiler option.

Example: The following example shows how to compile and bind using the LP64 option:

```
c89 -o -w c,LP64 -W1,LP64 file.c
```

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 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. When XPLINK is specified in the link-editing step, different link-edit libraries will be used.

Example: Here is an example of compiling and link-editing an XPLINK application in one command:

```
c89 -o outxpl -W c,XPLINK -W l,XPLINK,dll file.c
```

In order to execute an XPLINK program, the SCEERUN2 as well as the SCEERUN data set must be in the MVS program search order (see the {_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 `-0`, `-2`, and `-3`)
- Symbolic options (corresponding to `-s` and `-g`); symbolic options can be overridden only when neither `-s` nor `-g` is specified.

Notes:

1. Most compiler, prelinker, and IPA linker options have a positive and negative form. The negative form is the positive with a prepended NO (as in XREF and NOXREF).
2. The compiler `#pragma` options directives as well as any other `#pragma` directives which are overridden by compiler options, will have no effect in source code compiled by `c89/cc/c++`.
3. Link editor options must be specified in the `name=value` format. Both the option `name` and `value` must be spelled out in full. If you do not specify a value, a default value of YES is used, except for the following options, which if specified without a value, have the default values shown here:

ALIASES ALIASES=ALL

COMPAT COMPAT=CURRENT

Note: The binder default is COMPAT=MIN. For downward compatibility (when -Wc, 'target(release)' is used), COMPAT should also be used (for example, -Wl,compat=min, or the specific program object format level supported by the target deployment system, if it is known). For more information, see “z/OS Language Environment downward compatibility” on page 9.

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. Related information about the z/OS C/C++ run-time 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 *z/OS 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. For more information on the prelinker, see Appendix A, “Prelinking and linking z/OS C/C++ programs,” on page 535.
7. Any messages produced by it (CCN messages) are documented in *z/OS C/C++ Messages*.
8. You may see run-time messages (CEE or EDC) in executing your applications. These messages are described in *z/OS Language Environment Debugging Guide*.
9. The link editor (the Program Management binder) is described in *z/OS 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 486 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}.

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 481. 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 477, 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*.

Example: If a partitioned data set is specified, more than one member name may be specified by separating each with a comma (,):

```
c89 //file.OBJ(mem1,mem2,mem3)
```

file.p Specifies the name of a prelinker composite object file produced during the c89/cc/c++ link-editing phase. By default, the composite object file is written to a temporary file. To have the composite object file written to a permanent file, see the environment variable {_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 477, 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 477, for information on the name of the output file.

If the output file is named *a.dll*, then the definition side-deck is named *a.x*, and is always in the working directory. If the output file is named *//a.DLL*, then the definition side-deck is named *//a.EXP*. If the output file specified already has a suffix, that suffix is replaced. Otherwise the suffix is appended.

You can specify an MVS data set name to be link-edited, by preceding the file name with double slash (*//*), in which case the last qualifier of the data set name must be *EXP*.

Example: If a partitioned data set is specified, more than one member name may be specified by separating each with a comma (,):

```
c89 //file.EXP(mem1,mem2,mem3)
```

`-l libname`

Specifies the name of an archive library. c89/cc/c++ searches for the file **liblibname.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 506.

You can also specify an MVS data set; you must specify the full data set name, because there are no rules for searching library directories.

The data set specified must be a C370LIB object library or a load library. If a data set specified as a library has undefined (U) record format, then it is assumed to be a load library. For more information about how load libraries are searched, see Usage Note 7 on page 506.

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 value indicates that c89/cc/c++ should omit the corresponding parameters during dynamic allocation. Numbers in parentheses following the environment variable name correspond to usage notes, which begin on Page 505, 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 {_CLASSLIB_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 variable:

_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 name:**

_CXX_CLASSVERSION

{_CLIB_PREFIX} (14,17)

The prefix for the following named data set used during the compilation phase.

The data set `{_CLIB_PREFIX}.SCNCMP` contains the compiler programs called by `c89/cc/c++`.

The preceding data set contains MVS programs that are invoked during the execution of `c89/cc/c++` and during the execution of a C/C++ application built by `c89/cc/c++`. To be executed correctly, the data set must be made part of the MVS search order. Regardless of the setting of this or any other `c89/cc/c++` environment variable, `c89/cc/c++` does not affect the MVS search order. The data set is listed here for information only, to assist in identifying the correct data set to be added to the MVS program search order.

The following data set is also used:

The data set `{_CLIB_PREFIX}.SCCNOBJ` contains object files required to instrument the code for profile-driven feedback optimization.

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` run-time name to be used, and a non-null value must be a valid Language Environment `NATLANG` run-time option name (Language Environment run-time options are described in *z/OS Language Environment Programming Guide* . The default value is:

"" (null)

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

{_DEBUG_FORMAT} (21)

This variable is used to determine to which debug format (DWARF or ISD) the -g flag is translated. If _DEBUG_FORMAT is set to DWARF, then -g is translated to DEBUG(FORMAT(DWARF)). If _DEBUG_FORMAT is set to ISD, then -g is translated to TEST. The default value is DWARF.

Note: This environment variable only applies to 31-bit compilers.

Actual variable names: _CC_DEBUG_FORMAT, _C89_DEBUG_FORMAT, _CXX_DEBUG_FORMAT

{_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 473 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

{_IL6SYSIX} (7, 16)

The system definition side-deck list that is used to resolve symbols during the IPA link step of the link-editing phase when using LP64 (see the description of LP64 in "Options" on page 473). The default value is whatever {_L6SYSIX} is set to or defaults to.

Actual variable names: _C89_IL6SYSIX, _CC_IL6SYSIX, _CXX_IL6SYSIX

{_IL6SYSLIB} (7, 16)

The system library data set list that is used to resolve symbols during the

IPA link step of the link-editing phase when using LP64 (see the description of LP64 in “Options” on page 473). The default value is whatever `{_L6SYSLIB}` is set to or defaults to.

Actual variable names: `_C89_IL6SYSLIB`, `_CC_IL6SYSLIB`, `_CXX_IL6SYSLIB`

`{_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.ct1"
```

Actual variable names: `_C89_ILCTL`, `_CC_ILCTL`, `_CXX_ILCTL`

`{_ILMSG}` (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_ILMSG`, `_CC_ILMSG`, `_CXX_ILMSG`

`{_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 473). 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 473). 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/cbclib/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

{_L6SYSIX} (7, 16)

The system definition side-deck list that resolves symbols during the link-editing phase when using LP64 (see the description of LP64 in "Options" on page 473). A definition side-deck contains link-editing phase IMPORT control statements, which name symbols that are exported by a DLL. The default value depends on whether or not c++ is used. If c++ is not used, the default value is:

"{_PLIB_PREFIX}.SCEELLIB(CELQS003)"

If c++ is used, the default value is the list concatenation:

"{_PLIB_PREFIX}.SCEELIB(CELQS003,CELQSCPP,C64)"
 "{_CLIB_PREFIX}.SCLBSID(IOSX64)"

Actual variable names: _C89_L6SYSIX, _CC_L6SYSIX, _CXX_L6SYSIX

{_L6SYSLIB} (7, 16)

The system library data set concatenation that is used to resolve symbols during the link-editing step when using LP64 (see the description of LP64 in "Options" on page 473). The default value is the concatenation:

"{_PLIB_PREFIX}.SCEEBND2"
 "{_SLIB_PREFIX}.CSSLIB"

Actual variable names: _C89_L6SYSLIB, _CC_L6SYSLIB, _CXX_L6SYSLIB

{_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 473). 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 473). A definition side-deck contains link-editing phase IMPORT control statements naming symbols which are exported by a DLL. The default value depends on whether or not c++ is being used. For 32-bit objects, if c++ is not being used, the default value is the list:

```
"{_PLIB_PREFIX}.SCEELIB(CELHS003,CELHS001)"
```

For 32-bit objects, 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)"
"{_CLASSLIB_PREFIX}.SCLBSID(IOSTREAM,COMPLEX)"
```

For 32-bit objects, if c++ is being used with `{_PVERSION}` and `{_CLASSVERSION}` set to a release prior to z/OS Version 1 Release 2 for a 32-bit program, the default value is the list concatenation:

```
"{_PLIB_PREFIX}.SCEELIB(CELHS003,CELHS001,CELHSCPP)"
"{_CLASSLIB_PREFIX}.SCLBSID(ASCCOLL,COMPLEX,IOSTREAM)"
```

Note: For 64-bit objects, see `_L6SYSIX`.

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 data sets 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 data sets 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_STORCLAS

{_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

{_NOCMDOPTS} (27)

Controls how the compiler processes the default options set by c89. Setting this variable to 1, reverts the compiler to the behavior that was available prior to z/OS V1R5, when the compiler was unable to distinguish between the c89 defaults and the user-specified options. Setting this variable to 0, results in the default behavior where the compiler is now able to recognize c89 defaults. The default value is:

"0"

Actual variable names: _C89_NOCMDOPTS, _CC_NOCMDOPTS, _CXX_NOCMDOPTS

{_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 run-time 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}.SCEE0BJ` and `{_PLIB_PREFIX}.SCEECPP` contain run-time library bindings which exploit constructed reentrancy, used during the link-editing phase of non-XPLINK programs.
- The data set `{_PLIB_PREFIX}.SCEELKEX` contains C run-time library bindings which exploit L-names used during the link-editing phase of non-XPLINK programs. For more information about L-names, see usage note 23 on page 510.
- The data set `{_PLIB_PREFIX}.SCEELKED` contains all other Language Environment run-time library bindings, used during the link-editing phase of non-XPLINK programs.
- The data set `{_PLIB_PREFIX}.SCEEBIND` contains all static Language Environment run-time library bindings, used during the link-editing phase of XPLINK programs.
- The data set `{_PLIB_PREFIX}.SCEEBND2` contains all static Language Environment run-time library bindings, used during the link-editing phase of XPLINK programs.
- The data set `{_PLIB_PREFIX}.SCEELIB` contains the definition side-decks for the run-time library bindings, used during the link-editing phase of XPLINK programs.

The following data sets are also used:

- The data sets `{_PLIB_PREFIX}.SCEERUN` and `{_PLIB_PREFIX}.SCEERUN2` contains the run-time 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

{_PMSG} (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_PMSG, _CC_PMSG, _CXX_PMSG

{_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)"
"{_CLASSLIB_PREFIX}.SCLBSID(IOSTREAM,COMPLEX)"

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:

"{_CLASSLIB_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}.SCEE0BJ"
```

If c++ is being used, the default value is the list:

```
"{_PLIB_PREFIX}.SCEE0BJ"  
" {_PLIB_PREFIX}.SCEECPP"
```

Actual variable names: _C89_PSYSLIB, _CC_PSYSLIB, _CXX_PSYSLIB

{_PVERSION} (26)

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++ processing phases. These named data sets include those required for use of the C/C++ Run-Time Library as well as the ISO C++ Library. It also sets default values for other environment variables.

The format of this variable is the same as the result of the Language Environment C/C++ Run-Time Library function `_librel()`. See *z/OS 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 481.

Considering this variable to be a set of 32 switches, numbered left-to-right from 0 to 31, the steps corresponding to each of the switches are as follows:

0-27	Reserved
28	TEMPINC/IPATEMP
29	IPALINK
30	PRELINK
31	LINKEDIT

Example: To override the default behavior of c89/cc/c++ and cause the prelinker step to be run (this is also the default when the output file is a data set which is not a PDSE), set this variable to:

"0xffffffff" or the equivalent, -1

The default value when the output file is an HFS file or a PDSE data set is:

"0xffffffffd" 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.

Example: To capture the prelinker output, set this variable to:

"0xffffffff" 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

c89, cc, and c++

{_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 506).

libm.a C/C++ Runtime Library math function library (see Usage Note 7 on page 506).

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 505).

/lib The usual place to search for run-time library bindings (see Usage Note 7 on page 506).

/usr/lib

The usual place to search for run-time library bindings (see Usage Note 7 on page 506).

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 hierarchical files (and not to data sets).
 - If the compile or assemble is not successful, the corresponding object file (*file.o*) is always removed.
 - If the DLL option is passed to the link-editing phase, and afterwards the *file.x* file exists but has a size of zero, then that file is removed.
4. MVS data sets may be used as the usual place to resolve C and C++ #include directives during compilation.

Such data sets are installed with 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 data set being searched with these qualifiers.

In both cases the data set name and member name are converted to uppercase and underscores (_) are changed to at signs (@).

If the include (header) files provided by 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 related compiler options are set according to whether cc, c89, or c++ (cxx) was invoked. These options affect various aspects of the compilation, such as z/OS C/C++ predefined macros, which are used like feature test macros to control which symbols are made visible in a source file (typically a header file), but are normally not defined or undefined except by this compiler option. They can also affect the language rules used by the compiler. For more information about z/OS C/C++ predefined macros, see *z/OS C/C++ Language Reference*. The options are shown here in a syntax that the user can specify on the c89/cc/c++ command line to override them:

- c89 (also c++ (cxx) when using a C++ compiler older than z/OS v1r2)
 - W "c,langlvl(ansi),noupconv"
- c++ (cxx)
 - W "c,langlvl(extended,nolibext,nolonglong)"
- cc
 - W "c,langlvl(commonc),upconv"

7. By default the usual place for the –L option search is the **/lib** directory followed by the **/usr/lib** directory. See the description of environment variable **{_LIBDIRS}** for information on customizing the default directories to search.

The archive libraries **libc.a** and **libm.a** exist as files in the usual place for consistency with other implementations. However, the run-time library bindings are not contained in them.

Instead, MVS data sets installed with the Language Environment run-time library are used as the usual place to resolve run-time library bindings. In the final step of the link-editing phase, any MVS load libraries specified on the –l operand are searched in the order specified, followed by searching these data sets. See the **{_PLIB_PREFIX}** description, as well as descriptions of the environment variables featured in the following list.

Note: 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 473.

```
{_ILSYSLIB}
{_ILSYSIX}
{_LSYSLIB}
{_PSYSIX}
{_PSYSLIB}
```

This list of environment variables affects the link-editing phase of c89, but only for ILP32 XPLINK link-editing. See XPLINK (Extra Performance Linkage) in “Options” on page 473.

```
{_ILXSYSLIB}
```

```
{_ILXSYSIX}
{_LXSYSLIB}
{_LXSYSIX}
```

This list of environment variables affects the link-editing phase of c89, but only for LP64 link-editing. See the description of LP64 in “Options” on page 473.

```
{_IL6SYSLIB}
{_IL6SYSIX}
{_L6SYSLIB}
{_L6SYSIX}
```

8. Because archive library files are searched when their names are encountered, the placement of `-l` operands and *file.a* operands is significant. You may have to specify a library multiple times on the command string, if subsequent specification of *file.o* files requires that additional symbols be resolved from that library.
9. When the prelinker is used during the link-editing phase, you cannot use as input to c89/cc/c++ an executable file produced as output from a previous use of c89/cc/c++. The output of c89/cc/c++ when the `-r` option is specified (which is not an executable file) may be used as input.
10. All MVS data sets used by c89/cc/c++ must be cataloged (including the system data sets installed with the z/OS C/C++ compiler and the Language Environment run-time library).
11. c89/cc/c++ operation depends on the correct setting of their installation and configuration environment variables (see “Environment variables” on page 486). 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`, `-0`, `-1`, `-2`, `-3`, `-V`, `-c`, `-g` and `-s` (also ignores any *file.s* files).
 - `-g` Overrides `-0`, `-0`, `-1`, `-2`, `-3`, and `-s`.
 - `-s` Overrides `-g` (the last one specified is honored).
 - `-0`, `-0`, `-1`, `-2`, `-3`, `-V`, `-c`
All are honored if not overridden. `-0`, `-0`, `-1`, `-2`, `-3` and 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}
|     {_IL6SYSIX}
|     {_IL6SYSLIB}
|     {_ILSYSIX}
|     {_ILSYSLIB}
|     {_ILXSYSIX}
|     {_ILXSYSLIB}
|     {_L6SYSIX}
|     {_L6SYSLIB}
|     {_LSYSLIB}
|     {_LXSYSIX}
|     {_LXSYSLIB}
|     {_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 on reentrancy, 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. The {_CVERSION}, {_PVERSION} and {_CLASSVERSION} environment variables are set to a hex string in the format 0xPVVRRMMM where P is product, VV is version, RR is release and MMM is modification level. For example, the {_CVERSION} and {_CLASSVERSION} for the z/OS V1R2 compiler is 0x41020000. The {_CVERSION} and {_CLASSVERSION} for the OS/390 V2R10 compiler is 0x220A0000.
27. c89 passes some options to the compiler to ensure that expected behavior is achieved; for example, POSIX behavior. These options are passed onto the compiler as defaults that the user can overwrite. When default options passed by c89 are in conflict with options and/or pragmas that the user specified, the compiler issues warning and/or severe error messages. Since the user did not specify options that c89 passed as defaults, these messages may confuse the user. Prior to the z/OS V1R5 release, the compiler was unable to differentiate between the options that c89 passed as defaults and the user-specified options so it was unable to correctly resolve conflicting pragma/option combinations. In some cases, the compiler would overwrite pragmas with the options that c89 passed as defaults thus limiting a user's ability to use pragmas. As of V1R5,

the compiler is now able to recognize c89 defaults and avoid confusion from messages for options, which were not explicitly specified by the user, and overriding pragmas, when the user did not explicitly request it. Most users will benefit from this feature so it is the default behavior. To enable the old behavior, environment variable `_NOCMDOPTS` must have a non-zero value.

Example: The following sequence will preserve the old behavior:

```
export _C89_NOCMDOPTS=1
c89 -o hello hello.c
```

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: <ul style="list-style-type: none"> • Archive library was not in any of the library directories specified. • Archive library was incorrectly specified, or was not specified, following the <code>-l</code> 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: <ul style="list-style-type: none"> • 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 <code>-2</code> option, when processing assembler source files, and <code>-r</code> 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

c89, cc, and c++

- IPA optimization support
- The behavior of the `-o` option in combination with the `-c` option and a single source file.

Features have been added to z/OS releases, which have made it easier to port applications from other platforms to z/OS and improve performance. For compatibility reasons, these portability and performance enhancements could not be made the default. If you are porting an application from another platform to z/OS, you may want to start by specifying the following options:

```
c89 -o HelloWorld -2 -Wc,NOANSIALIAS -Wc,XPLINK\  
-Wl,XPLINK -Wc,'FLOAT(IEEE)' -Wc,'GONUM' HelloWorld.c
```

Note: The above string is one line (had to be split to fit page). A space exists between `-Wc,XPLINK` and `-Wl,XPLINK`.

Related information

See the information on the following utilities in *z/OS UNIX System Services Command Reference*: `ar`, `dbx`, `file`, `lex`, `makedepend`, `nm`, `strings`, `strip`, `yacc`

Chapter 19. xlc — Compiler invocation using a customizable configuration file

Format

```
xlc | x1C | xlc++ | cc | c89 | cxx | c++ |  
xlc_x | x1C_x | xlc++_x | cc_x | c89_x | cxx_x | c++_x |  
xlc_64 | x1C_64 | xlc++_64 | cc_64 | c89_64 | cxx_64 | c++_64  
[option | input file]...
```

Description

xlc is a utility that uses an external configuration file to control the invocation of the compiler. xlc and related commands compile C and C++ source files. They also process assembler source files and object files.

Note: Unless the `-c` option is specified, xlc calls the binder to produce an executable module.

All commands accept the following input files with their default HFS and host suffixes:

- filename with `.C` suffix (C++ source file)
- filename with `.c` suffix (C source file)
- filename with `.i` suffix (preprocessed C or C++ source file)
- filename with `.o` suffix (object file for binder/IPA link)
- filename with `.s` suffix (assembler source file)
- filename with `.a` suffix (archive library)
- filename with `.p` suffix (prelinker output file for the binder/IPA Link)
- filename with `.l` suffix (IPA Link output file for the binder)
- filename with `.x` suffix (definition side-file or side deck)
- filename with `.CXX` suffix (C++ source host file)
- filename with `.C` suffix (C source host file)
- filename with `.CEX` suffix (preprocessed C or C++ source host file)
- filename with `.OBJ` suffix (object host file for the binder/IPA Link)
- filename with `.ASM` suffix (assembler source host file)
- filename with `.LIB` suffix (host archive library)
- filename with `.CPOBJ` suffix (prelinker output host file for the binder/IPA Link)
- filename with `.IPA` suffix (IPA Link output host file for the binder)
- filename with `.EXP` suffix (host definition side-file or side deck)

The xlc utility invokes the assembler, the C/C++ compiler, and the binder. Invocation of the compiler and the binder is described in “Invoking the compiler” on page 522 and “Invoking the binder” on page 523.

Invocation commands

The `xlc` utility provides two basic compiler invocation commands, `xlc` and `x1C` (`x1c++`), along with several other compiler invocation commands to support various C/C++ language levels and compilation environments. In most cases, you would use the `xlc` command to compile C source files and `x1C` (`x1c++`) command to compile C++ source files.

You can however, use other forms of the command if your particular environment requires it. The various compiler invocation commands for C are:

- `xlc`
- `cc`
- `c89`
- `xlc_x`
- `cc_x`
- `c89_x`
- `xlc_64`
- `cc_64`
- `c89_64`

The various compiler invocation commands for C++ are:

- `x1C` (`x1c++`)
- `cxx`
- `c++`
- `x1C_x` (`x1c++_x`)
- `c++_x`
- `cxx_x`
- `x1C_64` (`x1c++_64`)
- `c++_64`
- `cxx_64`

The two basic compiler invocation commands appear as the first entry of each list item shown above. Select an invocation command using the following criteria:

xlc Invokes the compiler for C source files with a default language level of ANSI, and compiler option `-qansialias` to allow type-based aliasing.

x1C (x1c++)

Invokes the compiler so that source files are compiled as C++ language source code.

Files with `.c` suffixes, assuming you have not used the `-+` compiler option, are compiled as C language source code with a default language level of ANSI, and compiler option `-qansialias` to allow type-based aliasing.

If any of your source files are C++, you must use this invocation to link with the correct run-time libraries.

cc Invokes the compiler for C source files with a default language level of extended and compiler options `-qnoro` and `-qnoroconst` (to provide placement of string literals or constant values in read/write storage).

Use this invocation for legacy C code that does not require compliance with ANSI C. This invocation is intended to provide the same compiler behavior as when invoked by the cc command name of the c89 utility.

c89 Invokes the compiler for C source files, with a default language level of ANSI, and specifies compiler options `-qansialias` (to allow type-based aliasing) and `-qno1ong1ong` (disabling use of long long), and sets `-D_ANSI_C_SOURCE` (for ANSI-compliant headers). Use this invocation for strict conformance to the ANSI standard (ISO/IEC 9899:1990). This invocation is intended to provide the same compiler behavior as when invoked by the c89 command name of the c89 utility.

cxx/c++

`cxx` and `c++` invoke the compiler for C++ language source code. Both are intended to provide the same compiler behavior as when invoked using the `cxx` and `c++` command names of the c89 utility.

_x Command invocations using command names with suffix `_x` are the same as invocations using names without suffixes, except the `-qxplink` option is also specified and appropriate XPLINK libraries are used in the link step. If you are building an XPLINK application, you must use command names with suffix `_x` to link with the correct run-time libraries.

_64 Command invocations using command names with suffix `_64` are the same as invocations using names without suffixes, except the `-q64` option is also specified and appropriate 64-bit libraries are used in the link step. If you are building a 64-bit application, you must use command names with suffix `_64` to link with the correct run-time libraries.

Setting up the compilation environment

Before you compile your C and C++ programs, you must set up the environment variables and the configuration file for your application. For more information on the configuration file, see “Setting up a configuration file” on page 516.

Environment variables

You can use environment variables to specify necessary system information.

Setting environment variables

Different commands are used to set the environment variables depending on whether you are using the z/OS UNIX System Services shell (`sh`), which is based on the Korn Shell and is upward-compatible with the Bourne shell, or `tcsh` shell, which is upward-compatible with the C shell. To determine the current shell, use the `echo` command, which is `echo $SHELL`.

The z/OS UNIX System Services shell path is `/bin/sh`. The `tcsh` shell path is `/bin/tcsh`.

For more information about the `NLSPATH` and `LANG` environment variables, see *z/OS C/C++ Programming Guide* and *z/OS UNIX System Services Command Reference*.

Setting environment variables in z/OS shell

The following statements show how you can set environment variables in the z/OS shell:

```
LANG=En_US
NLSPATH=/usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/%L/%N.cat
PATH=/bin:/usr/lpp/clang/bin/xlc/bin${PATH:+:${PATH}}
export LANG NLSPATH PATH
```

To set the variables so that all users have access to them, add the commands to the file `/etc/profile`. To set them for a specific user only, add the commands to the `.profile` file in the user's home directory. The environment variables are set each time the user logs in.

Setting environment variables in tcsh shell

The following statements show how you can set environment variables in the tcsh shell:

```
setenv LANG En_US
setenv NLSPATH /usr/lib/nls/msg/%L/%N:/usr/lib/nls/msg/%L/%N.cat
setenv PATH /bin:/usr/lpp/cbc/lib/xlc/bin${PATH:+:${PATH}}
```

To set the variables so that all users have access to them, add the commands to the file `/etc/csh.cshrc`. To set them for a specific user only, add the commands to the `.tcshrc` file in the user's home directory. The environment variables are set each time the user logs in.

Setting environment variables for the message file

Before using the compiler, you must install the message catalogs and set the environment variables:

LANG Specifies the national language for message and help files.

NLSPATH

Specifies the path name of the message and help files.

XL_CONFIG

Specifies the name of an alternative configuration file (`.cfg`) for the `xlc` utility. Note: For the syntax of the configuration file, see the description for the `-F` flag option in "Flag options syntax" on page 525.

The `LANG` environment variable can be set to any of the locales provided on the system. See the description of locales in *z/OS C/C++ Programming Guide* for more information.

The national language code for United States English may be `En_US` or `C`. If the Japanese message catalog has been installed on your system, you can substitute `Ja_JP` for `En_US`.

To determine the current setting of the national language on your system, see the output from both of the following echo commands:

- `echo $LANG`
- `echo $NLSPATH`

The `LANG` and `NLSPATH` environment variables are initialized when the operating system is installed, and may differ from the ones you want to use.

Setting up a configuration file

The configuration file specifies information that the compiler uses when you invoke it. This file defines values used by the compiler to compile C or C++ programs. You can make entries to this file to support specific compilation requirements or to support other C or C++ compilation environments.

A configuration file is an HFS file consisting of named sections called stanzas. Each stanza contains keywords called configuration file attributes, which are assigned values. The attributes are separated from their assigned value by an equal sign. A

stanza can point to a default stanza by specifying the "use" keyword. This allows specifying common attributes in a default stanza and only the deltas in a specific stanza, referred to as the local stanza.

For any of the supported attributes not found in the configuration file, the xlc utility uses the built-in defaults. It uses the first occurrence in the configuration file of a stanza or attribute it is looking for. Unsupported attributes, and duplicate stanzas and attributes are not diagnosed.

Configuration file attributes

A stanza in the configuration file can contain the following attributes:

as	Path name to be used for the assembler. The default is /bin/c89.
asopt	The list of options for the assembler and not for the compiler. These override all normal processing by the compiler and are directed to the assembler specified in the as attribute. Options are specified following the c89 utility syntax.
asuffix	The suffix for archive files. The default is a.
asuffix_host	The suffix for archive data sets. The default is LIB.
ccomp	The C compiler. The default is usr/lpp/cbc/lib/xlc/exe/ccndrvr.
cinc	A comma separated list of directories or data set wild cards used to search for C header files. The default for this attribute is: -I//'CEE.SCEEH.+'
cppcomp	The C++ compiler. The default is /usr/lpp/cbc/lib/xlc/exe/ccndrvr.
cppinc	A comma separated list of directories or data set wild cards used to search for C++ header files. The default for this attribute is: -I//'CEE.SCEEH.+','-I//'CBC.SCLBH.+'
csuffix	The suffix for source programs. The default is c (lowercase c).
csuffix_host	The suffix for C source data sets. The default is C (uppercase C).
cxxsuffix	The suffix for C++ source files. The default is C (uppercase C).
cxxsuffix_host	The suffix for C++ source data sets. The default is CXX.
exportlist	A colon separated list of data sets with member names indicating definition side-decks to be used to resolved symbols during the link-editing phase. The default for this attribute depends on the type of stanza. Non-XPLINK C stanzas do not have a default. The default for non-XPLINK C++ stanzas is: CEE.SCEELIB(C128N):CBC.SCLBSID(IOSTREAM,COMPLEX) The default for XPLINK C stanzas is: CEE.SCEELIB(CELHS003,CELHS001) The default for XPLINK C++ stanzas is: CEE.SCEELIB(CELHS003,CELHSCPP,CELHS001,C128):CBC.SCLBSID(IOSTREAM,COMPLEX)

xlc and xlc

		The default for 64-bit C stanzas is:
		CEE.SCEELIB(CELQS003)
		The default for 64-bit C++ stanzas is:
		CEE.SCEELIB(CELQS003,CELQSCPP,C64):CBC.SCLBSID(I0SQ64)
	isuffix	The suffix for C preprocessed files. The default is i.
	isuffix_host	The suffix for C preprocessed data sets. The default is CEX.
	ilsuffix	The suffix for IPA output files. The default is I.
	ilsuffix_host	The suffix for IPA output data sets. The default is IPA.
	ixxsuffix	The suffix for C++ preprocessed files. The default is i.
	ixxsuffix_host	The suffix for C++ preprocessed data sets. The default is CEX.
	ld	The path name to be used for the binder. The default is /bin/c89.
	libraries2	libraries2 specifies the libraries that the binder is to use at bind time. The default is empty.
	options	A string of option flags, separated by commas, to be processed by the compiler as if they had been entered on the command line.
	osuffix	The suffix for object files. The default is .o.
	osuffix_host	The suffix for object data sets. The default is OBJ.
	psuffix	The suffix for prelinked files. The default is p.
	psuffix_host	The suffix for prelinked data sets. The default is CPOBJ.
	ssuffix	The suffix for assembler files. The default is .s.
	ssuffix_host	The suffix for assembler data sets. The default is ASM.
	steplib	A colon separated list of data sets or keyword NONE used to set the STEPLIB environment variable. The default is NONE, which causes all programs to be loaded from LPA or linklist.
	syslib	A colon separated list of data sets used to resolve run-time library references. Data sets from this list are used to construct the SYSLIB DD for the IPA Link and the binder invocation. The default for this attribute depends on the type of stanza.
		The default for non-XPLINK stanzas is:
		CEE.SCEELKEX:CEE.SCEELKED:CBC.SCCNOBJ:SYS1.CSSLIB
		The default for XPLINK and 64-bit stanzas is:
		CEE.SCEEBND2:CBC.SCCNOBJ:SYS1.CSSLIB
	sysobj	A colon separated list of data sets containing object files used to resolve run-time library references. Data sets from this list are used to construct the LIBRARY control statements and the SYSLIB DD for the IPA Link and the binder invocation. This attribute is only valid for non-XPLINK stanzas. For XPLINK stanzas, it can be omitted or set to NONE.
		The default for non-XPLINK stanzas is:
		CEE.SCEE0BJ:CEE.SCEE0CPP
		The default for XPLINK and 64-bit stanzas is:

		NONE
	use	Values for attributes are taken from the named stanza and from the local stanza. For single-valued attributes, values in the use stanza apply if no value is provided in the local, or default stanza. For comma-separated lists, the values from the use stanza are added to the values from the local stanza.
	x1C	The path name of the C++ compiler invocation command. The default is <code>/usr/lpp/cbclib/x1c/bin/x1c</code> .
	x1Ccopt	A string of option flags, separated by commas, to be processed when the x1C command is used for compiling a C file.
	xsuffix	The suffix for definition side-deck files. The default is <code>x</code> .
	xsuffix_host	The suffix for definition side-deck data sets. The default is <code>EXP</code> .

Tailoring a configuration file

The default configuration file is installed in `/usr/lpp/cbclib/x1c/etc/x1c.cfg`.

You can copy this file and make changes to the copy to support specific compilation requirements or to support other C or C++ compilation environments. The `-F` option is used to specify a configuration file other than the default. For example, to make `-qnor0` the default for the x1C compiler invocation command, add `-qnor0` to the x1C stanza in your copied version of the configuration file.

You can link the compiler invocation command to several different names. The name you specify when you invoke the compiler determines which stanza of the configuration file the compiler uses. You can add other stanzas to your copy of the configuration file to customize your own compilation environment.

Example: You can use the `-F` option with the compiler invocation command to make links to select additional stanzas or to specify a stanza or another configuration file:

```
x1C myfile.C -Fmyconfig:SPECIAL
```

would compile `myfile.C` using the `SPECIAL` stanza in a `myconfig` configuration file that you had created.

Default configuration file

The default configuration file, (`/usr/lpp/cbclib/x1c/etc/x1c.cfg`), specifies information that the compiler uses when you invoke it. This file defines values used by the compiler to compile C or C++ programs. You can make entries to this file to support specific compilation requirements or to support other C or C++ compilation environments. Options specified in the configuration file override the default settings of the option. Similarly, options specified in the configuration file are in turn overridden by options set in the source file and on the command line. Options that do not follow this scheme are listed in “Specifying compiler options” on page 528.

Example: The following example shows a default configuration file:

```
*
* FUNCTIONS: z/OS 1.6 C/C++ Compiler Configuration file
*
* (C) COPYRIGHT International Business Machines Corp. 2004
* All Rights Reserved
* Licensed Materials - Property of IBM
*
```

xlc and xlc

```
* US Government Users Restricted Rights - Use, duplication or
* disclosure restricted by GSA ADP Schedule Contract with IBM Corp.
*
* C compiler, extended mode
xlc: use = DEFLT
      options = -qansialias,-qlanglvl=extended
      sysobj = cee.sceobj:cee.sceecpp
      syslib = cee.sceelkex:cee.sceelked:cbc.sccnobj:sys1.csslib

* XPLINK C compiler, extended mode
xlc_x: use = DEFLT
        options = -qlanglvl=extended,-qxplink
        exportlist = cee.sceelib(celhs003,celhs001)

* 64 bit C compiler, extended mode
xlc_64: use = DEFLT
          options = -qlanglvl=extended,-q64
          exportlist = cee.sceelib(celqs003)

* C compiler, common usage C
cc: use = DEFLT
     options = -qnoro,-qnoroconst,-qlanglvl=extended
     sysobj = cee.sceobj:cee.sceecpp
     syslib = cee.sceelkex:cee.sceelked:cbc.sccnobj:sys1.csslib

* XPLINK C compiler, common usage C
cc_x: use = DEFLT
      options = -qnoro,-qnoroconst,-qlanglvl=extended,-qxplink
      exportlist = cee.sceelib(celhs003,celhs001)

* 64 bit C compiler, common usage C
cc_64: use = DEFLT
        options = -qnoro,-qnoroconst,-qlanglvl=extended,-q64
        exportlist = cee.sceelib(celqs003)

* Strict ANSI C compiler
c89: use = DEFLT
      options = -D_ANSI_C_SOURCE,-qansialias,-qolonglong,
-qstrict_induction
      sysobj = cee.sceobj:cee.sceecpp
      syslib = cee.sceelkex:cee.sceelked:cbc.sccnobj:sys1.csslib

* XPLINK Strict ANSI C compiler
c89_x: use = DEFLT
        options = -D_ANSI_C_SOURCE,-qansialias,-qolonglong,
-qstrict_induction,-qxplink
        exportlist = cee.sceelib(celhs003,celhs001)

* 64 bit Strict ANSI C compiler
c89_64: use = DEFLT
          options = -D_ANSI_C_SOURCE,-qansialias,-qolonglong,
-qstrict_induction,-q64
          exportlist = cee.sceelib(celqs003)

* ANSI C++ compiler
cxx: use = DEFLT
     xlc = /usr/lpp/cbclib/xlc/bin/.orig/xlc
     ipa = /bin/cxx
     ld = /bin/cxx
     exportlist = cee.sceelib(c128n):cbc.sclbsid(iostream,complex)
     sysobj = cee.sceobj:cee.sceecpp
     syslib = cee.sceelkex:cee.sceelked:cbc.sccnobj:sys1.csslib

* XPLINK ANSI C++ compiler
cxx_x: use = DEFLT
        xlc = /usr/lpp/cbclib/xlc/bin/.orig/xlc
        ipa = /bin/cxx
```

```

        ld          = /bin/cxx
        options     = -qxplink
        exportlist  = cee.sceelib(celhs003,celhs001,celhscpp,c128):
cbc.sclbsid(iostream,complex)

* 64 bit ANSI C++ compiler
cxx_64: use       = DEFLT
        xlc        = /usr/lpp/cbclib/xlc/bin/.orig/xlc
        ipa        = /bin/cxx
        ld         = /bin/cxx
        options    = -q64
        exportlist = cee.sceelib(celqs003,cleqscpp,c64):cbc.sclbsid(iosx64)

* ANSI C++ compiler
c++:   use       = DEFLT
        xlc        = /usr/lpp/cbclib/xlc/bin/.orig/xlc
        ipa        = /bin/cxx
        ld         = /bin/cxx
        exportlist = cee.sceelib(c128n):cbc.sclbsid(iostream,complex)
        sysobj     = cee.sceeobj:cee.sceecpp
        syslib     = cee.sceelkex:cee.sceelked:cbc.sccnobj:sys1.csslib

* XPLINK ANSI C++ compiler
c++_x: use       = DEFLT
        xlc        = /usr/lpp/cbclib/xlc/bin/.orig/xlc
        ipa        = /bin/cxx
        ld         = /bin/cxx
        options    = -qxplink
        exportlist = cee.sceelib(celhs003,celhs001,celhscpp,c128):
cbc.sclbsid(iostream,complex)

* 64 bit ANSI C++ compiler
c++_64: use      = DEFLT
        xlc        = /usr/lpp/cbclib/xlc/bin/.orig/xlc
        ipa        = /bin/cxx
        ld         = /bin/cxx
        options    = -q64
        exportlist = cee.sceelib(celqs003,cleqscpp,c64):cbc.sclbsid(iosx64)

* C++ compiler, extended mode
xlc:   use       = DEFLT
        xlc        = /usr/lpp/cbclib/xlc/bin/.orig/xlc
        ipa        = /bin/cxx
        ld         = /bin/cxx
        options    = -qlanglvl=extended
        exportlist = cee.sceelib(c128n):cbc.sclbsid(iostream,complex)
        sysobj     = cee.sceeobj:cee.sceecpp
        syslib     = cee.sceelkex:cee.sceelked:cbc.sccnobj:sys1.csslib

* XPLINK C++ compiler, extended mode
xlc_x: use       = DEFLT
        xlc        = /usr/lpp/cbclib/xlc/bin/.orig/xlc
        ipa        = /bin/cxx
        ld         = /bin/cxx
        options    = -qlanglvl=extended,-qxplink
        exportlist = cee.sceelib(celhs003,celhs001,celhscpp,c128):
cbc.sclbsid(iostream,complex)

* 64 bit C++ compiler, extended mode
xlc_64: use      = DEFLT
        xlc        = /usr/lpp/cbclib/xlc/bin/.orig/xlc
        ipa        = /bin/cxx
        ld         = /bin/cxx
        options    = -qlanglvl=extended,-q64
        exportlist = cee.sceelib(celqs003,cleqscpp,c64):cbc.sclbsid(iosx64)

* C++ compiler, extended mode

```

xlc and xlc

```
xlc++: use          = DEFLT
        xlc         = /usr/lpp/cbclib/xlc/bin/.orig/xlc
        ipa         = /bin/cxx
        ld          = /bin/cxx
        options     = -qlanglvl=extended
        exportlist  = cee.sceelib(c128n):cbc.sclbsid(iostream,complex)
        sysobj      = cee.sceobj:cee.scecpp
        syslib      = cee.sceekex:cee.sceeked:cbc.sccnobj:sys1.csslib

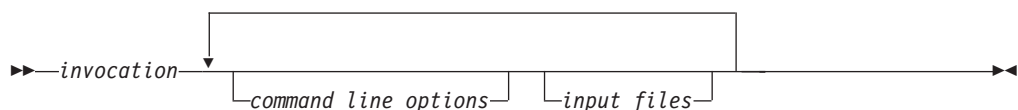
* XPLINK C++ compiler, extended mode
xlc+_x: use          = DEFLT
        xlc         = /usr/lpp/cbclib/xlc/bin/.orig/xlc
        ipa         = /bin/cxx
        ld          = /bin/cxx
        options     = -qlanglvl=extended,-qxplink
        exportlist  = cee.sceelib(celhs003,celhs001,celhscpp,c128):
cbc.sclbsid(iostream,complex)

* 64 bit C++ compiler, extended mode
xlc+_64: use         = DEFLT
        xlc         = /usr/lpp/cbclib/xlc/bin/.orig/xlc
        ipa         = /bin/cxx
        ld          = /bin/cxx
        options     = -qlanglvl=extended,-q64
        exportlist  = cee.sceelib(celqs003,celqscpp,c64):cbc.sclbsid(iosx64)

* common definitions
DEFLT: cppcomp      = /usr/lpp/cbclib/xlc/exe/ccndrvr
        ccomp       = /usr/lpp/cbclib/xlc/exe/ccndrvr
        ipacomp     = /usr/lpp/cbclib/xlc/exe/ccndrvr
        ipa         = /bin/c89
        as          = /bin/c89
        ld          = /bin/c89
        xlc         = /usr/lpp/cbclib/xlc/bin/xlc
        sysobj      = NONE
        syslib      = cee.sceebnd2:cbc.sccnobj:sys1.csslib
        steplib     = NONE
        cinc        = -I//'cee.sceeh.+'
        cppinc      = -I//'cee.sceeh.+',-I//'cbc.sclbh.+'
        xlcCopt     = -D_XOPEN_SOURCE
        options     = -qlocale=posix
```

Invoking the compiler

The z/OS C/C++ compiler is invoked using the following syntax, where *invocation* can be replaced with any valid z/OS C/C++ invocation command:



The parameters of the compiler invocation command can be names of input files, compiler options, and linkage-editor options. Compiler options perform a wide variety of functions such as setting compiler characteristics, describing object code and compiler output to be produced, and performing some preprocessor functions.

To compile without binding, use the `-c` compiler option. The `-c` option stops the compiler after compilation is completed and produces as output, an object file `file_name.o` for each `file_name.c` input source file, unless the `-o` option was used to

specify a different object filename. The binder is not invoked. You can bind the object files later using the invocation command, specifying the object files without the `-c` option.

Notes:

1. Any object files produced from an earlier compilation with the same name as expected object files in this compilation are deleted as part of the compilation process, even if new object files are not produced.
2. By default, the invocation command calls both the compiler and the binder. It passes binder options to the binder. Consequently, the invocation commands also accept all binder options.

Invoking the binder

All invocation commands invoke the binder using the `c89` utility, so all binder options must follow the syntax supported by the `c89` utility. Standard libraries required to bind your program are controlled by the `sysobj`, `syslib`, and `exportlist` attributes in the configuration file.

The specified object files are processed by the binder to create one executable file. Invoking the compiler with one of the invocation commands, automatically calls the binder unless you specify one of the following compiler options: `-E`, `-c`, `-P`, `-qsyntaxonly`, `-qpponly`, or `-#`.

All input and output files supported by the `c89` utility are valid for all invocation commands.

Supported options

In addition to `-W` syntax for specifying keyword options, the `xlc` utility supports AIX `-q` options syntax and several new flag options.

`-q` options syntax

All compiler options can be specified using the `-q` syntax with their z/OS names, except for the following options which must use AIX equivalent options. This ensures portability from z/OS to AIX.

- `ATTRIBUTE` (`-qattr`)
- `BITFIELD` (`-qbitfields`)
- `CHECKOUT` (`-qinfo`)
- `ENUMSIZE` (`-qenum`)
- `EXH` (`-qeh`)
- `ILP32` (`-q32`)
- `LP64` (`-q64`)
- `OBJECTMODEL` (`-qobjmodel`)
- `PHASEID` (`-qphsinfo`)
- `ROSTRING` (`-qro`)
- `SSCOMM` (`-qcpluscmt`)
- `TEST` (`-qdebug=format=isd`)

xlc and xlc

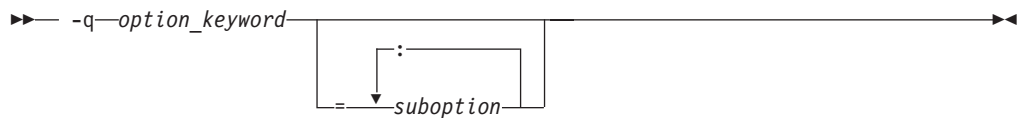
Options that do not exist on AIX, are not required to accomplish a z/OS specific task, and their effect can be accomplished by other means, are not supported with -q syntax (use the syntax in the list below instead). Options that fall in this category are:

- DEFINE (-D)
- OBJECT (-co)
- UNDEFINE (-U)

Suboptions with negative forms of -q options are not supported, unless they cause an active compiler action, as in the case of -qnokeyword=<keyword>.

Compiler options for AIX that do not apply to z/OS are accepted and ignored with a diagnostic message. For a brief description of the compiler options that can be specified with xlc, type xlc. For detailed descriptions of the compiler options that can be specified with xlc, refer to Chapter 4, "Compiler Options," on page 43.

The following syntax diagram shows how to specify keyword options using -q syntax:



In the diagram, *option_keyword* is an option name and the optional *suboption* is a value associated with the option. Keyword options with no suboptions represent switches that may be either on or off. The *option_keyword* by itself turns the switch on, and the *option_keyword* preceded by the letters NO turns the switch off. For example, -qLIST tells the compiler to produce a listing and -qNOLIST tells the compiler not to produce a listing. If an option that represents a switch is set more than once, the compiler uses the last setting.

Some keyword options only have values. Keywords which have values are specified as keyword=value pairs.

Example:

```
-qfloat=ieee
```

where *ieee* is a value.

Some keyword options have suboptions, which in turn have values. Suboptions which have values are specified as suboption=value pairs.

Example:

```
-qipa=level=2
```

where *level* is a suboption and 2 is a value.

Keyword options and suboptions may appear in mixed case letters in the command that invokes the xlc utility. Keyword options that have suboptions can also be preceded by the letters NO in which case they are similar to off switches described above and do not allow suboptions. This is a noticeable departure from the z/OS options, which allow suboptions even if they are preceded by the letters NO. However, the function that the z/OS behavior provides can easily be emulated by

specifying all desired suboptions with an option_keyword followed by the same option_keyword that is preceded by the letters NO. The subsequent specification of the same option_keyword unlocks all previously specified suboptions.

Example: NODEBUG(FORMAT(DWARF)) is equivalent to -qdebug=format=dwarf -qnodebug

The compiler recognizes all AIX -q options, but only those that have a matching z/OS native option are accepted and processed. All other AIX -q options are ignored with an informational message.

Flag options syntax

Except for the -W, -D, and -U flag options, all flag options that are supported by the c89 utility are supported by the xlc utility with the same semantics as documented in Chapter 18, “c89 — Compiler invocation using host environment variables,” on page 471. The xlc utility does not recognize constructs such as -Wl,I or -Wl,p. All other aspects of the -W flag are the same as with the c89 utility. -D and -U flag options are not preprocessed by the xlc utility. Instead, they are converted to the DEFINE and UNDEFINE native options and are passed to the compiler. The xlc utility also supports several new flag options, which are described below:

-# Displays language processing commands but does not invoke them; output goes to stdout.

►► -# ◄◄

-B Determines substitute path names for programs such as the assembler and binder, where program can be:

- a (assembler)
- c (C/C++ compiler)
- l (binder)
- L (IPA Link)

►► -B [prefix] [-t-program] ◄◄

Notes:

1. The optional prefix defines part of a path name to the new programs. The compiler does not add a / between the prefix and the program name.
2. To form the complete path name for each program, the xlc utility adds prefix to the program names indicated by the -t option. The program names can be any combination of C/C++ compiler, assembler, IPA Link and binder.
3. If -Bprefix is not specified, or if -B is specified without the prefix, the default path (/usr/lpp/cbclib/xlc/bin/) is used.
4. -tprograms specifies the programs for which the path name indicated by the -B option is to be applied.
5. -Bprefix and -tprograms options override the path names of the programs that are specified inside the configuration file indicated by the -Fconfig_file option.

xlc and xlc

Example: To compile myprogram.c using a substitute compiler and binder from /lib/tmp/mine/, enter:

```
xlc myprogram.c -B/lib/tmp/mine/
```

Example: To compile myprogram.c using a substitute binder from /lib/tmp/mine/, enter:

```
xlc myprogram.c -B/lib/tmp/mine/ -t1
```

-F Names an alternative configuration file (.cfg) for the xlc utility.

Suboptions are:

- config_file (specifies the name of an xlc configuration file.)
- stanza (specifies the name of the command used to invoke the compiler. This directs the compiler to use the entries under stanza in the config_file to set up the compiler environment.)

►► -F config_file :—stanza

Notes:

1. The default configuration file supplied at installation time is called /usr/lpp/cbclib/xlc/etc/xlc.cfg. Any file names or stanzas that you specify on the command line override the defaults specified in the /usr/lpp/cbclib/xlc/etc/xlc.cfg configuration file.
2. The -B, -t, and -W options override entries in the configuration file indicated by the -F option.

Example: To compile myprogram.c using a configuration file called /usr/tmp/mycbc.cfg, enter:

```
xlc myprogram.c -F/usr/tmp/mycbc.cfg
```

-O Optimizes generated code.

►► -O

-O2 Same as -O.

►► -O2

-O3 Performs some memory and compile time intensive optimizations in addition to those executed with -O2. The -O3 specific optimizations have the potential to alter the semantics of a user's program. The compiler guards against these optimizations at -O2 and the option -qstrict is provided at -O3 to turn off these aggressive optimizations.

►► -O3

-O4 Equivalent to -O3 -qipa.

►► -O4

-O5 Equivalent to -O3 -qipa=level=2.

►► -05 ◄◄

-P Produces preprocessed output in a file that has a suffix that is defined by `isuffix`, `isuffix_host`, `ixxsuffix`, and `ixxsuffix_host`. The default for host files is `.CEX` and for HFS files is `.i`.

As with the `-E` option, the `-C` option can be combined with the `-P` option to preserve the comments.

-t Adds the prefix specified by the `-B` option to the designated programs, where programs are:

- a (assembler)
- c (C/C++ compiler)
- L (Interprocedural Analysis tool - link phase)
- l (binder)



Note: This option must be used together with the `-B` option.

If `-B` is specified but the prefix is not, the default prefix is `/usr/lpp/cbclib/xlc/bin/`. If `-Bprefix` is not specified at all, the prefix of the standard program names is `/usr/lib/cbclib/xlc/bin/`.

If `-B` is specified but `-tprograms` is not, the default is to construct path names for all of the standard program names: `a`, `c`, `L`, and `l`.

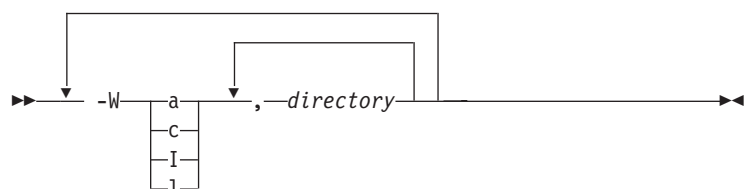
Example: To compile `myprogram.c` so that the name `/u/new/compilers/` is prefixed to the binder and assembler program names, enter:

```
xlc myprogram.c -B/u/new/compilers/ -tla
```

-W Passes the listed options to a designated compiler program where programs are:

- a (assembler)
- c (C/C++ compiler)
- I (Interprocedural Analysis tool - compile phase)
- l (binder)

Note: When used in the configuration file, the `-W` option requires the escape sequence back slash comma (`\,`) to represent a comma in the parameter string.



Example: To compile myprogram.s so that the option map is passed to the binder and the option list is passed to the assembler, enter:

```
xlc myprogram.s -Wl,map -Wa,list
```

Example: In a configuration file, use the \, sequence to represent the comma (,):

```
-Wl\,map,-Wa\,list
```

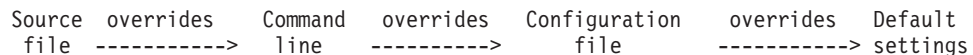
Specifying compiler options

Compiler options perform a wide variety of functions, such as setting compiler characteristics, describing the object code and compiler output to be produced, and performing some preprocessor functions. You can specify compiler options in one or more of the following ways:

- On the command line
- In your source program
- In a configuration file

The compiler uses default settings for the compiler options not explicitly set by you in the ways listed above. The defaults can be compiler defaults, installation defaults, or the defaults set by the c89 or the xlc utility. The compiler defaults are overridden by installation defaults, which are overridden by the defaults set by the c89 or the xlc utilities.

When specifying compiler options, it is possible for option conflicts and incompatibilities to occur. z/OS C/C++ resolves most of these conflicts and incompatibilities in a consistent fashion, as follows:



Options that do not follow this scheme are summarized in the following table:

Table 37. Compiler option conflict resolution

Option	Conflicting Options	Resolution
-qxref	-qxref=FULL	-qxref=FULL
-qattr	-qattr=FULL	-qattr=FULL
-E	-o	-E
-#	-v	-#
-F	-B -t -W -qpath <i>configuration file settings</i>	-B -t -W -qpath
-qpath	-B -t	-qpath overrides -B and -t

In general, if more than one variation of the same option is specified (with the exception of xref and attr), the compiler uses the setting of the last one specified. Compiler options specified on the command line must appear in the order you want the compiler to process them.

If a command-line flag is valid for more than one compiler program (for example -B, -W, or -I applied to the compiler, binder, and assembler program names), you must

specify it in options, or asopt in the configuration file. The command-line flags must appear in the order that they are to be directed to the appropriate compiler program.

Three exceptions to the rules of conflicting options are the *-Idirectory* or *-I//dataset_name*, *-llibrary*, and *-Ldirectory* options, which have cumulative effects when they are specified more than once.

Specifying compiler options on the command line

There are two kinds of command-line options:

- *-qoption_keyword* (compiler-specific)
- Flag options (available to z/OS C/C++ compilers in z/OS UNIX System Service environment)

Command-line options in the *-qoption_keyword* format are similar to on and off switches. For most *-q* options, if a given option is specified more than once, the last appearance of that option on the command line is the one recognized by the compiler. For example, *-qsource* turns on the source option to produce a compiler listing, and *-qnosource* turns off the source option so that no source listing is produced.

Example: The following example would produce a source listing for both *MyNewProg.C* and *MyFirstProg.C* because the last source option specified (*-qsource*) takes precedence:

```
xlc -qnosource MyFirstProg.C -qsource MyNewProg.C
```

You can have multiple *-qoption_keyword* instances in the same command line, but they must be separated by blanks. Option keywords can appear in mixed case, but you must specify the *-q* in lowercase.

Example: You can specify any *-qoption_keyword* before or after the file name:

```
xlc -qLIST -qnomaf file.c
xlc file.c -qxref -qsource
```

Some options have suboptions. You specify these with an equal sign following the *-qoption*. If the option permits more than one suboption, a colon (:) must separate each suboption from the next.

Example: The following example compiles the C source file *file.c* using the option *-qipa* to specify the inter procedural analysis options. The suboption *level=2* tells the compiler to use the full inter procedural data flow and alias analysis, *map* tells the compiler to produce a report, and the *noobj* tells the compiler to produce only an IPA object without a regular object. The option *-qattr* with suboption *full* will produce an attribute listing of all identifiers in the program.

```
xlc -qipa=level=2:map:noobj -qattr=full file.c
```

Specifying flag options

The z/OS C/C++ compilers use a number of common conventional flag options. Lowercase flags are different from their corresponding uppercase flags. For example, *-c* and *-C* are two different compiler options:

- *-c* specifies that the compiler should only preprocess, compile, and not invoke the binder
- *-C* can be used with *-E* or *-P* to specify that user comments should be preserved

xlc and x1C

Some flag options have arguments that form part of the flag.

Example:

```
x1C stem.c -F/home/tools/test3/new.cfg:myc -qflag=w
```

where new.cfg is a custom configuration file.

You can specify flags that do not take arguments in one string.

Example:

```
x1c -0cv file.c
```

has the same effect as:

```
x1c -0 -v -c test.c
```

Specifying compiler options in a configuration file

The default configuration file, (/usr/lpp/cbclib/xlc/etc/xlc.cfg), specifies information that the compiler uses when you invoke it. This file defines values used by the compiler to compile C or C++ programs. You can make entries to this file to support specific compilation requirements or to support other C or C++ compilation environments.

Options specified in the configuration file override the default settings of the option. Similarly, options specified in the configuration file are in turn overridden by options set in the source file and on the command line.

Specifying compiler options in your program source files

You can specify compiler options within your program source by using #pragma directives. Options specified with pragma directives in program source files override all other option settings.

Specifying compiler options for architecture-specific 32-bit or 64-bit compilation

You can use z/OS C/C++ compiler options to optimize compiler output for use on specific processor architectures. You can also instruct the compiler to compile in either 32-bit or 64-bit mode.

The compiler evaluates compiler options in the following order, with the last allowable one found determining the compiler mode:

1. Compiler default (32-bit mode)
2. Configuration file settings
3. Command line compiler options (-q32, -q64, -qarch, -qtune)
4. Source file statements (#pragma options(ARCH(suboption),TUNE(suboption)))

The compilation mode actually used by the compiler depends on a combination of the settings of the -q32, -q64, -qarch, and -qtune compiler options, subject to the following conditions:

- Compiler mode is set according to the last-found instance of the -q32, or -q64 compiler options. If neither of these compiler options is chosen, the compiler mode is set to 32-bit.

- Architecture target is set according to the last-found instance of the `-qarch` compiler option, provided that the specified `-qarch` setting is compatible with the compiler mode setting. If the `-qarch` option is not set, the compiler assumes a `-qarch` setting of 5.
- Tuning of the architecture target is set according to the last-found instance of the `-qtune` compiler option, provided that the `-qtune` setting is compatible with the architecture target and compiler mode settings. If the `-qtune` option is not set, the compiler assumes a default `-qtune` setting according to the `-qarch` setting in use.

Possible option conflicts and compiler resolution of these conflicts are described below:

- `-q32` or `-q64` setting is incompatible with user-selected `-qarch` option.
Resolution: `-q32` or `-q64` setting overrides `-qarch` option; compiler issues a warning message, sets `-qarch` to 5, and sets the `-qtune` option to the `-qarch` setting's default `-qtune` value.
- `-q32` or `-q64` setting is incompatible with user-selected `-qtune` option.
Resolution: `-q32` or `-q64` setting overrides `-qtune` option; compiler issues a warning message, and sets `-qtune` to the `-qarch` settings's default `-qtune` value.
- `-qarch` option is incompatible with user-selected `-qtune` option.
Resolution: Compiler issues a warning message, and sets `-qtune` to the `-qarch` setting's default `-qtune` value.
- Selected `-qarch` and `-qtune` options are not known to the compiler.
Resolution: Compiler issues a warning message, sets `-qarch` to 5, and sets `-qtune` to the `-qarch` setting's default `-qtune` setting. The compiler mode (32 or 64-bit) is determined by the `-q32/-q64` compiler settings.

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 9, “Binding z/OS C/C++ programs,” on page 355 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 540. For information on creating object libraries in z/OS C++, refer to Chapter 12, “Object Library Utility,” on page 421. 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 566.

Note: When you use the prelinker to prelink C++ object modules, you may get duplicate symbol warnings due to virtual function symbols generated by the compiler. You can ignore these symbols and warnings. You will not get these messages if you use the binder.

Restrictions on using the prelinker

You cannot use the prelinker if you specified either the XPLINK or GOFF compiler option when you compiled.

Note: The prelinker cannot be used for 64-bit compiled object modules, therefore you cannot use the prelinker if any of the object modules were compiled using the LP64 option.

Prelinking an application

To prelink multiple object modules and then link with a load module, you must run the multiple object modules through the prelinker and add the load module in the link step (for example, when prelinking and linking a CICS program).

You must prelink together all components that require prelinking prior to linking. For example, LINK(PRELINK(XOBJ1,XOBJ2)) and LINK(PRELINK(XOBJ1,XOBJ2),OBJ3) are valid but LINK(PRELINK(XOBJ1), PRELINK(XOBJ2)) is not. The prelinker only handles a subset of what the linker handles, in particular, it does not understand load modules (or program objects).

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 long names to resolve exported and imported symbols. For information on how to create a DLL or an application that uses DLLs, see *z/OS 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 `SYSMSGGS`.

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 38. For details on the attributes of specific data sets see “Description of data sets used” on page 595.

Table 38. Data sets used for prelinking

ddname	Type	Function
<code>SYSIN</code>	Input	Primary input data, usually the output of the compiler
<code>SYSMSGGS</code>	Input	Location of prelinker message file
<code>STEPLIB</code> ²	Utility Library	Location of prelinker and z/OS Language Environment run-time data sets
<code>SYSLIB</code>	Library	Secondary input
<code>SYSDEFSD</code> ¹	Output	Definition side-deck
<code>SYSOUT</code>	Output	Prelinker Map
<code>SYSMOD</code>	Output	Output data set for the prelinked object module

Table 38. Data sets used for prelinking (continued)

ddname	Type	Function
SYSPRINT	Output	Destination of error messages generated by the prelinker
User-specified	Input	Obtain additional object modules and load modules
Notes:		
¹	Required output from the prelinker if you are exporting variables or functions.	
²	Optional data sets, if the compiler and run-time library are installed in the LPA or ELPA. To save resources and improve compile time, especially in z/OS UNIX System Services, do not unnecessarily specify data sets on the STEPLIB DD name.	

Primary input (SYSIN)

Primary input to the prelinker consists of a sequential data set, a member of a partitioned data set, or an 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 569.)

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 (SYSMSG)

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 and SCEERUN2 must be in the JOBLIB or STEPLIB. For information on the search order, see Chapter 11, “Running a C or C++ application,” on page 409.

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 302.

Note: SYSLIB data sets that are used as input to the prelinker must be cataloged.

Definition side-deck (SYSDEFSD)

The prelinker generates a definition side-deck if you are prelinking an application that exports external symbols for functions and variables (a DLL). You must provide this side-deck to any user of your DLL. The users of the DLL must prelink the side-deck of the DLL with their other object modules. The definition side-deck (SYSDEFSD) is not generated if the NODYNAM option is in effect.

Listing (SYSOUT)

If you specify the MAP prelinker option, the prelinker writes a map to the SYSOUT data set. This map provides you with warnings, files that are included in input to the prelinker, and names of external symbols.

Output (SYSMOD)

The prelinker produces a single prelinked object module, and stores it in the SYSMOD data set. The linkage editor uses this data set as input.

Prelinker error messages (SYSPRINT)

If the prelinker encounters problems in its attempt to prelink your program, it generates error messages and places them in the SYSPRINT data set.

Input to the prelinker

Input to the prelinker can be:

- One or more object modules (not previously prelinked)
- Prelinker control statements (INCLUDE, LIBRARY ...)
- Object module libraries

The process of resolving or including input from these sources depends on the type of the source and the current input and prelink options.

Unresolved references or undefined writable static objects often result if you give the prelinker input object modules produced with a mixture of inconsistent compiler options (for example, RENT | NORENT, LONGNAME | NOLONGNAME, or DLL options). These options may expose symbol names in different ways in your object file, so that the prelinker may be unable to find the matching definition of a referenced symbol if the definition and the reference are exposed differently.

Primary input

Primary input to the prelinker consists of a sequential data set (file) that contains one or more separately compiled object modules, possibly with prelinker control statements. Specify the primary input data set through the SYSIN ddname.

Secondary input

Secondary input to the prelinker consists of object modules that are not part of the primary input data set but are to be included as a result of processing of primary input. Object modules that are brought in because of INCLUDE control statements are secondary input. Object modules brought in as a result of automatic call library (library search) processing of currently unresolved symbols through a LIBRARY control statement or through SYSLIB are also secondary input.

An automatic call library may be in the form of:

- PDS Libraries that contain object modules
- PDSE Libraries that contain object modules
- Archive Libraries that contain object modules (if you used OMVS prelinker option)

Prelinker output

Writable static references that are not resolved by the prelinker cannot be resolved later. Only the prelinker can be used to resolve writable static. The output object module of the prelinker should not be used as input to another prelink.

Prelinker Map

When you use the MAP prelinker option, the 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 571.
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 *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 39 shows the five data set names and their characteristics.

Table 39. 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 be introduced. (See “INCLUDE control statement” on page 569.)

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 OBJECT compiler option)
- Linkage editor control statements (NAME and ALIAS) that are generated by the ALIAS compiler option
- Previously link-edited load modules that you want to combine into one load module
- 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 595.

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 40.

Table 40. 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

Table 40. Options for controlling link-edit output (continued)

To Get This Output	Use This Option
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 SYSLOAD 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 *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 *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.

Example: The following is an example of a definition side-deck generated by the prelinker when prelinking a C object module:

```
IMPORT CODE 'BASICIO'   bopen
IMPORT DATA 'BASICIO'  bclose
IMPORT DATA 'BASICIO'  bread
IMPORT DATA 'BASICIO'  bwrite
IMPORT DATA 'BASICIO'  berror
```

You can edit the definition side-deck to remove any functions or variables that you do not want to export. For instance, in 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.

Example: The following is an example of a definition side-deck generated by the prelinker when prelinking a C++ object module:

```
IMPORT CODE 'TRIANGLE' getarea__8triangleFv
IMPORT CODE 'TRIANGLE' getperim__8triangleFv
IMPORT CODE 'TRIANGLE' __ct__8triangleFv
```

You can edit the definition side-deck to remove any functions and variables that you do not want to export. For instance, in 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 13, “Filter Utility,” on page 433.

Note: You should also provide users of your DLL with a header file that contains the prototypes for exported functions and extern variable declarations for exported variables.

The prelinker `NODYNAM` option must not be in effect when building DLLs.

Linking your code

When you link your code, ensure that you specify the `RENT` or `REUS(SERIAL)` options.

Using DLLs

The prelinker is used to build DLLs that export defined external functions and variables, and to build programs or DLLs that import external functions and variables from other DLLs.

Note: The prelinker `NODYNAM` option must not be in effect when using or building DLLs.

To assign a name to a DLL, use either the `DLLNAME()` prelinker option, or the `NAME` control statement. If you do not assign a name, and the data set `SYSMOD` is a PDS member, the member name is used as the DLL name. Otherwise, the name `TEMPNAME` is used.

To build a DLL, you need to compile object code that exports external functions or variables, then prelink and link that code into a load module. During the prelink step you need to capture the definition side-deck which is written to the ddname `SYSDEFSD`. The definition side-deck is a list of `IMPORT` control statements that correspond to the external functions and variables exported by the DLL.

Include the `IMPORT` statements at prelink time for any program that imports variables or functions from the DLL.

Example: In the following C example, `EXPONLY` is a DLL which only exports a single variable `year`:

```
/* EXPONLY.C */
int year = 2001;      /* exported from this DLL */
```

Example: In the following example, `IMPEXP` is a DLL that both imports and exports external functions and variables. It imports the external variable `year` from DLL `EXPONLY`, and exports external functions `next_year` and `get_year`.

```

/* IMPEXP.C */
extern int year;          /* imported from DLL EXPONLY */

void next_year(void) { /* exported from this DLL */
    ++year;           /* load DLL EXPONLY, modify 'year' in DLL */
}

int get_year(void) { /* exported from this DLL */
    return year;     /* get value of 'year' from DLL EXPONLY */
}

```

Example: In the following example, IMPONLY is a program that only imports functions and variables. It imports the variable `year` from DLL EXPONLY, and it imports functions `next_year` and `get_year` from DLL IMPEXP.

```

/* IMPONLY.C */
#include <stdio.h>
extern int get_year(void); /* import from DLL IMPEXP */
extern void next_year(void); /* import from DLL IMPEXP */
extern int year;          /* import from DLL EXPONLY */
int main(void)
{
    int y;
    next_year();          /* load DLL IMPEXP, call function from DLL */
    y = get_year();       /* call function in DLL IMPEXP */
    if ( y == 2002
        && year == 2002) /* get value of 'year' from DLL EXPONLY */
        printf("pass\n");
    else
        printf("fail\n");
    return 0;
}

```

Example: The following JCL builds the DLLs EXPONLY, IMPEXP, and the program IMPONLY, and then runs IMPONLY:

```

/* -----
//CEXPONLY EXEC EDCC,
// INFILE='USERID.DLL.C(EXPONLY) ',
// OUTFILE='USERID.DLL.OBJECT(EXPONLY),DISP=SHR ',
// CPARM='LONG RENT EXPORTALL'
/* -----
//CIMPEXP EXEC EDCC,
// INFILE='USERID.DLL.C(IMPEXP) ',
// OUTFILE='USERID.DLL.OBJECT(IMPEXP),DISP=SHR ',
// CPARM='LONG RENT DLL EXPORTALL'
/* -----
//CIMPONLY EXEC EDCC,
// INFILE='USERID.DLL.C(IMPONLY) ',
// OUTFILE='USERID.DLL.OBJECT(IMPONLY),DISP=SHR ',
// CPARM='LONG RENT DLL'
/* -----
//LINK1 EXEC CBCL,PPARM='DLLNAME(EXPONLY) ',
// OUTFILE='USERID.DLL.LOAD(EXPONLY),DISP=SHR '
//PLKED.SYSIN DD DSN=USERID.DLL.OBJECT(EXPONLY),DISP=SHR
//PLKED.SYSDEFSD DD DSN=USERID.DLL.IMPORTS(EXPONLY),DISP=SHR
/* -----
//LINK2 EXEC CBCL,PPARM='DLLNAME(IMPEXP) ',
// OUTFILE='USERID.DLL.LOAD(IMPEXP),DISP=SHR '
//PLKED.SYSIN DD DSN=USERID.DLL.OBJECT(IMPEXP),DISP=SHR
// DD DSN=USERID.DLL.IMPORTS(EXPONLY),DISP=SHR
//PLKED.SYSDEFSD DD DSN=USERID.DLL.IMPORTS(IMPEXP),DISP=SHR

```

Figure 45. JCL to build DLLs (Part 1 of 2)

```

/* -----
//LINK3 EXEC CBCL,
// OUTFILE='USERID.DLL.LOAD(IMPONLY),DISP=SHR '
//PLKED.SYSIN DD DSN=USERID.DLL.OBJECT(IMPONLY),DISP=SHR
// DD DSN=USERID.DLL.IMPORTS(EXPONLY),DISP=SHR
// DD DSN=USERID.DLL.IMPORTS(IMPEXP),DISP=SHR
/* -----
//GO EXEC PGM=IMPONLY
//STEPLIB DD DSN=USERID.DLL.LOAD,DISP=SHR
// DD DSN=CEE.SCEERUN,DISP=SHR
// DD DSN=CEE.SCEERUN2,DISP=SHR
//SYSOUT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*

```

Figure 45. 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 G0 has the program IMPONLY and the DLLs. EXPONLY and IMPEXP in its STEPLIB concatenation so that the DLLs can be dynamically loaded at run time.

To see which functions and variables are imported or exported use the Prelinker Map. The following is a portion of the Prelinker Map from step LINK2:

```

=====
|                               Load Module Map 1                               |
=====

MODULE ID  MODULE NAME

    00001   EXPONLY

=====

|                               Import Symbol Map 2                               |
=====

*TYPE      FILE ID  MODULE ID  NAME
    D       00001   00001     year

*TYPE:  D=imported data  C=imported code

=====

|                               Export Symbol Map 3                               |
=====

*TYPE      FILE ID  NAME
    C       00001   get_year
    C       00001   next_year

*TYPE:  D=exported data  C=exported code

```

1 Load Module Map

This section lists the load modules from which functions and variables are imported. The load module names come from the input IMPORT control statements processed.

2 Import Symbol Map

This section lists the imported functions and variables. The MODULE ID indicates the DLL from which the function or variable is imported. The FILE ID indicates the file in which the IMPORT control statement was processed that resulted in this import.

3 Export Symbol Map

This section lists the external functions and variables which are exported. For each symbol that is listed in this section, an IMPORT control statement is written out to the DDname SYSDEFSD, the definition side-deck.

Note: The export symbol map will not be produced when the NODYNAM option is in effect.

Prelinking and linking an application under z/OS batch and TSO

Figure 46 shows the basic prelinking and linking process for your C or C++ application.

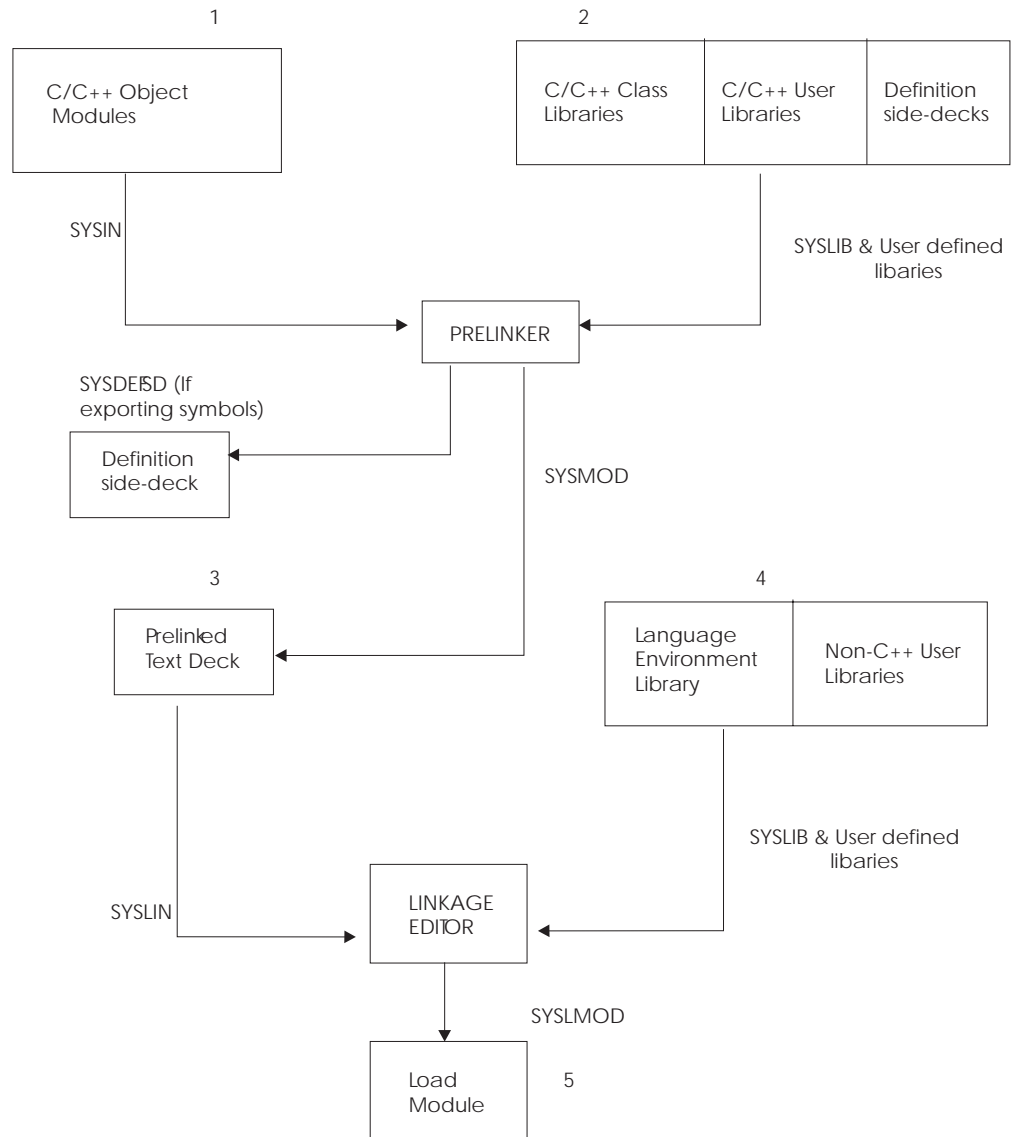


Figure 46. Basic prelinker and linkage editor Processing

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 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.

Example: Consider the following example. The data set USERID.DLL.SOURCE(EXPONLY) contains

```
/* EXPONLY.C */
   int year = 2001; /* exported from this DLL */
```

After step LINK0 in Figure 48 on page 551, the definition side-deck USERID.DLL.IMPORTS(EXPONLY) contains the record `IMPORT DATA 'EXPONLY' year.`

The map that is shown in Figure 49 on page 551 was created by compiling the program that is shown in Figure 47 on page 551. Figure 49 on page 551 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 47. 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 48. 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  DYNAM

=====
|                               Object Resolution Warnings 2                               |
=====

WARNING EDC4015: Unresolved references are detected:
CEESTART CEESG003 @@TRGLOR

```

Figure 49. 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 49. 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 49. 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.

Note: The export symbol map will not be produced if the `NODYNAM` option is in effect.

8 ESD Map of Defined and Long Names

This section lists the names of external symbols that are not in writable static. It also shows a mapping of input long names to output short names.

If the object is defined, the FILE ID indicates the file that contains the definition. Otherwise, this field is left blank. For any name, the input name and output short name are listed. If the input name is indeed an long name, the rule that is used to map the long name to the short name is applied. If the name is not an long name, this field is left blank.

Note: Although mangled names exist in the object modules, the Prelinker Map and messages emit the demangled equivalent, which is like the names seen in the C++ source code.

Processing the prelinker automatic library call

The following hierarchy is used to resolve a referenced and currently undefined symbol.

- The undefined name is an short name, for example SNAME.
 - If the NONCAL command option is in effect, the partitioned data sets that are concatenated to SYSLIB are searched in order as follows:
 - If the data set contains a C370LIB-directory created using the z/OS 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 12, “Object Library Utility,” on page 421.

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 49 on page 551.

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 571 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: 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:

```
EDCCCL  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 EDCCCL, 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 591 for more information on REXX EXECs and their uses.

Example: The following example shows the general job control procedure for link-editing a program under z/OS batch using the Language Environment Library.

```
// jobcard
//*
/* THE FOLLOWING STEP LINKS THE MEMBERS TESTFILE AND DECODE FROM
/* THE LIBRARIES USERID.WORK.OBJECT AND USERID.LIBRARY.OBJECT AND
/* PLACES THE LOAD MODULE IN USERID.WORK.LOAD(TEST)
/*
//LKED  EXEC  PGM=IEWL,REGION=1024K,PARM='AMODE=31,RMODE=ANY,MAP'
//SYSLIB DD  DSNAME=CEE.SCEELKED,DISP=SHR
//SYSLIN DD  DDNAME=SYSIN
//SYSLMOD DD  DSNAME=USERID.WORK.LOAD(TEST),DISP=SHR
//OBJECT DD  DSNAME=USERID.WORK.OBJECT,DISP=SHR
//LIBRARY DD  DSNAME=USERID.LIBRARY.OBJECT,DISP=SHR
//SYSPRINT DD  SYSOUT=*
//SYSUT1 DD  UNIT=VIO,SPACE=(32000,(30,30))
//SYSIN  DD  DATA,DLM=@@
          INCLUDE  OBJECT(TESTFILE)
          INCLUDE  LIBRARY(DECODE)
@@
```

Figure 50. 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:

```
CBCCCL  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 591.

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 577, for linkage editor options see “Linkage editor options” on page 579.

Example of JCL to prelink and link

Figure 51 on page 558 shows a typical sequence of job control statements to link-edit an object module into a load module.

```

/*-----
/* PRE-LINKEDIT STEP:
/*-----
//PLKED EXEC PGM=EDCPRLK,REGION=2048K,PARM='MAP'
//STEPLIB DD DSN=CEE.SCEERUN,DISP=SHR
// DD DSN=CEE.SCEERUN2,DISP=SHR
//SYSMSG 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)
//SYSLOAD DD DSN=USERID.LOAD(PROG1),DISP=SHR
//SYSUT1 DD UNIT=VIO,SPACE=(32000,(30,30))
//SYSPRINT DD SYSOUT=*

```

Figure 51. 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 591 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 52 on page 559.


```

:
//SYSLIN DD      DSNAME=&&GOFILE,DISP=(SHR,DELETE)
//          DD      *
          INCLUDE ddname(member)
          LIBRARY ADDLIB(CPGM10)
/*

```

Figure 52. Linkage Editor control statements

As 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 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.

Example: Code these statements after your DD statements as part of your job control procedure; for example:

```

:
//SYSIN DD      DSNAME=&&GOFILE,DISP=(SHR,DELETE)
//          DD      *
          INCLUDE ddname(member)
          LIBRARY ADDLIB(CPGM10)
/*

```

Data sets specified by the INCLUDE statement are incorporated as the prelinker and linkage editor encounter the statement. In contrast, data sets specified by the LIBRARY statement are used only when there are unresolved references after all the other input is processed.

Any prelinker and linkage editor processing beyond the basic processing 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 601.

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.

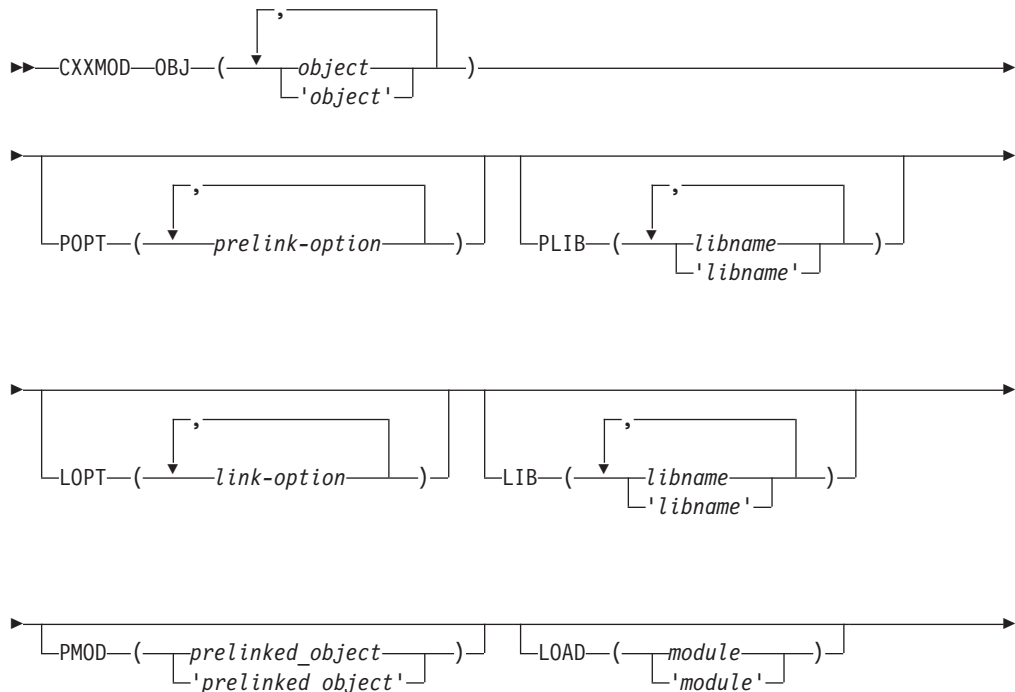
Any linkage editor processing beyond the basic processing described above must be specified by linkage editor control statements in the primary input. The IBM-supplied catalog procedures and REXX EXECs use the DLL versions of the IBM-supplied class libraries by default.

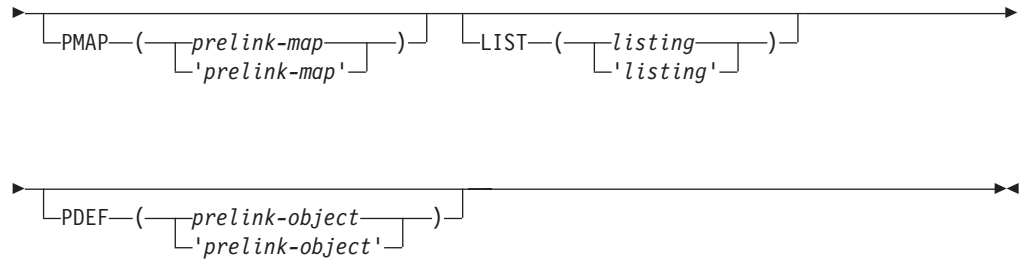
To link-edit your z/OS C program under TSO, use either the CXXMOD, CMOD, or the LINK command. It is recommended that you use CXXMOD, particularly when linking z/OS C and z/OS C++ object decks. For a description of the CXXMOD REXX EXEC see “Prelinking and linking under TSO.” For a description of CMOD and the TSO LINK command see “Other z/OS C utilities” on page 601.

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

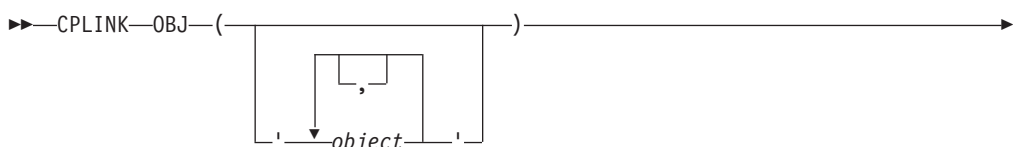
In the following example, the user prefix is **RYAN** and the input object module members **MAIN** and **FN** are in the PDS called **'RYAN.ACCOUNT.OBJ'**. A prelink map is to be generated and placed in **'RYAN.ACCOUNT.MAP(SALES)'**. The load module will be placed in a PDS member called **'GROUP.ACCOUNT.LOAD(SALES)'**. The linkage editor listing will be written to **'RYAN.ACCOUNT.LIST(SALES)'**.

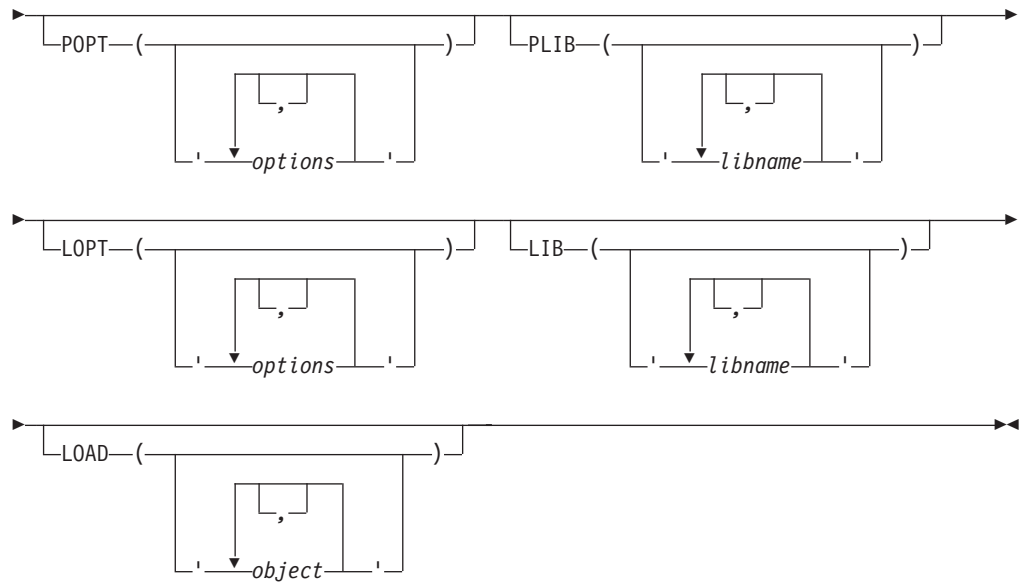
```
CXXMOD OBJ(ACCOUNT.OBJ(MAIN), ACCOUNT.OBJ(FN))
      POPT(MAP) LOPT(XREF, MAP)
      LOAD('GROUP.ACCOUNT.LOAD(SALES)') MAP(ACCOUNT.MAP(SALES))
      LIST(ACCOUNT.LIST(SALES))
```

In this instance, both the z/OS Language Environment stub library and the partitioned data set (library) **SAESLIB** 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) '
/*-----
```

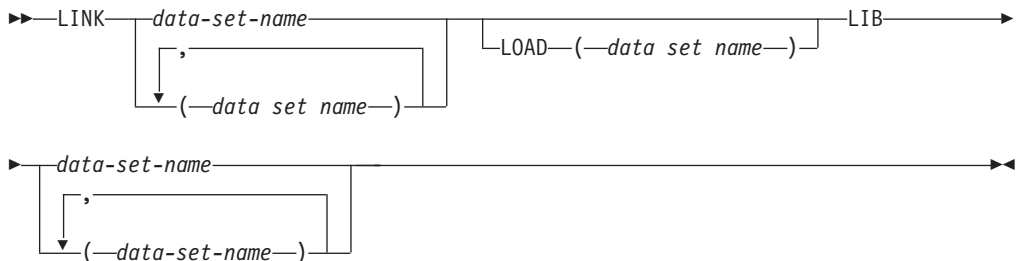
Figure 53. Example of prelinking under z/OS batch

```
CPLINK OBJ(''PAUL.C.OBJ(INCPYRL)''')
      POPT('NOMAP,NONCAL')
      PLIB(''RAINBOW.LIB.SUB''')
      LOAD('TBD.LOAD(MOD)')
```

Figure 54. 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.

Example: The following example shows how to link-edit two object modules and place the resulting load module in *member* TEMPNAME of the *userid.LM.LOAD* data set.

```
LINK program1,program2 LOAD(lm)
```

You can also specify link-edit options in the link statement:

```
LINK program1 LOAD(lm) LET
```

Options for the linkage editor are discussed in "Output from the linkage editor" on page 542.

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 577.

Using your JCL

The example JCL in Figure 55 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,ONCAL'
//STEPLIB DD DISP=SHR,DSN=CEE.SCEERUN
//          DD DISP=SHR,DSN=CEE.SCEERUN2
//SYMSGS 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 55. Using OMVS to prelink and link

The JCL in Figure 55 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 CEER00TA goo      CEESG003 EDCINPL

=====
|                               File Map                                         |
=====

*ORIGIN  FILE ID  FILE NAME

      PI   00001  /u/myusrd/callfoogoo.hoo.o
      PI   00002  MYUSRID.QAPARTNR.OBJ(MEM1)
      A    00003  /u/myusrd/mylibrary.a(foo.o)
      A    00004  MYUSRID.QAPARTNR.OBJ(MEMH00)

*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                   |
=====

      *REASON  FILE ID  OUTPUT
                        ESD NAME  INPUT NAME

      P        00001  CEESTART  CEESTART
      P        00001  CEEMAIN  CEEMAIN
      D        00001  MAIN     main
      D        00003  FOO      foo
      D        00003  GOO      goo
      D        00004  HOO      hoo
      P        00003  CEESG003 CEESG003
      P        00003  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 56. 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 Chapter 18, “c89 — Compiler invocation using host environment variables,” on page 471. 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 577.

Prelinker control statement processing

The only control statements that the prelinker processes are IMPORT, INCLUDE, LIBRARY, and RENAME statements. The remaining control statements remain unchanged until the link step.

You can place the control statements in the input stream, or store them in a permanent data set. If you cannot fit all of the information on one control statement, you can use one or more continuations. The long name, for example, can be split across more than one statement. You can enable continuations in one of two ways:

- Place a non-blank character in column 72 of the statement that is to be continued. The continuation must begin in column 16 of the next statement.
- Enclose the name in single quotation marks. When such a name is continued across statements, it extends up to and includes column 71. Although column 72 is not considered part of the name, it must be non-blank for the name to be

continued. On the following statement, column 1 must be blank (containing the X'40' character); the name then continues in column 2.

If you have a name that contains a single quotation mark, and you want to enclose the whole name in single quotation marks, put two single quotation marks next to each other where you want the single one to appear in the name.

Example: If you want the name

```
SymbolNameWithAQuote'InTheMiddle
```

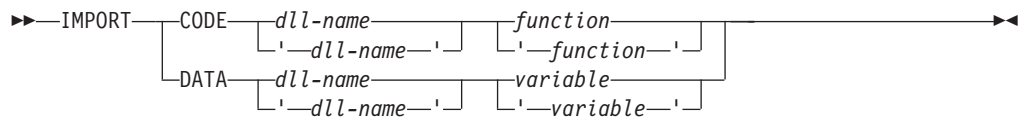
specify it as follows:

```
'SymbolNameWithAQuote'InTheMiddle'
```

If you mix the two style of continuation in one control statement, after you continue a statement in column 2 due to a 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 long name. The *dll-name* comes from the value specified on the DLLNAME prelinker option. For more information, see “Prelinker options” on page 577.

variable

An exported variable name. It is a mixed case long name. To indicate a continuation across statements, either use a non-blank character in column 72 of the card and begin the next line in column 16, or enclose the name in single quotation marks, end the first line in column 71, and put a blank character in column 1 of the next line.

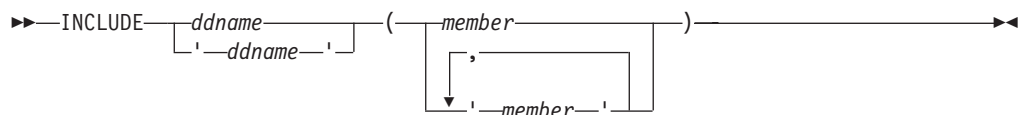
function

An exported function name. It is a mixed case long name. You can indicate a continuation the same way you would for a variable.

The prelinker processes IMPORT statements, but does not pass them on to the link step.

INCLUDE control statement

The INCLUDE control statement has the following syntax:



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.

short names and long names can be specified, case distinction is significant. If you use an long name, you can use the same kinds of continuations that you can for the *variable* on the `IMPORT` control statement.

Note: The `LIBRARY` control statement is removed and not placed in the prelinker output object module; the system linkage editor does not see the `LIBRARY` control statement.

RENAME control statement

The `RENAME` control statement has the following syntax:

NOOMVS

```
►►—RENAME— $\left[ \begin{array}{l} \textit{long name} \\ \text{'—long name—'} \end{array} \right]$ — $\textit{short name}$ — $\left[ \text{SEARCH} \right]$ —►►
```

OMVS

```
►►—RENAME— $\left[ \begin{array}{l} \textit{long name} \\ \text{'—long name—'} \end{array} \right]$ — $\textit{short name}$ —►►
```

long name

the name of the long name to be renamed on output. All occurrences of this long name are renamed. You can use the same kinds of continuations that you can for the *variable* on the `IMPORT` control statement.

short name

the name of the short name to which the long name will be changed. This name can be at most 8 characters, and case is respected.

SEARCH

an optional parameter specifying that if the short name is undefined, the prelinker searches by an automatic library call for the definition of the short name. This is not available with the `OMVS` option.

The `RENAME` control statement is processed by the prelinker. You can use this statement to do the following:

- Explicitly override the default name that is given to an long name when an long name is mapped to a short name.

You can explicitly control the names that are presented to the system linkage editor so that external variable and function names are consistent from one linkage editor run to the next. This consistency makes it easier to recognize control section and label names that appear in system dumps and linkage editor listings. Another mapping rule can provide the suitable name, but if you need to replace the linkage editor control section, you need to maintain consistent names. See “Mapping long names to short names” on page 539 for a description of this rule.

- Explicitly bind a long name to a short name. This binding may be necessary when linking with other languages that use a different name for the same object.

A `RENAME` control statement cannot be used to rename a writable static object because its name is not contained in the output from the prelinker.

You can place RENAME control statements before, between, or after other control statements or object modules. An object module can contain only RENAME statements. RENAME statements can also be placed in input that is included because of other RENAME statements.

Usage notes

- A RENAME statement is ignored if the long name is not encountered in the input.
- A RENAME statement for an long name is valid provided **all** of the following are true:
 - The long name was not already mapped because of a rule that preceded the RENAME statement rule in the hierarchy described in “Mapping long names to short names” on page 539.
 - 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 *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 you are not using the binder, and your program contains writable static, you can use the prelinker to make your program reentrant. If the program is C and does not

contain writable static, you do not need to use the prelinker to ensure reentrancy; the program is naturally reentrant. C++ programs always contain writable static.

If you compile your code and wish to link it using the z/OS system link procedures such as IEWL, you must first call the prelinker.

The z/OS UNIX System Services features require that all z/OS 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.

Steps for generating a reentrant load module in C

Perform the following steps to generate a reentrant load module in C:

1. Determine whether or not your program contains writable static. If you are unsure about whether your program contains writable static, compile it with the RENT option. Invoking the prelinker with the MAP option and the object module as input produces a Prelinker Map. Any writable static data in the object module appears in the writable static section of the map. Unresolved writable static references may also appear in the map as errors.

If you see the symbol @STATIC defined in the writable static section, your code contains unnamed writable static such as modifiable literal strings, or variables with the static qualifier. To 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.

Steps for generating a reentrant load module in C++

Perform the following steps to generate a reentrant load module in C++:

1. Compile your source code.
If you see the symbol `@STATIC` defined in the writable static section, your code contains unnamed writable static such as modifiable literal strings, or variables with the static qualifier. To ensure that literal strings stay in the code area, recompile with `#pragma strings(readonly)`, and prelink again.

2. Use the supplied prelink and link utilities on the module. Under TSO, you can use the `CXXMOD REXX EXEC` to both prelink and link your module. Under z/OS batch, use these JCL procedures:
 - CBCCL: compile and link
 - CBCL: link
 - CBCCLG: compile, link, and go
 - CBCLG: link and goFor all of these, linking involves two steps: invocation of the prelinker, and then a call to the system linker.

Resolving multiple definitions of the same template function

Note: For complete information on using C++ templates, see *z/OS C/C++ Programming Guide*

When the prelinker generates template functions, it resolves multiple function definitions as follows:

- If a function has both a specialization and a generalization, the specialization takes precedence.
- If there is more than one specialization, the prelinker issues a warning message.

Because the link step does not remove unused instantiations from the executable program, instantiating the same functions in multiple compilation units may generate very large executable programs.

External variables

For more information on external variables, see *z/OS C/C++ Programming Guide*.

The POSIX 1003.1 and X/Open CAE Specification 4.2 (XPG4.2) require that the C system header files declare certain external (global) variables. Additional variables are defined for use with POSIX or XPG4.2 functions. If you define one of the POSIX or XPG4 feature test macros and include one of these headers, the global variables will be declared in your program. These global variables are treated differently than other global variables in a multi-threaded environment (values are thread-specific rather than global to the process) and across a call to a fetched module (values are propagated rather than module-specific). To access the global variables, you must use either C with the RENT compiler option, C++, or the XPLINK compiler option. If you are not using XPLINK, you must also specify the SCEEOBJ autocall library. The SCEEOBJ library must be specified before the SCEELKEX and the SCEELKED libraries in the bind step. If the SCEEOBJ library is specified after the SCEELKEX and SCEELKED libraries, the bind step will resolve the external variables to the user application, but 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 *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. If you use the XPLINK compiler option for a 32-bit program, the global variables are resolved by import using the CELHS003 member of the SCEELIB data set. The thread-specific values are always used.

For a dynamically called DLL module to share access to the POSIX external variables, with its caller, the DLL module must define the `_SHARE_EXT_VARS` feature test macro. For more information, see the section on feature test macros in the *z/OS 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 535.

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 569. If you specify the DLLNAME option, the prelinker sets the DLL name to the value that you listed on the option.

If you do not specify DLLNAME, the prelinker sets the DLL name to the name that appeared on the last `NAME` control statement that it processed. If there are no `NAME` control statements, and the output object module of the prelinker is a PDS member, it sets the DLL name to the name of that member. Otherwise, the prelinker sets the DLL name to the value `TEMPNAME`, and issues a warning.

DUP | NODUP

DEFAULT: DUP

DUP specifies that if duplicate symbols are detected, their names should be directed to the console, and the return code minimally set to a warning level of 4. NODUP does not affect the return code setting when the prelinker detects duplicates.

DYNAM | NODYNAM

DEFAULT: DYNAM

When the NODYNAM option is in effect, export symbol processing is not performed by the prelinker even when export symbols are present in the input objects. The side-deck is not created and the resulting module will not be a DLL. Specify NODYNAM for prelinked C/C++ programs involved in COBOL C/C++ ILC calls.

ER | NOER

DEFAULT: ER

Note: For the z/OS UNIX Systems Services environment, the default is NOER.

If there are unresolved symbols, ER instructs the prelinker to write a messages and a list of unresolved symbols to the console. If there are unresolved references, the prelinker sets the return code to a minimum warning level of 4. If there are unresolved writable static references, the prelinker sets the return code to a minimum error level of 8. If you use NOER, the prelinker does not write the list of unresolved symbols to the console. If there are unresolved references, the return code is not affected. If there are unresolved writable static references, prelinker sets the return code to a minimum warning level of 4.

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 550 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 `N00MVS`, the data set must be partitioned, but for `0MVS` 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.

0MVS | N00MVS

DEFAULT: N00MVS

The `0MVS` option causes the prelinker to change the way that it processes `INCLUDE` and `LIBRARY` control statements. The `c89` utility turns on the `0E` option (which maps to the `0MVS` 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 processes it again. For example, if you pass your object files and object libraries as follows:

```
c89 file1.o lib1.a file2.o lib2.a
```

The prelinker processes the INCLUDE control statement for file1.o, and incorporates new symbol definitions and unresolved references from the object file into the output file. The prelinker then processes the LIBRARY control statement for lib1.a, and searches the library for currently unresolved symbols. It then processes file2.o followed by lib2.a. If the processing of file2.o results in unresolved symbols, the prelinker will not search the library lib1.a again, because it has already processed it. If you have unresolved symbols that may be defined in a library that has already been processed, you must specify a new LIBRARY statement after your INCLUDE statement to resolve those symbols. You can do this on a c89 command line as follows:

```
c89 file1.o lib1.a file2.o lib1.a lib2.a
```

RENAME control statements are processed on output from the prelinker, after all of its input has been processed. Because a library can be processed once only, the SEARCH option on the RENAME control statement has no effect.

Note: The 0E 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 5, “Binder options and control statements,” on page 285 or the *z/OS DFSMS Program Management* manuals.

Appendix C. Diagnosing problems

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 <http://techsupport.services.ibm.com/guides/handbook.html> for further information on obtaining IBM service and support.

Problem checklist

The following list contains suggestions to help you rule out some common sources of problems.

- Check that the program has not changed since you last compiled or executed it successfully. If it has, examine the changes. If the error occurs in the changed code and you cannot correct it, note the change that caused the error. Whenever possible, you should retain copies of both the original and the changed source programs.
- Be sure to correct all problems that are diagnosed by error messages, and ensure that the messages that were previously generated have no correlation to the current problem. Be sure to pay attention to warning messages.
- The message prefix can identify the system or subsystem that issued the message. This can help you determine the cause of the problem. Following are some of the prefixes and their origins.
 - CCN - indicates messages from the z/OS C/C++ compiler, its utility components, or the z/OS C/C++ IPA Link step. Information on the messages is found in *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 *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 *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 *z/OS Language Environment Run-Time Messages*.
 - CLB for messages that relate to OS/390 V2R10 class libraries and CLE for messages that relate to z/OS class libraries. See *z/OS C/C++ Messages* for more information.
 - BPX - messages that relate to z/OS UNIX System Services.
 - FSUM - messages for the c89 and xlc utilities.

You can cross reference the prefix to the message 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.

- Ensure that you are compiling the correct version of the source code. It is possible that you have incorrectly indicated the location of your source file. For example, check your high-level qualifiers.

- In any program failure, keep a record of the conditions and options in effect at the time the problem occurred. The listing file shows the options. To get the listing, compile with the `SOURCE` option. The listing only contains options that appear after the command line is processed, hence `C #pragma` options do not appear.

Information about some of the options appears as a comment at the end of the object file. For both C or C++ compilers, there is always a comment showing the `OPTIMIZE` level. For C compilers, information about some of the options (for example, `ALIAS`, `GONUMBER`, `INLINE`, `RENT`, or `UPCONV` options) is included only if you specify the option when you compile. Note any changes from the previous compilation.

- Your installation may have received an IBM Program Temporary Fix (PTF) for the problem. Verify that you have received all issued PTFs and have installed them, so that your installation is at the current maintenance level. Specifying the compiler option `PHASEID` when doing a compile provides information about the maintenance level of each compiler component (phase).
- The preventive service planning (PSP) bucket, which is an online database available to IBM customers through IBM service channels. It gives information about product installation problems and other problems. See the *z/OS Program Directory* for more details.
- Use the Debug Tool, `dbx` (for *z/OS UNIX System Services*) or some other debugging aid to determine the statement where the program fails and possible causes of the failure.
- If a failing application is communicating with other IBM products, make sure that it uses the correct interface procedure as documented in *z/OS C/C++ Programming Guide*. In many cases, you can localize the failing condition by taking out the function calls or making them no-ops.
- If your application has been developed on a different platform (such as a microcomputer or workstation) and you try to compile and run using the *z/OS 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 IEEE long double data types may have different sizes on a different platform.
 - The source code is system dependent
- If your application was prelinked, make sure that the prelinking was successful as indicated in Appendix A, “Prelinking and linking *z/OS C/C++* programs,” on page 535.

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 the *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 recreates the problem. A test case helps you to isolate the problem and to report problems to IBM.

To create a small test case from a large program that appears to be failing, try the suggestions listed below, after you have either backed up or made a copy of your original source code. Begin with the suggestion that seems most appropriate for the problem that you are having. If the problem persists after you have tried one of the steps below, try another in the list. Continue to break your program down until you obtain the smallest possible segment of code that still reproduces the error. Compile with the `PPONLY` option and send the expanded file as your source code. This is 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.

Remove any code that has not been processed at the time of failure (except for code necessary to ensure the syntactic and semantic validity of the program).

Find unreferenced variables using the `IPA(XREF)` option, the `CHECKOUT(GEN)` option, which is for C only, or the `INFO(USE)` option, which is for C++ only, and remove the unreferenced variables.

Remove all code and declarations from the body of any other functions that are not necessary to reproduce the problem. The function should be removed if it is not necessary.

If your program uses structure variables, try replacing them with scalar variables.

Steps for problem diagnosis using optimization levels

Before you begin: For diagnostic purposes, you should always begin by using the simplest optimization level on your program. Once you address all problems at your current level, progress toward the more complex levels of optimization.

Perform the following steps to progress through the various levels of optimization:

1. Begin with a non-IPA compile and link using progressively higher levels of optimization:
 - `OPT(0)`
 - `OPT(2)`
 - `OPT(3)`

If your program works successfully at `OPT(0)` and fails at `OPT(2)`, try rebuilding the program specifying the compiler option `NOANSIALIAS` and re-running. You may suffer a performance penalty for this as the optimizer has to make worst-case aliasing assumptions but it may resolve the problem.

2. Use `IPA(OBJONLY)` and `OPT(2)`. This adds the IPA compile-time optimizations and often locates the problematic source file before you invest a lot of time and effort diagnosing problems in your code at IPA Link time.
3. Use the full IPA Compile and `IPA(Level(1))` Link path. IPA Compile-time optimizations are performed on the IPA object. IPA Link-time optimizations are performed on the entire application.

4. Use the full IPA Compile and IPA(Level(2)) Link path. IPA Level 2 performs additional link-time optimizations.
-

You know you are done when you have exploited all optimizations offered by the compiler.

Steps for diagnosing errors that occur at compile time

Perform the following steps to diagnose errors that occur at compile time:

1. If your program uses any of the library routines, insert an `#include` directive for the appropriate header files. Also insert an `#include` directive for any of your own header files. The compiler uses function prototypes, when present, to help detect type mismatches on function calls. You can use the C `CHECKOUT` option to find missing prototyping. Note that z/OS 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 *z/OS C/C++ Programming Guide* for more details on `#pragma filetag`.

-
7. Check for duplicate static constructors and destructors in your C++ source. Entries for constructors are created in the object and in a table. When a static constructor is removed, the entry in the object is removed, but the table entry stays. This will cause the static constructor and destructor to be called multiple times. If the destructor deletes (or frees) dynamically allocated storage that is associated with a pointer, it will tend to fail on subsequent invocations.
-
8. A compile-time abend can indicate an error in the compiler. An unsuccessful compilation due to an error in the source code or an error from the operating system should result in error messages, not an abend. However, the cause of the compiler failure may be a syntax error or an error from the operating system. Use the `PHASEID` compiler option to obtain the maintenance service level of the compiler, as well as the name of the failing compiler component, in the output listing.
-

If you still have a compilation problem, contact IBM support.

Steps for diagnosing errors that occur at IPA Link time

Perform the following steps to diagnose errors that occur at IPA Link time:

1. Ensure that the region that is used for the IPA Link step is sufficient. 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 an IPA object.
-
3. Ensure that all application program parts (object modules, load modules) and all necessary interface libraries (Language Environment object modules and load module, SQL, CICS, etc) are made available to the IPA Link step.
-
4. Ensure that the IPA Compile step has processed all object modules for which source is available.
-
5. Use the `IPA(LINK,MAP)` option to obtain an IPA Link listing.
-
6. Do not attempt to IPA Link unsupported file formats, such as Program Objects.
-
7. Verify that there are no unresolved symbol references. All user symbols must be resolved before invoking the binder (or prelinker and linkage editor). Any run-time symbol references generated by IPA Link must be resolved by the subsequent step to that no unresolved symbols remain.

If you have unresolved symbols, make sure that the definition of an object and all its references are used consistently in both the code area and the writable static area. Also, make sure that symbol references appear consistently in the same case.

If you have unresolved symbols after using autocall, and you are searching for longnamed or writable static objects, make sure that each object module library has a current directory generated by the C370LIB utility. Without this directory, autocall can only be done on the member name of the object module and not on what is actually defined within the member.

-
8. If problems occur during IPA Link processing of DLL code, note that a symbol can only be imported if all of the following conditions hold true:
- The symbol remains unresolved after autocall
 - Only DLL references were seen for the symbol
 - An IMPORT control statement was encountered for the symbol
-
9. A compiler ABEND during IPA Link step processing can indicate an error in the compiler. An unsuccessful IPA Link due to an error in the program source code, an invalid object module, an invalid load module, or an error from the operating system should result in error messages, not an ABEND.

If the compiler ABEND during IPA Link step processing is related to an invalid IPA object module, it will require further diagnosis:

- Save and recompile any IPA object modules created by a previous release 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.

You should now have a clean IPA Link compilation. If you still have a problem with the IPA Link step, contact IBM support.

The error occurs at bind time

For information on bind-time errors, see “Error recovery” on page 403.

The error occurs at prelink time

If the error occurs at prelink time:

- Do not prelink the object modules separately.
- Use the prelinker option MAP to obtain a full map of input data sets and symbols.
- Use the prelinker options DUP and ER to obtain a full list of duplicate and unresolved symbols.

- If you have unresolved symbols, make sure that the definition of an object and all references to that object are used consistently in both the code area and the writable static area. Also, make sure that symbol references appear consistently in the same case.
- A symbol can only be imported if all of the following conditions hold true:
 - The symbol remains unresolved after `autocall`.
 - Only DLL references were seen for the symbol.
 - An `IMPORT` control statement was encountered for the symbol.

For more information on using DLL, see “Using DLLs” on page 545, or the DLL description in *z/OS C/C++ Programming Guide*.

- If you have unresolved symbols after using `autocall`, make sure that the libraries that are searched contain only object modules and no load modules. If you are searching for longnamed or writable static objects, make sure that each library has a current directory member generated by the `C370LIB` utility. Without this directory, `autocall` can only be done on the member name of the object module and not on what is actually defined within the member.
- Only naturally reentrant code can be linked with the output of the prelinker. For more information on reentrancy, see *z/OS C/C++ Programming Guide*.

The error occurs at link time

If the error occurs at link time:

- If you have a link-time error while working with the C/C++ component of z/OS Language Environment, you can find diagnostics and debugging information in *z/OS DFSMS Program Management*.
- 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*.

Steps for diagnosing errors that occur at run time

Before you begin: If you are diagnosing run-time errors when executing with z/OS Language Environment, refer to *z/OS Language Environment Customization* and *z/OS Language Environment Debugging Guide*.

Perform the following steps to diagnose errors that occur at run time:

1. Specify one or more of the following compiler options, in addition to the options originally specified, to produce the most diagnostic information:

Option	Information produced
AGGREGATE	(C only). Aggregate layout.
ATTRIBUTE	(C++ only). Cross reference listing with attribute information.
CHECKOUT	(C only). Indication of possible programming errors.
DEBUG	Instructs the compiler to generate debug information based on the DWARF Version 3 debugging information format, which has been developed by the UNIX International Programming Languages Special Interest Group (SIG), and is an industry standard format.
EXPMAC	Macro expansions with the original source.
FLAG	Specifies the minimum message severity level that you want returned from the compiler.
GONUMBER	Generates line number information that corresponds to input source files. This applies to 31-bit compiles only.
INFO	(C++ only). Indication of possible programming errors.

INLINE	Inline Summary and Detailed Call Structure Reports. (Specify with the REPORT suboption.)
INLRPT	Generates a report on status of functions that were inlined. The OPTIMIZE option must also be specified.
LIST	Listing of the pseudo-assembly listing produced by the compiler.
OFFSET	Offset addresses of functions in the listing.
PPONLY	Completely expanded C or C++ source code, by activating the preprocessor (PPONLY). The output shows, for example, all the #include and #define directives.
SHOWINC	All included text in the listing.
SOURCE	Listing of the source file.
TEST	For 31-bit only, used to obtain information about the contents of variables at the point of the error, and to enable the use of Debug Tool.
XREF	Cross reference listing with reference, definition, and modification information. If you specify ATTRIBUTE, the listing also contains attribute information.

-
2. If the failure is in a statement that can be isolated, for example, an if, switch, for, while, or do-while statement, try placing the failing statement in the mainline code. If the problem is occurring as a result of a switch statement, make sure that you have “breaks” on all appropriate statements.
-
3. If you have used the compiler options RENT or NORENT in #pragma options or #pragma variable statements, and compiled your program at OPT(2), you can detect a possible pointer initialization error by compiling your program at OPT(0).
-
4. Check if you are running IBM C/370 Version 1 or Version 2 modules. Some IBM C/370 Version 1 and Version 2 modules may not be compatible with 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 for the Application Programmer* documents these solutions.
-
5. If IPA Link processed the program:
 - a. Ensure that the program functions correctly when compiled NOIPA at the same OPT level.
 - b. Subprograms (functions and C++ methods) which are not referenced will be removed unless appropriate “retain” directives are present in the IPA Link control file.
 - c. IPA Link may expose existing problems in the program:
 - Ensure that any coalesced global variables which are character strings have sufficient space to contain all characters plus an additional byte for the terminating null.
 - Ensure that there are no dependencies on the order in which data items or subprograms (functions, C++ methods) are generated.
 - d. Do the following to check for a code generation problem:

- Specify a different OPT level during IPA Link processing. If the program executes correctly, contact IBM service and be prepared to supply the relevant source (PPONLY) and object modules.
- Specify the option NOOPT during IPA Link processing. If the program executes correctly, contact IBM service and be prepared to supply the relevant source (PPONLY) and object modules.

If the program executes correctly at a different OPT level or NOOPT, perform a binary search for the IPA object file which contains the function for which code is incorrectly generated. Contact IBM service and be prepared to supply the relevant source (PPONLY) and object modules.

e. Do the following to check for an IPA optimization problem:

- Specify NOINLINE IPA(LEVEL(1)) during IPA Link processing.

If the program executes correctly, perform a binary search using INLINE IPA(LEVEL(1)) for the IPA object file which contains the function which is incorrectly optimized. Once you have located the IPA object file with the problem, use "noinline" directives within the IPA Link control file to determine the functions that are not correctly inlined. Contact IBM service and be prepared to supply the relevant source (PPONLY) and object modules and the IPA Link control file.

Functions that are inconsistently prototyped may cause problems of this type. Verify that all interfaces are consistent and complete.

- Specify IPA(LEVEL(0)) during IPA Link processing.

If the program executes correctly, perform a binary search using INLINE IPA(LEVEL(1)) for the IPA object file which contains the function which is incorrectly optimized. Contact IBM service and be prepared to supply the relevant source (PPONLY) and object modules.

- Specify IPA(LEVEL(1)) instead of IPA(LEVEL(2))

If you are linking with IPA Level 2, try linking with Level 1.

At this point, if you still encounter problems that you think are the result of the compilation, contact IBM support.

Steps for avoiding installation problems

Perform the following steps to avoid or solve most installation problems:

1. Review the step-by-step installation procedure that is documented in the z/OS Program Directory that is applicable to your environment.
-
2. Consult the PSP bucket as described on page 582.
-

If you still cannot solve the problem, 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.

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 9, "Binding z/OS C/C++ programs," on page 355. For a description of CXXMOD see "Prelinking and linking under TSO" on page 560. For a list of the old syntax REXX EXECs, see "Other z/OS C utilities" on page 601.

Name	Task Description
REXX EXECs for z/OS 31-bit C and C++	
C370LIB	Maintain an object library under TSO
CXXBIND	Generate an executable module under TSO
CXXMOD	Generate an executable module under TSO
Cataloged Procedures for z/OS C and z/OS C++	
EDCLIB	Maintain an object library
CNNPD1B	Binds C or C++ object compiled using the IPA(PDF1) and NOXPLINK options
CCNQPD1B	Binds C or C++ object compiled using the IPA(PDF1) and LP64 options
CNNXPD1B	Binds C or C++ object compiled using the IPA(PDF1) and XPLINK options
REXX EXECs for z/OS C	
CC	Compile (new syntax - recommended approach) Note: It applies to 31-bit only.
CDSECT	Run DSECT utility
CPLINK	Interactively prelink and link a C program (31-bit only)
GENXLT	Generate a translate table
ICONV	Run the character conversion utility
LOCALEDEF	Produce a locale object
Cataloged procedures for z/OS C	
CEEWL	Link a 31-bit program
CEEWLG	Link and run a 31-bit program
CEEXL	Bind an XPLINK z/OS C 31-bit program
CEEXLR	Bind and run an XPLINK z/OS C 31-bit program
CEEXR	Run an XPLINK z/OS C 31-bit program

Name	Task Description
EDCC	Compile a 31-bit program
EDCCB	Compile and Bind a 31-bit program
EDCCBG	Compile, bind, and run a 31-bit program
EDCCL	Compile and link-edit a 31-bit program
EDCCLG	Compile, link-edit, and run a 31-bit program
EDCCLIB	Compile and maintain an object library
EDCI	Run IPA Link step for a 31-bit program
EDCPL	Prelink and link-edit a 31-bit program
EDCCPLG	Compile, prelink, link-edit, and run a 31-bit program
EDCDSECT	Run the DSECT Conversion Utility
EDCGNXL	Generate a translate table
EDCICONV	Run the character conversion utility
EDCLDEF	Produce a locale object
EDCQB	Bind a 64-bit program
EDCQBG	Bind and run a 64-bit program
EDCQCB	Compile and bind a 64-bit program
EDCQCBG	Compile, bind, and run a 64-bit program
EDCXCB	Compile, and bind an XPLINK 31-bit program
EDCXCBG	Compile, bind, and run an XPLINK z/OS C 31-bit program
EDCXI	Run IPA Link step for XPLINK or 64-bit
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 a 31-bit program
CBCCB	Compile and bind a 31-bit program
CBCCBG	Compile, bind and run a 31-bit program
CBCB	Bind a 31-bit program
CBCBG	Bind and run a 31-bit program
CBCCL	Compile, prelink and link a 31-bit program
CBCCLG	Compile, prelink, link and run a 31-bit program
CB CG	Run a 31-bit program
CBCI	Run IPA Link step for a 31-bit program
CBCL	Prelink and link a 31-bit program
CBCLG	Prelink, link and run a 31-bit program
CBCQB	Bind a 64-bit program
CBCQBG	Bind and run a 64-bit program
CBCQCB	Compile and bind a 64-bit program

Name	Task Description
CBCQCBG	Compile, bind, and run a 64-bit program
CBCXB	Bind an XPLINK program
CBCXBG	Bind and run an XPLINK program
CBCXCB	Compile and bind an XPLINK program
CBCXCBG	Compile, bind, and run an XPLINK program
CBCXG	Run a z/OS C++ 31-bit or 64-bit program
CBCXI	Run IPA Link for XPLINK or 64-bit

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 LNGPRFX and LIBPRFX 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

Most customization for REXX EXECs is in CBC.SCCNUTL(CCNCCUST) and CEE.SCEECLST(CEL4CUST).

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 (for example, EDCI, EDCQI, CBCI, and CBCQI), the input IPA object. For prelink, link and bind procedures, the input object. If you do not specify the input data set name, you must use JCL statements to override the appropriate SYSIN DD statement in the cataloged procedure.
OUTFILE	Output module name and file characteristics. For the cataloged procedures ending in a link-edit, bind or go step, specify the name of the file where the load module is to be stored. For most other cataloged procedures, specify the name of the file where the object module is to be stored. If you do not specify an OUTFILE name, a temporary data set will be generated.
CPARM	Compiler options: If two contradictory options are specified, the last is accepted and the first ignored.

Parameter	Description
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 run-time (Go step) options and parameters: If two contradictory Language Environment run-time options are specified, the last is accepted and the first ignored.
CRUN	Compile step execution run-time parameters for the z/OS C/C++ compiler.
IRUN	IPA Link step run-time 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 41. Cross reference of data set used and job step

DD Statement	COMPILE	IPALINK	BIND	PLKED (Prelink)	LKED (Link-Edit)	GO (Run)	EDCALIAS (Object Library)
STEPLIB ¹	X	X	X	X		X	X
SYSCPRT	X	X					
SYSIN	X	X	X	X	X		X
SYSLIB	X	X	X	X	X		X
SYSLIN	X	X	X		X		
SYSLMOD			X		X		
SYSMOD				X			
SYSMSGs				X			X
SYSOUT	X	X		X			X
SYSPRINT	X	X		X	X	X	X

Table 41. Cross reference of data set used and job step (continued)

DD Statement	COMPILE	IPALINK	BIND	PLKED (Prelink)	LKED (Link-Edit)	GO (Run)	EDCALIAS (Object Library)
SYSUTx	X	X			X (SYSUT1)		
IPACNTL		X					

Note: ¹ Optional data sets, if the compiler and run-time 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 42. 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 or PDSE (DSORG=PO) and RECFM=VS, V, VB, VBS, F, FB LRECL≤32760. RECFM cannot be mixed. The LRECLs for F or FB RECFM must match. For more information on searching system header files, see "SEARCH NOSEARCH" on page 184.
COMPILE	SYSLIN	Data set for object module. One of the following: • RECFM=F or FS • RECFM=FB or FBS. It can be a PDS. LRECL=80
COMPILE	SYSOUT	Data set for displaying compiler error messages. LRECL=137, RECFM=VBA, BLKSIZE=882. (Defaults for SYSOUT=*)

Table 42. 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	STEPLIB	Data set for z/OS C/C++ compiler and run-time library modules. STEPLIB must be a PDS or PDSE (DSORG=P0) 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	Obsolete work data set. LRECL=80 and RECFM=F or FB or FBS.
COMPILE	SYSUT5, SYSUT6, SYSUT7, SYSUT8, SYSUT10, SYSUT14, SYSUT16, and SYSUT17	Work data sets. LRECL=3200, RECFM=FB, and BLKSIZE=3200*n (where n is an integer value).
COMPILE	SYSUT9	Work data set. LRECL=137, RECFM=VB, and BLKSIZE=137*n (where n is an integer value) in z/OS 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 or PDSE. 72≤LRECL≤32760, RECFM=VS, V, VB, VBS, F or FB (default is V).
COMPILE	USERLIB	User header files. Must be a PDS or PDSE. LRECL≤32760, and RECFM=VS, V, VB, VBS, F or FB. For more information on searching user header files, see “SEARCH NOSEARCH” on page 184.
IPALINK	SYSIN	Data set containing object module for the IPA Link step. LRECL=80 and RECFM=F or FB.

Table 42. Data set descriptions for cataloged procedures (continued)

In Job Step	DD Statement	Description and Supported Attributes (You should check the defaults at your site for SYSOUT=*)
IPALINK	IPACNTL	IPA Link control file directives. RECFM=VS, V, VB, VBS, F, FB, FBS, or FS, LRECL≤32760.
IPALINK	SYSLIB	IPA Link step secondary input. SYSLIB can be a mix of two types of libraries: <ul style="list-style-type: none"> Object module libraries. These can be PDSs (DSORG=P0) or PDSEs, with attributes RECFM=F or RECFM=FB, and LRECL=80. Load module libraries. These must be PDSs (DSORG=P0) with attributes RECFM=U and BLKSIZE≤32760. SYSLIB member libraries must be cataloged.
IPALINK	SYSLIN	Data set for generated object module. One of the following: <ul style="list-style-type: none"> RECFM=F or FS RECFM=FB or FBS
IPALINK	SYSOUT	Data set for displaying compiler error messages. LRECL=137, RECFM=VBA, BLKSIZE=882. (Defaults for SYSOUT=*).
IPALINK	STEPLIB	Data set for z/OS C/C++ compiler/run-time library modules. STEPLIB must be a PDS or PDSE (DSORG=P0) with RECFM=U, BLKSIZE≤32760.
IPALINK	SYSCPRT	Output data set for IPA Link step listings. LRECL=137, RECFM=VBA, BLKSIZE=882 (default for SYSOUT=*).
IPALINK	SYSUT1	Obsolete work data set. LRECL=80 and RECFM=F or FB or FBS.
IPALINK	SYSUT5, SYSUT6, SYSUT7, SYSUT8, SYSUT10, SYSUT14, SYSUT16, and SYSUT17	Work data sets. LRECL=3200, RECFM=FB, and BLKSIZE=3200*n (where n is an integer value).
IPALINK	SYSUT9	Work data set. LRECL=137, RECFM=VB, and BLKSIZE=137*n (where n is an integer value).

Table 42. Data set descriptions for cataloged procedures (continued)

In Job Step	DD Statement	Description and Supported Attributes (You should check the defaults at your site for SYSOUT=*)
BIND	SYSDEFSD	Output from binding a DLL (an application that exports symbols). LRECL=80 and RECFM=F or FB or FBS
BIND	SYSIN	Data set for additional object for the binder. It defaults to Dummy. LRECL=80 and RECFM=F, FB or FBS.
BIND	SYSLIB	Data set for binder automatic call libraries.
BIND	SYSLIN	Primary input data set for the binder One of the following: RECFM=F or FS RECFM=FB or FBS.
BIND	SYSLMOD	Output Program Object Library. PDSE with RECFM=U and BLKSIZE<=32760.
BIND	SYSPRINT	Data set for listing of binder diagnostic messages. LRECL=137, RECFM=VBA, BLKSIZE=882. (Default attributes for SYSOUT=*).
PLKED	STEPLIB	Data set containing prelink utility modules. STEPLIB must be a PDS or PDSE (DSORG=PO) and RECFM=U and BLKSIZE<=32760.
PLKED	SYSDEFSD	Output from prelinking a DLL (an application that exports symbols). LRECL=80 and RECFM=F or FB or FBS
PLKED	SYSIN	Data set containing object module for the prelink utility. This is the primary input data set. LRECL=80 and RECFM=F, FB or FBS.
PLKED	SYSLIB	Data set for automatic call libraries to be used with the prelinker. SYSLIB must be cataloged and LRECL=80 and RECFM=F or FB or FBS. DSORG=PO
PLKED	SYSMOD	Data set for output of the prelink utility LRECL=80 and RECFM=F or FB or FBS.
PLKED	SYSMSGGS	Data set containing prelink utility messages. LRECL=150, RECFM=F or FB or FBS and BLKSIZE=6150.

Table 42. Data set descriptions for cataloged procedures (continued)

In Job Step	DD Statement	Description and Supported Attributes (You should check the defaults at your site for SYSOUT=*)
PLKED	SYSOUT	Data set for the prelinker map. LRECL=80 and RECFM=F or FB or FBS
PLKED	SYSPRINT	Data set for listing of prelink utility diagnostic messages. LRECL=137, RECFM=VBA, BLKSIZE=882. (Default attributes for SYSOUT=*).
LKED	SYSLIB	Data set for z/OS C/C++ autocall library. SYSLIB must be a PDS or PDSE and have the attributes RECFM=U and BLKSIZE≤32760.
LKED	SYSLIN	Primary input data set for linkage editor One of the following: • RECFM=F or FS • RECFM=FB or FBS
LKED	SYSLMOD	Output load module library. RECFM=U and BLKSIZE≤32760.
LKED	SYSPRINT	Data set for listings and diagnostics produced by the linkage editor. One of the following: • LRECL=121, and RECFM=FA • LRECL=121, RECFM=FBA, and BLKSIZE=121*n (where n is less than or equal to 40).
LKED	SYSIN	Data set for additional object for the binder. It defaults to Dummy. LRECL=80 and RECFM=F, FB or FBS.
LKED	SYSUT1	Work data set. The data set attributes will be supplied by the linkage editor.
GO	STEPLIB	Run-time libraries. STEPLIB must be one or more PDSEs or PDSEs and have the attributes RECFM=U and BLKSIZE≤32760.
GO	CEEDUMP	Data set for error messages generated by Language Environment Dump Services. CEEDUMP must be a sequential data set and it must be allocated to SYSOUT, a terminal, or a unit record device, or a data set with the attributes RECFM=VBA, LRECL=125, and BLKSIZE=882.

Table 42. Data set descriptions for cataloged procedures (continued)

In Job Step	DD Statement	Description and Supported Attributes (You should check the defaults at your site for SYSOUT=*)
GO	SYSPRINT	Data set for listings and diagnostics from user program. LRECL=137, RECFM=VBA, BLKSIZE=882. (default attributes for SYSOUT=*).
OUTILITY	SYSIN	Input data set for object module to be added to the library. It can be sequential or partitioned (with a member name specified). LREL=80, RECFM=F or FB or FBS.
OUTILITY	SYSLIB	Library for which the member name is to be added (ADD); for which the member name is to be deleted (DEL); which is to be listed (MAP); for which the C370LIB-directory is to be built. This DD must point to a single partitioned data set. Concatenations cannot be used. Member names must not be specified. LREL=80, RECFM=F or FB or FBS.
OUTILITY	SYSOUT	Output data set for the C370LIB-directory map. It can be sequential or partitioned (with a member name specified). LREL=80, RECFM=F or FB or FBS.
OUTILITY	SYSMSG	Data set containing the input messages. LRECL=150, RECFM=F or FB or FBS.
OUTILITY	SYSPRINT	Data set diagnostics from the C370LIB program. The default is to SYSOUT=*. LRECL=137, RECFM=VBA, BLKSIZE=882

Examples using cataloged procedures

```

/*-----
/* Compile a Partitioned Data Set program with various options
/*-----
//EXAMPLE1 EXEC EDCC,
//      INFILE='PATRICK.TEST.PDSSRC(CPROG1)',
//      OUTFILE='PATRICK.TEST.OBJECT(CPROG1),DISP=SHR',
//      CPARAM='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',
//      CPARAM='OPT SOURCE XREF FLAG(E)'
//COMPILE.USERLIB DD DSNAME=PATRICK.HDR.FILES,DISP=SHR

```

Figure 57. 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 ',
//      CPARAM='OPT SOURCE SHOWINC LIST'
/*
//CCPDS EXEC CBCC,          * Compile C++ source PDS
//      INFILE='MIKE.CPP',
//      OUTFILE='MIKE.PROJECT.OBJ,DISP=SHR ',
//      CPARAM='NOOPT'

```

Figure 58. 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 59. Utilities for z/OS C

For a description of CXXMOD see “Prelinking and linking under TSO” on page 560.

Using the old syntax for CC

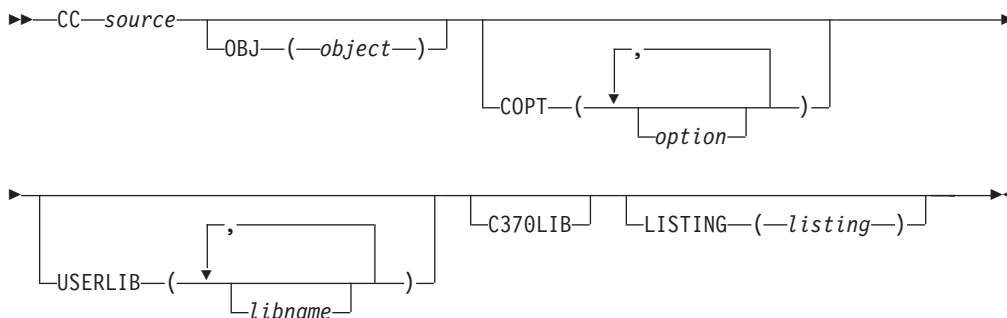
The CC command can now be invoked using a new syntax. At installation time, your system programmer can customize the CC EXEC to accept:

- 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 303. 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:

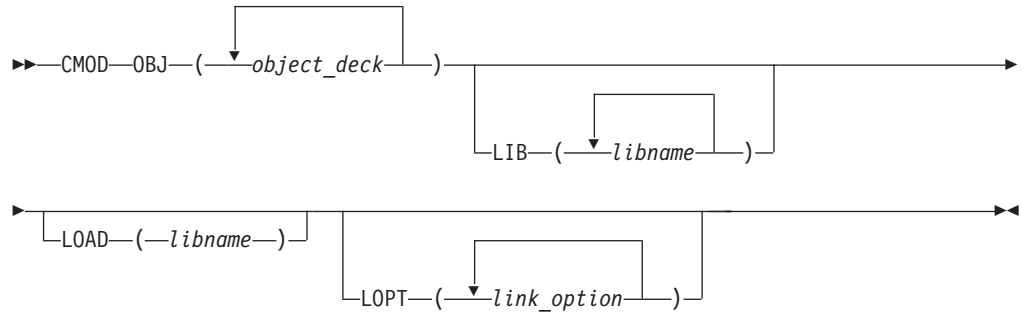
- When you are specifying a data set name, if the name is not enclosed in single quotation mark ('), your user prefix will be added to the beginning of the data set name. If the data set name is enclosed in single quotation marks, it will be treated as a fully qualified name.
- When you need to use spaces, commas, single quotation marks, or parentheses within a REXX EXEC option, the text must be placed inside a string using single quotation marks.
- If you want to use a single quotation mark inside a string, you must use two quotation marks in place of each quotation mark.

Example: The following example demonstrates these rules:

```
CC TEST.C(STOCK) COPT ('SEARCH(CLOTHES.H 'MARK.SUPPLY.C(ORDER)')')
```

Using CMOD

The CMOD REXX EXEC makes a call to LINK with the appropriate library. The syntax of the CMOD REXX EXEC is:



- OBJ** Specifies the object decks that you want to link.
- LIB** Specifies the libraries that are to be used to resolve external entries.
- LOAD** Specifies the output library in which the load module is to be stored.
- LOPT** Specifies the options that you want to pass to the linkage editor. All options are passed to the TSO LINK command.

A non-zero return code indicates that an error has occurred. For diagnostic information, refer to Appendix C, “Diagnosing problems,” on page 581. CMOD can also return the return code from LINK. See the appropriate document 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
- SYSUT16
- SYSUT17
- 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 a ddname, the compiler will use the appropriate default ddname from the above list.

VL or VL=1	Specifies that the sign bit is to be set to 1 in the last fullword of the address parameter.
DCB	Specifies the address of the control block for the partitioned data set that contains the compiler.
TASKLIB	Specifies the address of the DCB for the library that is to be used as the attached tasks library.

The return code from the compiler is returned in register 15.

If you code the macro instructions incorrectly, the compiler is not invoked, and the return code is 32. This error could be caused if the count of bytes in the alternative ddnames list is not a multiple of 8, or is not between 0 to 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.

Example of using the Assembler ATTACH macro (CCNUAAP)

```
*****
*
* This assembler routine demonstrates DD Name renaming
* (Dynamic compilation) using the Assembler ATTACH macro.
*
* In this specific scenario all the DD NAMES 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 60. 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 60. Using the assembler ATTACH Macro (Part 2 of 2)

Example of JCL for the Assembler ATTACH macro (CCNUAAQ)

```
/*-----  
/* Standard DDname Renaming (ASM ATTACH from driver program)  
/*   compiles           MYID.MYPROG.SOURCE(HELLO)  
/*   and places the object in MYID.MYPROG.OBJECT(HELLO)  
/*  
/*   User header files come from MYID.MYHDR.FILES  
/*   using MYCOMP as the compile time steplib.  
/*  
/*   Compilation is controlled by the assembler module named  
/*   CCNUAAP which is stored in MYID.ATTACHDD.LOAD  
/*  
/*   This example uses the Language Environment Library  
/*-----  
//G001001B EXEC PGM=CCNUAAP  
//STEPLIB DD DSN=MYID.ATTACHDD.LOAD,DISP=SHR  
//MYCOMP DD DSN=CBC.SCCNCMP,DISP=SHR  
// DD DSN=CEE.SCEERUN,DISP=SHR  
// DD DSN=CEE.SCEERUN2,DISP=SHR  
//NEWIN DD DSN=MYID.MYPROG.SOURCE(HELLO),DISP=SHR  
//NEWLIB DD DSN=CEE.SCEEH.H,DISP=SHR  
//NEWLIN DD DSN=MYID.MYPROG.OBJECT(HELLO),DISP=SHR  
//NEWPRINT DD SYSOUT=*  
//NEWCPRT DD SYSOUT=*,DCB=(RECFM=VBA,LRECL=137,BLKSIZE=882)  
//NEWPUNCH DD DSN=...  
//SYSTEM 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 61. JCL for the assembler ATTACH macro

Note that the sharing of resources between attached programs is not supported.

Example of using the Assembler LINK macro (CCNUAAR)

```
*****
*
* This assembler routine demonstrates DD Name renaming
* (Dynamic compilation) using the assembler LINK macro.
*
* In this specific scenario a subset of all the DDNAMES are
* renamed. The DDNAMES you do not want to rename are set to zero.
*
* The Compiler and the Library should be in the LPA, or should
* be specified on the STEPLIB DD in your JCL
*
*****
*
LINK      CSECT
          STM 14,12,12(13)
          BALR 3,0
          USING *,3
          LR 12,15
          ST 13,SAVE+4
          LA 15,SAVE
          ST 15,8(,13)
          LR 13,15
*
* Invoke the compiler using LINK macro
*
          LINK EP=CCNDRVR,PARAM=(OPTIONS,DDNAMES),VL=1
          L 13,4(,13)
          LM 14,12,12(13)
          SR 15,15
          BR 14
```

Figure 62. Using the assembler LINK macro (Part 1 of 2)

Example of JCL for the Assembler LINK macro (CCNUAAS)

```
/*-----  
/* Standard DDname Renaming using the assembler LINK macro  
/*   compiles           MYID.MYPROG.SOURCE(HELLO)  
/*   and places the object in MYID.MYPROG.OBJECT(HELLO)  
/*  
/*   User header files come from MYID.MYHDR.FILES  
/*  
/*   Compilation is controlled by the assembler module named  
/*   CCNUAAR that is stored in MYID.LINKDD.LOAD  
/*  
/*   This JCL uses the Language Environment Library.  
/*  
/*-----  
//G001003A EXEC PGM=CCNUAAR  
//STEPLIB DD DSN=CBC.SCCNCMP,DISP=SHR  
//        DD DSN=CEE.SCEERUN,DISP=SHR  
//        DD DSN=CEE.SCEERUN2,DISP=SHR  
//        DD DSN=MYID.LINKDD.LOAD,DISP=SHR  
//NEWIN DD DSN=MYID.MYPROG.SOURCE(HELLO),DISP=SHR  
//SYSLIB DD DSN=CEE.SCEEH.H,DISP=SHR  
//SYSLIN DD DSN=MYID.MYPROG.OBJECT(HELLO),DISP=SHR  
//SYSPRINT DD SYSOUT=*  
//NEWCPRT DD SYSOUT=*,DCB=(RECFM=VBA,LRECL=137,BLKSIZE=882)  
//SYSPUNCH DD SYSOUT=*  
//SYSTEM DD DUMMY  
//SYSUT1 DD DSN=...  
//SYSUT5 DD DSN=...  
//SYSUT6 DD DSN=...  
//SYSUT7 DD DSN=...  
//SYSUT8 DD DSN=...  
//SYSUT9 DD DSN=...  
//SYSUT10 DD SYSOUT=*  
//SYSUT14 DD DSN=...  
//SYSEVENT DD DSN=...  
//NEWRLIB DD DSN=MYID.MYHDR.FILES,DISP=SHR  
/*-----
```

Figure 63. JCL for the assembler LINK macro

Example of using the Assembler CALL macro (CCNUAAT)

```
*****
*
* This assembler routine demonstrates DD Name renaming
* (Dynamic compilation) using the Assembler CALL macro.
*
* In this specific scenario, a subset of all the DDNAMES are
* renamed. This renaming is accomplished by shortening
* the list of ddnames.
*
* The Compiler and the Library should be either 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 64. 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 64. Using the assembler CALL macro (Part 2 of 2)

Example of JCL for Assembler CALL macro (CCNUAAU)

```
/*-----  
/* Standard DDname Renaming using the assembler CALL macro  
/*   compiles           MYID.MYPROG.SOURCE(HELLO)  
/*   and places the object in MYID.MYPROG.OBJECT(HELLO)  
/*  
/*   User Header files come from MYID.MYHDR.FILES  
/*  
/*   Compilation is controlled by the assembler module named  
/*   CCNUAAT which is stored in MYID.CALLDD.LOAD  
/*  
/*   This JCL uses the Language Environment Library.  
/*  
/*-----  
//G001004C EXEC PGM=CCNUAAT  
//STEPLIB DD DSN=ABC.SCCNCMP,DISP=SHR  
// DD DSN=CEE.SCEERUN,DISP=SHR  
// DD DSN=CEE.SCEERUN2,DISP=SHR  
// DD DSN=MYID.CALLDD.LOAD,DISP=SHR  
//NEWIN DD DSN=MYID.MYPROG.SOURCE(HELLO),DISP=SHR  
//NEWLIB DD DSN=CEE.SCEEH.H,DISP=SHR  
//NEWLIN DD DSN=MYID.MYPROG.OBJECT(HELLO),DISP=SHR  
//NEWPRINT DD SYSOUT=*  
//NEWCPRT DD SYSOUT=*,DCB=(RECFM=VBA,LRECL=137,BLKSIZE=882)  
//NEWPUNCH DD DSN=...  
//SYSTEM DD DUMMY  
//NEWUT1 DD DSN=...  
//NEWUT4 DD DSN=...  
//NEWUT5 DD DSN=...  
//NEWUT6 DD DSN=...  
//SYSUT7 DD DSN=...  
//SYSUT8 DD DSN=...  
//SYSUT9 DD DSN=...  
//SYSUT10 DD SYSOUT=*  
//SYSUT14 DD DSN=...  
//NEWRLIB DD DSN=MYID.MYHDR.FILES,DISP=SHR  
/*-----
```

Figure 65. JCL for the assembler CALL macro

Appendix F. 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 99.

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 68 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 66. `simple.c`

```
1 add some;
2 errors in the header file;
```

Figure 67. `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 68. 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 68. Table 43 on page 618 describes the FILEID identifiers.

```
FILEID 0 1 0 10 ./simple.c
      A B C D E
```

Table 43. 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 Filend field

The following is an example of the FILEEND field from the sample SYSEVENT file that is shown in Figure 68 on page 617. Table 44 describes the FILEEND identifiers.

```
FILEEND 0 1 6
        A B C
```

Table 44. 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 68 on page 617. Table 45 describes the ERROR identifiers.

```
ERROR 0 1 0 0 3 3 0 0 CBCMMM E 12 26 Undeclared identifier add.
      A B C D E F G H I       J K L M
```

Table 45. 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 45. Explanation of the ERROR field layout (continued)

Column	Identifier	Description
E	Starting line number	The source line number for which the message was issued. A value of 0 indicates the message was not associated with a line number.
F	Starting column number	The column number or position within the source line for which the message was issued. A value of 0 indicates the message was not associated with a line number.
G	Reserved	Do not build a dependency on this identifier. It is reserved for future use.
H	Reserved	Do not build a dependency on this identifier. It is reserved for future use.
I	Message identifier	String Containing the message identifier.
J	Message severity character	I=Informational W=Warning E=Error S=Severe U=Unrecoverable
K	Message severity number	Return code associated with the message.
L	Message length	Length of message text.
M	Message text	String containing message text.

Appendix G. Accessibility

Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Volume I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

z/OS information

z/OS information is accessible using screen readers with the BookServer/Library Server versions of z/OS books in the Internet library at:

www.ibm.com/servers/eserver/zseries/zos/bkserv/

One exception is command syntax that is published in railroad track format; screen-readable copies of z/OS books with that syntax information are separately available in HTML zipped file form upon request to compinfo@ca.ibm.com.

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 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 or z/OS.e C/C++ programs.

Trademarks

The following terms are trademarks or registered 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
eServer	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	pSeries
QMF	RACF	Resource Link
SOM	S/370	S/390
SP	System Object Model	VisualAge
VM/ESA	VSE/ESA	z/OS
z/Architecture	zSeries	z/VM
400		

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States and/or other countries.

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

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

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

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

Standards

The following standards are supported in combination with the z/OS Language Environment:

- The C language is consistent with the International Standard C (ANSI/ISO-IEC 9899–1990 [1992]). This standard has officially replaced American National Standard for Information Systems-Programming Language C (X3.159–1989) and is technically equivalent to the ANSI C standard. The compiler supports the changes adopted into the C Standard by ISO/IEC 9899:1990/Amendment 1:1994. For more information on ISO, visit their web site at <http://www.iso.ch/>
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).

The following standards are supported in combination with the z/OS Language Environment and z/OS UNIX System Services:

- 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. For more information on IEEE, visit their web site at <http://www.ieee.org/>.
- A subset of IEEE P1003.1a Draft 6 July 1991, Draft Revision to Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language], copyright 1992 by the Institute of Electrical and Electronic Engineers, Inc.

- IEEE Std 1003.2—1992, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shells and Utilities, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- A subset of IEEE Std P1003.4a/D6—1992, IEEE Draft Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 2: Threads Extension [C language], copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.
- A subset of IEEE 754-1985 (R1990) IEEE Standard for Binary Floating-Point Arithmetic (ANSI), copyright 1985 by the Institute of Electrical and Electronic Engineers, Inc.
- X/Open CAE Specification, System Interfaces and Headers, Issue 4 Version 2, copyright 1994 by The Open Group
- X/Open CAE Specification, Networking Services, Issue 4, copyright 1994 by The Open Group
- X/Open Specification Programming Languages, Issue 3, Common Usage C, copyright 1988, 1989, and 1992 by The Open Group
- United States Government's Federal Information Processing Standard (FIPS) publication for the programming language C, FIPS-160, issued by National Institute of Standards and Technology, 1991

Glossary

This glossary defines technical terms and abbreviations that are used in z/OS 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: 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 alphabets 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'CO'.

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, 31, or 64 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. It invokes the compiler using host environment variables.

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.

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 generator. The part of the compiler that physically generates the object code.

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.

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.*

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 of 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 System Services 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 $m \leq n$, 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 breakpoints to instruct 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 `i conv()`.

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++ documents that are listed in “z/OS C/C++ and related publications” on page xvii 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
<C>	C
<D>	D
<E>	E
<F>	F
<G>	G
<H>	H
<I>	I

<J>	J	<h>	h
<K>	K	<i>	i
<L>	L	<j>	j
<M>	M	<k>	k
<N>	N	<l>	l
<O>	O	<m>	m
<P>	P	<n>	n
<Q>	Q	<o>	o
<R>	R	<p>	p
<S>	S	<q>	q
<T>	T	<r>	r
<U>	U	<s>	s
<V>	V	<t>	t
<W>	W	<u>	u
<X>	X	<v>	v
<Y>	Y	<w>	w
<Z>	Z	<x>	x
<left-square-bracket>	[<y>	y
<backslash>	\	<z>	z
<reverse-solidus>	/	<left-brace>	{
<right-square-bracket>]	<left-curly-bracket>	{
<circumflex>	^	<vertical-line>	
<circumflex-accent>	^	<right-brace>	}
<underscore>	_	<right-curly-bracket>	}
<low-line>	~	<tilde>	~
<grave-accent>	`		
<a>	a		
	b		
<c>	c		
<d>	d		
<e>	e		
<f>	f		
<g>	g		

```

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 . _ -

```

Figure 69.

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.

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 consist only of the following characters:

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 form 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 through 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 bytes 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

| **xlC.** A utility that uses an external configuration file to
| control the invocation of the compiler. xlC and related
| commands compile C and C++ source files. They also
| process assembler source files and object files.

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 USS). An element of the z/OS operating system, (formerly known as OpenEdition). z/OS UNIX System Services 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 and z/OS.e Planning for Installation*, GA22-7504
- *z/OS Summary of Message and Interface 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
- *z/OS 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 for the Application Programmer*, GC09-4913
- *IBM Open Class Library Transition Guide*, SC09-4948
- *Standard C++ Library Reference*, SC09-4949

z/OS Run-Time Library Extensions

- *C/C++ Legacy Class Libraries Reference*, SC09-7652
- *z/OS Common Debug Architecture User's Guide*, SC09-7653
- *z/OS Common Debug Architecture Library Reference*, SC09-7654
- *DWARF/ELF Extensions Library Reference*, SC09-7655

Debug Tool

- *Debug Tool* documentation, which is available at:
<http://www.ibm.com/software/awdtools/debugtool/library/>

z/OS Language Environment

- *z/OS Language Environment Concepts Guide*, SA22-7567
- *z/OS Language Environment Customization*, SA22-7564
- *z/OS Language Environment Debugging Guide*, GA22-7560
- *z/OS Language Environment Programming Guide*, SA22-7561

- *z/OS Language Environment Programming Reference*, SA22-7562
- *z/OS Language Environment Run-Time Application Migration Guide*, GA22-7565
- *z/OS Language Environment Writing Interlanguage Communication Applications*, SA22-7563
- *z/OS Language Environment Run-Time Messages*, SA22-7566

Assembler

- *HLASM Language Reference*, SC26-4940
- *HLASM Programmer's Guide*, SC26-4941

COBOL

- *COBOL 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*, SC34-5993
- *CICS Application Programming Reference*, SC34-5994
- *CICS Distributed Transaction Programming Guide*, SC34-5998
- *CICS Front End Programming Interface User's Guide*, SC34-5996
- *CICS Messages and Codes*, GC34-6003
- *CICS Resource Definition Guide*, SC34-5990
- *CICS System Definition Guide*, SC34-5988
- *CICS System Programming Reference*, SC34-5995
- *CICS User's Handbook*, SC34-5986
- *CICS Family: Client/Server Programming*, SC33-1435
- *CICS Transaction Server for z/OS Migration Guide*, GC34-5984
- *CICS Transaction Server for z/OS Release Guide*, GC34-5983
- *CICS Transaction Server for z/OS Installation Guide*, GC34-5985

DB2

- *DB2 Administration Guide*, SC18-7413
- *DB2 Application Programming and SQL Guide*, SC18-7415
- *DB2 ODBC Guide and Reference*, SC18-7423
- *DB2 Command Reference*, SC18-7416
- *DB2 Data Sharing: Planning and Administration*, SC18-7417
- *DB2 Installation Guide*, GC18-7418
- *DB2 Messages and Codes*, GC18-7422
- *DB2 Reference for Remote DRDA Requesters and Servers*, SC18-7424
- *DB2 SQL Reference*, SC18-7426
- *DB2 Utility Guide and Reference*, SC18-7427

IMS/ESA

- *IMS Version 8: Application Programming: Design Guide*, SC27-1287
- *IMS Version 8: Application Programming: Transaction Manager*, SC27-1289
- *IMS Version 8: Application Programming: Database Manager*, SC27-1286
- *IMS Version 8: Application Programming: EXEC DLI Commands for CICS and IMS Version 8:*, SC27-1288

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 for Catalogs*, SC26-7394
- *z/OS MVS Program Management: User's Guide and Reference*, SA22-7643
- *z/OS MVS Program Management: Advanced Facilities*, SA22-7644

INDEX

Special characters

-q options syntax 523

#pragma

See runtime options

A

abbreviated compiler options 50, 54

Abstract Code Unit (ACU) 119

accessibility 621

ACU (Abstract Code Unit) 119

AGGRCOPY compiler option 64

AGGREGATE compiler option 65, 587

aggregate layout 587

allocation, standard files with BPXBATCH 467

AMODE restriction 415

ANSIALIAS compiler option 67

ar utility

creating archive libraries 465

maintaining program objects 465

ARCHITECTURE compiler option 70

archive libraries

ar utility 465

creating 465

displaying the object files in 465

file naming convention for c89 use 465

ARGPARSE compiler option 73

argv, under TSO 413

ASCII compiler option 74

assemble

z/OS C and z/OS C++ source files 472

assembler

generation of C structures 448

macros 605

ATTACH assembler macro 605

ATTRIBUTE compiler option 75, 587

attributes, for DD statements 595

AUTO prelinker option 577

automatic library call

input to linkage editor 542

library search processing 386

prelinking and 555

processing 384

SYSLIB data set 541

B

BITF0XL DSECT utility option 440

BITFIELD compiler option 76

BLKSIZE DSECT utility option 448

BookManager documents xxii

BPARM JCL parameter 593

BPXBATCH program

invoking from TSO/E 416

invoking from z/OS batch 416

running an executable HFS file 415

syntax 467

C

C370LIB

EXEC 422

c89 utility

compiling and binding application programs 309

compiling source and object files 307

invoked through the make utility 311

linkage editor options 566

run by the make utility 307

c89/cc/c++ environment variable

_ACCEPTABLE_RC 486

_ASUFFIX 486

_ASUFFIX_HOST 486

_CCMODE 487

_CLASSLIB_PREFIX 487

_CLASSVERSION 487

_CLIB_PREFIX 488

_CMEMORY 488

_CMSGS 488

_CNAME 488

_CSUFFIX 489

_CSUFFIX_HOST 489

_CSYSLIB 489

_CVERSION 489

_CXXSUFFIX 490

_CXXSUFFIX_HOST 490

_DAMPLEVEL 490

_DAMPNAME 490

_DCB121M 491

_DCB133M 491

_DCB137 491

_DCB137A 491

_DCB3200 491

_DCB80 491

_DCBF2008 490

_DCBU 491

_DEBUG_FORMAT 492

_ELINES 492

_EXTRA_ARGS 492

_IL6SYSIX 492

_ILCTL 493

_ILMSGs 493

_ILNAME 493

_ILSUFFIX 493

_ILSUFFIX_HOST 493

_ILSYSIX 494

_ILSYSLIB 493

_ILXSYSIX 494

_ILXSYSLIB 494

_INCDIRS 494

_INCLIBS 494

_ISUFFIX 494

_ISUFFIX_HOST 494

_IXXSUFFIX 495

_IXXSUFFIX_HOST 495

_L6SYSIX 495

_L6SYSLIB 495

c89/cc/c++ environment variable (continued)

_LIBDIRS 495
 _LSYSLIB 495
 _LXSYSIX 496
 _LXSYSLIB 496
 _MEMORY 496
 _NEW_DATACLAS 496
 _NEW_DSNTYPE 497
 _NEW_MGMTCLAS 497
 _NEW_SPACE 497
 _NEW_STORCLAS 497
 _NEW_UNIT 497
 _NOCMDOPTS 497
 _OPERANDS 497
 _OPTIONS 498
 _OSUFFIX 498
 _OSUFFIX_HOST 498
 _OSUFFIX_HOSTQUAL 498
 _OSUFFIX_HOSTRULE 498
 _PMEMORY 500
 _PMSGs 500
 _PNAME 500
 _PSUFFIX 500
 _PSUFFIX_HOST 500
 _PSYSIX 500
 _PSYSLIB 501
 _PVERSION 501
 _SLIB_PREFIX 501
 _SNAME 501
 _SSUFFIX 501
 _SSUFFIX_HOST 502
 _SSYSLIB 502
 _STEPS 502
 _SUSRLIB 503
 _TMPS 503
 _WORK_DATACLAS 503
 _WORK_DSNTYPE 503
 _WORK_MGMTCLAS 503
 _WORK_SPACE 504
 _WORK_STORCLAS 504
 _WORK_UNIT 504
 _XSUFFIX 504
 _XSUFFIX_HOST 504
 IL6SYSLIB 492

c89/cc/c++ shell command

-W option
 compiler, prelinker, IPA linker and link editor
 options 479
 DLL and IPA extensions 479
 environment variables 486
 options 473
 specifying
 system and operational information to
 c89/cc/c++/cxx 486

CALL

assembler macro 605
 command 412
 command, under TSO 412

CALLBACKANY 95

cataloged procedures

descriptions

CBCB 366
 CBCCB 366
 CBCCBG 366
 CBCCL 556
 CBCCLG 556
 CBCI 591
 CBCL 556
 CBCLG 556
 CBCQB 366, 591
 CBCQBG 366, 591
 CBCQCB 366, 591
 CBCQCBG 366, 591
 CBCQI 366
 CBCXB 366
 CBCXBG 366
 CBCXCB 366
 CBCXCBG 366
 CBCXG 366
 CBCXI 591
 CEEWL 591
 CEEWLG 591
 CNNPD1B 366, 591
 CNNPD1L 591
 CNNXP1B 366, 591
 CNNXP1L 591
 data sets used by 595
 EDCB 366
 EDCC 591
 EDCCB 366, 591
 EDCCBG 366, 591
 EDCCCL 591
 EDCCLGB 591
 EDCCLIB 421, 591
 EDCCPLG 591
 EDCCSECT 591
 EDCGNXLT 456
 EDCI 591
 EDCICONV 453
 EDCLDEF 458
 EDCLIB 421, 591
 EDCPL 591
 EDCQB 366, 591
 EDCQBG 366, 591
 EDCQCB 366, 591
 EDCQCBG 366, 591
 EDCQI 366
 EDCXB 591
 EDCXCB 366
 EDCXCBG 366
 EDCXI 591
 EDCXLDEF 366

for binding 366

for compiling, prelinking and linking 555

for compiling, prelinking, linking and running 555

for prelinking and linking 555

for prelinking, linking and running 555

for specifying prelinker and linkage editor

options 556

specifying run-time options 411

CBCB cataloged procedure 366
 CBCCB cataloged procedure 366
 CBCCBG cataloged procedure 366
 CBCCL cataloged procedure 556
 CBCCLG cataloged procedure 556
 CBCL cataloged procedure 556
 CBCLG cataloged procedure 556
 CBCXB cataloged procedure 366
 CBCXBG cataloged procedure 366
 CBCXCB cataloged procedure 366
 CBCXCBG cataloged procedure 366
 CC REXX EXEC
 C370LIB parameter 423
 new syntax 303
 old syntax 601
 using under TSO 305
 using with HFS 304
 CCN message prefix 581
 CDSECT EXEC 452
 CEE message prefix 581
 CEESTART
 CSECT 543
 START compiler option 193
 STATICINLINE compiler option 194
 character
 trigraph representation 584
 unprintable 584
 characters
 converting from one code set to another 455
 CHARS compiler option 76
 CHECKOUT compiler option 77, 584, 587
 class libraries
 compiling with 327
 input to the prelinker 555
 class names used with CXXFILT 433
 CLASSNAME option of CXXFILT utility 435
 CMOD REXX EXEC, syntax 602
 CNNPD1B cataloged procedure 366
 CNNXPD1B cataloged procedure 366
 code set conversion utilities
 genxlt
 TSO 456
 usage 453
 z/OS Batch 456
 iconv
 TSO 454
 usage 453
 z/OS Batch 453
 command
 syntax diagrams xv
 COMMENT DSECT utility option 441
 COMPACT compiler option 79
 compile
 link-edit object file 472
 z/OS C and z/OS C++ source file 472
 compile-time error 584
 compiler
 See also compiler options
 See also compiling
 c89 utility interface to 307
 error messages 104
 compiler (*continued*)
 input 291, 301
 valid input/output file types 295
 listing
 include file option (SHOWINC) 187
 list inlined subprograms (INLRPT) 121
 object module option (LIST) 143
 source program option (SOURCE) 188
 z/OS C cross reference listing 255
 z/OS C error messages 255
 z/OS C external symbol cross reference 257
 z/OS C external symbol dictionary 257
 z/OS C heading information 254
 z/OS C includes section 255
 z/OS C inline report 256
 z/OS C object code 257
 z/OS C prolog 255
 z/OS C pseudo assembly listing 257
 z/OS C source program 255
 z/OS C static map 257
 z/OS C storage offset listing 257, 271
 z/OS C structure and union maps 255
 z/OS C++ cross reference listing 269
 z/OS C++ error messages 270
 z/OS C++ external symbol cross reference 271
 z/OS C++ external symbol dictionary 271
 z/OS C++ heading information 269
 z/OS C++ includes section 270
 z/OS C++ inline report 270
 z/OS C++ object code 271
 z/OS C++ prolog 269
 z/OS C++ pseudo assembly listing 271
 z/OS C++ source program 269
 z/OS C++ static map 271, 283
 object module optimization 169
 options to produce debug information
 AGGREGATE 587
 ATTRIBUTE 587
 CHECKOUT 584, 587
 DEBUG 587
 EXPMAC 587
 FLAG 587
 GONUMBER 587
 INFO 587
 INLINE 588
 INLRPT 588
 LIST 588
 MARGINS 584
 NOMARGINS 584
 NOOPTIMIZE 584
 NOSEQUENCE 584
 OFFSET 588
 OPTIMIZE 584
 PPONLY 584, 588
 SEQUENCE 584
 SHOWINC 588
 SOURCE 588
 TEST 588
 XREF 588
 output
 create listing file 294

compiler (*continued*)
 output (*continued*)
 create object module 294
 create preprocessor output 294
 create template instantiation output 294
 using compiler options to specify 292
 using DD statements to specify 302
 valid input/output file types 295
 compiler options
 #pragma options 47
 abbreviations 50, 54
 AGGRCOPY 64
 AGGREGATE | NOAGGREGATE 65
 ALIAS | NOALIAS 66
 ANSIALIAS | NOANSIALIAS 67
 ARCHITECTURE 70
 ARGPARSE | NOARGPARSE 73
 ASCII | NOASCII 74
 ATTRIBUTE | NOATTRIBUTE 75
 BITFIELD 76
 CHARS 76
 CHECKOUT | NOCHECKOUT 77
 COMPACT | NOCOMPACT 79
 COMPRESS | NOCOMPRESS 81
 CONVLIT | NOCONVLIT 82
 CSECT | NOCSECT 84
 CVFT | NOCVFT 86
 DBRMLIB 88
 DEBUG | NODEBUG 88
 defaults 50, 54
 DEFINE 92
 DIGRAPH | NODIGRAPH 93
 DLL | NODLL 95
 ENUMSIZE 97
 EVENTS | NOEVENTS 99
 EXECOPS | NOEXECOPS 100
 EXHINOEXH 101
 EXPMAC | NOEXPMAC 102
 EXPORTALL | NOEXPORTALL 103
 FASTT | NOFASTT 103
 FLAG | NOFLAG 104
 FLOAT 105
 GOFF | NOGOFF 110
 GONUMBER | NOGONUMBER 111
 HALT 112
 HALTONMSG | NOHALTONMSG 113
 IGNERRNO | NOIGNERRNO 114
 INFO | NOINFO 115
 INITAUTO 116
 INLINE | NOINLINE 118
 INLRPT | NOINLRPT 121
 IPA | NOIPA 122
 IPA considerations 44
 KEYWORD | NOKEYWORD 130
 LANGLVL 131
 LIBANSI | NOLIBANSI 142
 LIST | NOLIST 143
 LOCALE | NOLOCALE 145
 LONGNAME | NOLONGNAME 147
 LP64 | ILP32 148
 LSEARCH | NOLSEARCH 150

compiler options (*continued*)
 MARGINS | NOMARGINS 156
 MAXMEM | NOMAXMEM 157
 MEMORY | NOMEMORY 158
 Namemangling 159
 NESTINC | NONESTINC 161
 OBJECT | NOBJECT 162
 OBJECTMODEL 163
 OE | NOOE 165
 OFFSET | NOOFFSET 166
 OPTFILE | NOPTFILE 167
 OPTIMIZE | NOOPTIMIZE 169
 overriding defaults 43
 PHASEID 172
 PLIST 173
 PORT | NOPORT 174
 PPONLY | NOPPONLY 175
 pragma options 47
 REDIR | NOREDIR 178
 RENT | NORENT 179
 ROCONST | NOROCONST 180
 ROSTRING | NOROSTRING 181
 ROUND 182
 RTTI | NORTTI 183
 SEARCH | NOSEARCH 184
 SEQUENCE | NOSEQUENCE 185
 SERVICE | NOSERVICE 186
 SHOWINC | NOSHOWINC 187
 SOURCE | NOSOURCE 188
 specifying under TSO 305
 SPILL | SPILL 189
 SQL | NOSQL 191
 SSCOMM | NOSSCOMM 192
 START | NOSTART 193
 STATICINLINE | NOSTATICINLINE 194
 STRICT | NOSTRICT 195
 STRICT_INDUCTION |
 NOSTRICT_INDUCTION 196
 SUPPRESS | NOSUPPRESS 197
 TARGET 197
 TEMPINC | NOTEMPINC 203
 TEMPLATERECOMPILE |
 NOTEMPLATERECOMPILE 204
 TEMPLATEREGISTRY |
 NOTEMPLATEREGISTRY 205
 TERMINAL | NOTERMINAL 206
 TEST | NOTEST 206
 TMPLPARSE 211
 TUNE | NOTUNE 212
 UNDEFINE 214
 UNROLL 215
 UPCONV | NOUPCONV 216
 WARN64 | NOWARN64 216
 WSIZEOF | NOWSIZEOF 217
 XPLINK | NOXPLINK 218
 XREF | NOXREF 222
 compiling
 See also compiler
 See also compiler options
 dynamically with z/OS macro instructions 605
 TSO, under 303

- compiling (*continued*)
 - using cataloged procedures supplied by IBM 296
 - using compiler invocation command names
 - supported by c89 and xlc to compile and bind 309
 - using make to compile and bind 311
- compiling and binding in one step using compiler invocation command names supported by c89 and xlc 309
- COMPRESS compiler option 81
- concatenation
 - multiple libraries 302
- concatenation, multiple libraries 302
- configuration file for xlc 516
- continuation character
 - prelinker control statements 568
- control section (CSECT)
 - See* CSECT (control section)
- control statements
 - IMPORT, prelinker 569
 - INCLUDE 559
 - INCLUDE, prelinker 569
 - LIBRARY 559
 - LIBRARY, prelinker 570
 - linkage editor 558
 - processing 568
 - RENAME, prelinker 571
- convert
 - characters from one code set to another 455
 - source definitions for locale categories 459
- Convlit 79, 81, 82
- CONVLIT compiler option 82
- CPARM JCL parameter 593
- CPLINK REXX EXEC
 - example 564
 - syntax 562
- create executable files 472
- cross reference listing 588
- cross reference table
 - creating with XPLINK compiler option 218
 - creating with XREF compiler option 222
 - z/OS C listing 255
 - z/OS C++ listing 269
- CSECT (control section)
 - CEESTART 543
 - compiler option 84
 - pragma 539
- customizing locales 457
- CVFT compiler option 86
- CXX REXX EXEC
 - syntax 303
 - using under TSO 305
 - using with HFS 304
- CXXBIND REXX EXEC 377
- CXXFILT utility (*continued*)
 - class names 433
 - input under TSO 436
 - input under z/OS batch 435
 - options 434
 - overview 433
 - PROC for z/OS 435
 - regular names 433
- CXXFILT utility (*continued*)
 - special names 433
 - termination 437
 - termination under z/OS batch 436
 - TSO 436
 - unknown names 435
 - z/OS batch 435
- CXXMOD REXX EXEC
 - keyword parameters
 - LIB 561
 - LIST 562
 - LOAD 562
 - LOPT 561
 - OBJ 561
 - PLIB 561
 - PMAP 562
 - PMOD 561
 - POPT 561
 - syntax 560

D

- data sets
 - concatenating 302
 - for linking 540
 - for prelinking 536
 - supported attributes 595
 - usage 594
 - user prefixes 25, 34
- data types, preserving unsignedness 216
- DBRMLIB compiler option 88
- DD statement
 - for linkage editor data sets 540
 - for prelinker data sets 536
- ddname
 - alternative 605
 - defaults 594
- DEBUG compiler option 88, 587
- Debug Tool 16
- debugging
 - Debug Tool 16
 - error traceback (GONUMBER compiler option) 111
 - errors 77, 99
 - SERVICE compiler option 186
 - TEST compiler option 206
- DECIMAL DSECT utility option 441
- default
 - compiler options 50, 54
 - output file names 144
 - overriding compiler option 43
- DEFINE compiler option 92
- define local environments 459
- definition side-deck 544
- DEFSUB DSECT utility option 442
- digraphs, DIGRAPH compiler option 93
- disability 621
- disk search sequence
 - LSEARCH compiler option 150
 - SEARCH compiler option 184
- DLL (dynamic link library)
 - building 545

DLL (dynamic link library) *(continued)*
 definition side-deck 549
 description of 480
 DLL compiler option 95
 DLLNAME() prelinker option 545
 EXPORTALL compiler option 103
 IMPORT control statement 545
 link-editing 479
 NAME control statement 545
 prelinking 536
 prelinking a DLL 544
 prelinking a DLL application 544
 documents, licensed xxiii
 doublebyte characters, converting 455
 DSECT utility
 BITF0XL option 440
 BLKSIZE option 448
 COMMENT option 441
 DECIMAL option 441
 DEFSUB option 442
 EQUATE option 443
 HDRSKIP option 445
 INDENT option 445
 LOCALE option 445
 LOWERCASE option 446
 LP64 option 446
 LRECL option 448
 OUTPUT option 448
 PPCOND option 446
 RECFM option 448
 SECT option 439
 SEQUENCE option 447
 structure produced 448
 TSO 452
 UNIQUE option 447
 UNNAMED option 448
 z/OS batch 451
 DUP prelinker option 577
 Dynamic Link Libraries (DLLs)
 See DLLs
 dynamic link library (DLL)
 description of 480
 link-editing 479

E

EDC message prefix 581
 EDCB cataloged procedure 366
 EDCCB 368
 EDCCB cataloged procedure 366
 EDCCBG cataloged procedure 366
 EDCCLIB cataloged procedure 421
 EDCDSECT cataloged procedure 451
 EDCGNXLT cataloged procedure 456
 EDCICONV cataloged procedure 453
 EDCLDEF cataloged procedure 458
 EDCLDEF CLIST 458
 EDCLIB cataloged procedure 421
 EDCXCB cataloged procedure 366
 EDCXCBG cataloged procedure 366
 efficiency, object module optimization 169

ENTRY linkage editor control statement 543
 ENUMSIZE compiler option 97
 environment variable
 _ACCEPTABLE_RC
 used by c89/cc/c++ 486
 _ASUFFIX
 used by c89/cc/c++ 486
 _ASUFFIX_HOST
 used by c89/cc/c++ 486
 _CCMODE
 used by c89/cc/c++ 487
 _CLASSLIB_PREFIX
 used by c89/cc/c++ 487
 _CLASSVERSION
 used by c89/cc/c++ 487
 _CLIB_PREFIX
 used by c89/cc/c++ 488
 _CMEMORY
 used by c89/cc/c++ 488
 _CMSGS
 used by c89/cc/c++ 488
 _CNAME
 used by c89/cc/c++ 488
 _CSUFFIX
 used by c89/cc/c++ 489
 _CSYSLIB
 used by c89/cc/c++ 489
 _CVERSION
 used by c89/cc/c++ 489
 _CXXSUFFIX
 used by c89/cc/c++ 490
 _CXXSUFFIX_HOST
 used by c89/cc/c++ 490
 _DAMPLEVEL
 used by c89/cc/c++ 490
 _DAMPNAME
 used by c89/cc/c++ 490
 _DCB121M
 used by c89/cc/c++ 491
 _DCB133M
 used by c89/cc/c++ 491
 _DCB137
 used by c89/cc/c++ 491
 _DCB137A
 used by c89/cc/c++ 491
 _DCB3200
 used by c89/cc/c++ 491
 _DCB80
 used by c89/cc/c++ 491
 _DCBF2008
 used by c89/cc/c++ 490
 _DCBU
 used by c89/cc/c++ 491
 _DEBUG_FORMAT
 used by c89/cc/c++ 492
 _ELINES
 used by c89/cc/c++ 492
 _EXTRA_ARGS
 used by c89/cc/c++ 492
 _IL6SYSIX
 used by c89/cc 492

environment variable (continued)

<code>_IL6SYSLIB</code>	used by c89/cc	492
<code>_ILCTL</code>	used by c89/cc	493
<code>_ILMSGs</code>	used by c89/cc	493
<code>_ILNAME</code>	used by c89/cc/c++	493
<code>_ILSUFFIX</code>	used by c89/cc	493
<code>_ILSUFFIX_HOST</code>	used by c89/cc	493
<code>_ILSYSIX</code>	used by c89/cc/c++	494
<code>_ILSYSLIB</code>	used by c89/cc/c++	493
<code>_ILXSYSIX</code>	used by c89/cc/c++	494
<code>_ILXSYSLIB</code>	used by c89/cc/c++	494
<code>_INCDIRS</code>	used by c89/cc/c++	494
<code>_INCLIBS</code>	used by c89/cc/c++	494
<code>_ISUFFIX</code>	used by c89/cc/c++	494
<code>_ISUFFIX_HOST</code>	used by c89/cc/c++	494
<code>_IXXSUFFIX</code>	used by c89/cc/c++	495
<code>_L6SYSIX</code>	used by c89/cc/c++	495
<code>_L6SYSLIB</code>	used by c89/cc/c++	495
<code>_LIBDIRS</code>	used by c89/cc/c++	495
<code>_LSYSLIB</code>	used by c89/cc/c++	495
<code>_LXSYSIX</code>	used by c89/cc/c++	496
<code>_LXSYSLIB</code>	used by c89/cc/c++	496
<code>_MEMORY</code>	used by c89/cc/c++	496
<code>_NEW_DATACLAS</code>	used by c89/cc/c++	496
<code>_NEW_DSNTYPE</code>	used by c89/cc/c++	497
<code>_NEW_MGMTCLAS</code>	used by c89/cc/c++	497
<code>_NEW_SPACE</code>	used by c89/cc/c++	497
<code>_NEW_STORCLAS</code>	used by c89/cc/c++	497
<code>_NEW_UNIT</code>	used by c89/cc/c++	497
<code>_NOCMDOPTS</code>	used by c89/cc/c++	497
<code>_OPERANDS</code>	used by c89/cc/c++	497

environment variable (continued)

<code>_OPTIONS</code>	used by c89/cc/c++	498
<code>_OSUFFIX</code>	used by c89/cc/c++	498
<code>_OSUFFIX_HOST</code>	used by c89/cc/c++	498
<code>_OSUFFIX_HOSTQUAL</code>	used by c89/cc/c++	498
<code>_OSUFFIX_HOSTRULE</code>	used by c89/cc/c++	498
<code>_PLIB_PREFIX</code>	used by c89/cc/c++	499
<code>_PMEMORY</code>	used by c89/cc/c++	500
<code>_PMSGs</code>	used by c89/cc/c++	500
<code>_PNAME</code>	used by c89/cc/c++	500
<code>_PSUFFIX</code>	used by c89/cc/c++	500
<code>_PSUFFIX_HOST</code>	used by c89/cc/c++	500
<code>_PSYSIX</code>	used by c89/cc/c++	500
<code>_PSYSLIB</code>	used by c89/cc/c++	501
<code>_PVERSION</code>	used by c89/cc/c++	501
<code>_SLIB_PREFIX</code>	used by c89/cc/c++	501
<code>_SNAME</code>	used by c89/cc/c++	501
<code>_SSUFFIX</code>	used by c89/cc/c++	501
<code>_SSUFFIX_HOST</code>	used by c89/cc/c++	502
<code>_SSYSLIB</code>	used by c89/cc/c++	502
<code>_STEPS</code>	used by c89/cc/c++	502
<code>_SUSRLIB</code>	used by c89/cc/c++	503
<code>_TMPS</code>	used by c89/cc/c++	503
<code>_WORK_DATACLAS</code>	used by c89/cc/c++	503
<code>_WORK_DSNTYPE</code>	used by c89/cc/c++	503
<code>_WORK_MGMTCLAS</code>	used by c89/cc/c++	503
<code>_WORK_SPACE</code>	used by c89/cc/c++	504
<code>_WORK_STORCLAS</code>	used by c89/cc/c++	504
<code>_WORK_UNIT</code>	used by c89/cc/c++	504
<code>_XSUFFIX</code>	used by c89/cc/c++	504
<code>_XSUFFIX_HOST</code>	used by c89/cc/c++	504

environment variable (*continued*)

- LIBPATH
 - used by c89/cc/c++ 480
 - used to specify system and operational information to c89/cc/c++/cxx 486
 - used to specify system and operational information to xlc/xlC 515
- environment, defining local 459
- EQA message prefix 581
- EQUATE DSECT utility option
 - BIT suboption 443
 - BITL suboption 444
 - DEF suboption 444
- ER prelinker option 577
- error
 - compile-time 584
 - determining source of 581
 - link time 587
 - messages
 - directing to your terminal 206
 - re-creating 583, 584
 - run-time 587
- escape sequence 584
- escaping special characters 46, 298, 305
- EVENTS compiler option 99
- example
 - ccnghi1.c 330
 - ccnghi2.c 330
 - ccnghi3.c 331
 - ccnuaam 23
 - ccnuaan 24
 - ccnuaap 607
 - ccnuaaq 609
 - ccnuaar 610
 - ccnuaas 612
 - ccnuaat 613
 - ccnuaau 615
 - ccnubrc 32
 - ccnubrh.h 30
 - ccnuncl 40
 - clb3atmp.cxx 38
- examples
 - assembler macro 607
 - compile, link and run 34, 39
 - machine-readable xxii
 - naming of xxii
 - sample program 29
 - sample template program 37
 - softcopy xxii
 - z/OS C source 23
 - z/OS C++ source 29
- exception handling
 - compiler error message severity levels 104
 - linkage editor 543
- EXEC
 - JCL statement
 - GPARM parameter 412
 - specifying run-time options 411
 - statement
 - invoking linkage editor 557
 - invoking prelinker 557

EXEC (*continued*)

- supplied by IBM
 - CDSECT 452
 - DLLRNAME 591
 - GENXLT 456
 - ICONV 453
- executable
 - files
 - invoking z/OS load modules from the shell 415
 - placing z/OS load modules in the HFS 415
 - running 415
 - running, under z/OS batch 410
 - reentrant 510
- executable file
 - creating 472
- EXH compiler option 101
- EXPMAC compiler option 102, 587
- EXPORTALL compiler option 103
- external
 - entry points 66
 - names
 - long name support 147
 - prelinker 536
 - references
 - resolving 565
 - unresolved 577
 - variables
 - exporting 103
 - importing 103

F

- FASTTEMPINC compiler option 103
- feature test macro 316
- files
 - names
 - generated default 144
 - include files 317
 - user prefixes 25, 34
 - searching paths 150, 184
- FLAG compiler option 104, 587
- flag options syntax 525
- FLOAT
 - C/C++ programs 474
 - floating-point numbers 474
 - select format of floating-point numbers 474
- FLOAT compiler option 105
- functions
 - exporting 103
 - importing 103
 - linking 543

G

- genxlt utility
 - CLIST 456
 - TSO 456
 - usage 453
 - z/OS Batch 456
- GOFF compiler option 110

GONUMBER
 C/C++ programs 475
 debugging 475
 improved performance 475
 GONUMBER compiler option 111, 587
 GPARM
 JCL parameter 594
 parameter of EXEC statement 412

H

HALT compiler option 112
 HALTONMSG compiler option 113
 HDRSKIP DSECT utility option 445
 header files
See also include files
 system 302
 heading information
 for IPA Link listings 279
 for z/OS C listings 254
 for z/OS C++ listings 269
 HFS (Hierarchical File System)
 placing z/OS load modules 415

I

IBM message prefix 581
 iconv shell command 455
 iconv utility
 CLIST 454
 TSO 454
 usage 453
 z/OS Batch 453
 IGNERRNO compiler option 114
 IGZ message prefix 581
 ILP32 compiler option 148
 IMPORT statement
 syntax description 569
 improved debugging
 GONUMBER 475
 improved performance
 XPLINK 482
 IMS
 PLIST compiler option 173
 INCLUDE control statement
 for prelinking and linking 559
 linkage editor and 542
 syntax description 569
 z/OS C/C++ prelinker and 569
 include files
 naming 317
 nested 161
 preprocessor directive
 syntax 316
 record format 316
 SHOWINC compiler option 187
 system files and libraries
 OPTFILE compiler option 167
 SEARCH compiler option 184
 using 316

include files (*continued*)
 user files and libraries
 using 316
 INDENT DSECT utility option 445
 INFILE parameter 593
 INFO compiler option 115, 587
 INITAUTO compiler option 116
 inline
 report for IPA inliner 281
 z/OS C report 256
 z/OS C++ report 270
 INLINE compiler option 588
 description 118
 INLRPT compiler option 121, 588
 input
 compiler 291, 301
 linkage editor 541
 prelinker 537, 538
 record sequence numbers 185
 installation
 problems 589
 PTF (Program Temporary Fix) 582
 Interprocedural Analysis
See IPA
 Interprocedural Analysis (IPA) optimization
 explanation of 481
 IPA
 enabling 479
 explanation of 479
 invoking from the c89 utility 310
 IPA Compile step
 flow of processing 312
 IPA compiler option 122
 IPA Link step
 compiler options map listing section 280
 creating a DLL with IPA 341
 creating a module with IPA 329
 error source 585
 external symbol cross reference listing
 section 283
 external symbol dictionary listing section 283
 flow of processing 313
 global symbols map listing section 281
 invoking IPA from the c89 utility 348
 IPA inliner listing section 281
 IPA Link step control file 349
 listing heading information 279
 listing message summary 284
 listing messages section 283
 listing overview 223, 258, 272
 listing prolog 280
 object file directives 352
 object file map listing section 280
 overview 329
 partition map listing section 282
 pseudo assembly listing section 283
 source file map listing section 280
 storage offset listing section 283
 troubleshooting 352
 using profile-directed feedback 345
 object modules under IPA 294

IPA (*continued*)
 overview 312
 using cataloged procedures 297, 298
IPA (Interprocedural Analysis) optimization
 explanation of 481
IPA(OBJONLY) and c89 311
IPA(OBJONLY) compiler option 314
IPACNTL data set 595, 597
IPARM JCL parameter 593
IRUN JCL parameter 593

J

JCL (Job Control Language)
 C comments 193
 control statement
 See control statements
 ENTRY control statement 543
 specifying prelinker and linkage editor options 557,
 558

K

keyboard 621
KEYWORD compiler option 130

L

LANGLVL compiler option 131
LIB parameter CXXMOD EXEC 561
LIBANSI compiler option 142
LIBPATH environment variable
 used by c89/cc/c++ 480
library
 archive
 creating 465
 displaying the object files in 465
 file naming convention for c89 use 465
 use by application programs 465
 search sequence
 with LSEARCH compiler option 150
 with SEARCH compiler option 184
 z/OS Language Environment
 components 543
 required to run the compiler 291
 runtime 291
LIBRARY control statement
 linkage editor and 542
 prelinker and 559, 570
 using with linkage editor 559
LIBRARY JCL parameter 594
licensed documents xxiii
LINK
 assembler macro 605
 command
 input 564
 LOAD operand 565
 syntax 564
link time error 587
link-edit
 z/OS C and z/OS C++ object files 472

linkage editor
 creating a load module under z/OS batch 556
 errors 543
 function of 549
 INCLUDE statement and 542
 input to 541, 550
 LIBRARY statement and 542
 options
 MAPINOMAP 538
 specifying 556
 output 541, 542, 550
 requesting options with c89 566
 using c89 and xlc to compile and bindt 309
 using make to compile and bind 311
 using under TSO 560
linking 555
 See also linkage editor
 IBM-supplied class libraries 555
 multiple object modules 543
LIST compiler option 143, 588
LIST parameter CXXMOD EXEC 562
listings
 all included text 588
 cross reference 588
 from linkage editor 541
 from prelinker 538, 550
 include file option (SHOWINC) 187
 IPA Compile step, using 223, 258
 IPA Link step compiler options map 280
 IPA Link step external symbol cross reference 283
 IPA Link step external symbol dictionary 283
 IPA Link step global symbols map 281
 IPA Link step heading information 279
 IPA Link step inliner 281
 IPA Link step message summary 284
 IPA Link step messages 283
 IPA Link step object file map 280
 IPA Link step partition map 282
 IPA Link step prolog 280
 IPA Link step pseudo assembly 283
 IPA Link step source file map 280
 IPA Link step storage offset 283
 IPA Link step, using 272
 message summary, z/OS C 255
 message summary, z/OS C++ 270
 object code 588
 object library utility map 421
 object module option (LIST) 143
 source file 588
 using z/OS C++ 257
 z/OS C cross reference table 255
 z/OS C external symbol cross reference 257
 z/OS C external symbol dictionary 257
 z/OS C includes section 255
 z/OS C messages 255
 z/OS C object code 257
 z/OS C pseudo assembly listing 257
 z/OS C source program 255
 z/OS C static map 257
 z/OS C structure and union maps 255
 z/OS C, using 223

listings (*continued*)
z/OS C++ cross reference table 269
z/OS C++ external symbol cross reference 271
z/OS C++ external symbol dictionary 271
z/OS C++ includes section 270
z/OS C++ IPA Link step static map 283
z/OS C++ messages 270
z/OS C++ object code 271
z/OS C++ pseudo assembly listing 271
z/OS C++ source program 269
z/OS C++ static map 271

load library
storing object modules 566

load module
creating 540
inputs for 550

LOAD parameter CXXMOD EXEC 562

local environment, defining 459

local variables 190

locale
converting source definitions for categories 459
customizing 457
definition file 457
DSECT utility option 445
object 457

LOCALE compiler option 145

localedef shell command 459

localedef utility
TSO 458
z/OS batch 458

long names
definition of 536
LIBRARY control statement and 570
mapping to short names 539
RENAME control statement and 571
resolving undefined 555
support 147
unresolved 555
UPCASE prelink option and 577

LONGNAME compiler option 147

LookAt message retrieval tool xxiv

LOPT parameter CXXMOD EXEC 561

LOWERCASE DSECT utility option 446

LP64 compiler option 148

LP64 DSECT utility option 446

LPARM parameter 556, 593

LRECL (logical record length) parameter
DSECT utility option 448

LRECL DSECT utility option 448

LSEARCH compiler option 150

M

macro

assembler
ATTACH 605
CALL 605
compiling z/OS C/C++ programs with 605
LINK 605
expanded in source listing 102
expansion 587

macro (*continued*)
feature test 316

maintaining
objects in an archive library 465
programs through makefiles 466
programs with make using c89 311

make utility
compiling and binding application programs 311
compiling source and object files 307
creating makefiles 466
maintaining z/OS C/C++ application programs 466

Makedepend Utility 466

makefiles
creating 466
maintaining application programs 466

mangled name filter utility 433

map

load module 542
pragma 539
prelinker 538, 539, 550

MAP prelinker option 538, 550, 577

mapping
long names to short names
rules for 539
of load modules 557

MARGINS compiler option 156, 584

MAXMEM compiler option 157

MEMBER JCL parameter 594

memory

files, compiler work-files 158
MAXMEM compiler option 157
MEMORY compiler option 158
MEMORY prelinker option 577

message prefixes

CCN 581
CEE 581
EDC 581
EQA 581
IBM 581
IGZ 581
others 581
PLI 581

message retrieval tool, LookAt xxiv

messages

directing to your terminal 206
generate warning 115
on IPA Link step listings 283
on z/OS C compiler listings 255
on z/OS C++ compiler listings 270
specifying severity of 104

mismatches, type 584

MVS (Multiple Virtual System)

batch environment
running shell scripts and z/OS C/C++
applications 467

N

NAME control statement 537, 542

NAMEMANGLING compiler option

NAMEMANGLING compiler option 159

natural reentrancy
generating 179
linking 587

NCAL prelinker option 577

NESTINC compiler option 161

NOAGGREGATE compiler option 65

NOANSIALIAS compiler option 67

NOARGPARSE compiler option 73

NOASCII compiler option 74

NOATTRIBUTE compiler option 75

NOAUTO prelinker option 577

NOCALLBAKANY 95

NOCHECKOUT compiler option 77

NOCLASSNAME option of CXXFILT utility 435

NOCSECT compiler option 84

NOCVFT compiler option 86

NODEBUG compiler option 88

NODIGRAPH compiler option 93

NODLL compiler option 95

NODUP prelinker option 577

NOER prelinker option 577

NOEVENTS compiler option 99

NOEXECOPS compiler option 100

NOEXPMAC compiler option 102

NOEXPORTALL compiler option 103

NOFASTTEMPINC compiler option 103

NOFLAG compiler option 104

NOGOFF compiler option 110

NOGONUMBER compiler option 111

NOINFO compiler option 115

NOINLINE compiler option 118

NOINLRPT compiler option 121

NOIPA compiler option 122

NOLIBANSI compiler option 142

NOLIST compiler option 143

NOLOCALE compiler option 145

NOLONGNAME compiler option 147

NOLSEARCH compiler option 150

NOMAP prelinker option 577

NOMARGINS compiler option 156, 584

NOMAXMEM compiler option 157

NOMEMORY compiler option 158

NOMEMORY prelinker option 577

Non-XPLINK version of the Standard C++ Library and
c89 365

Non-XPLINK version of the Standard C++ Library and
z/OS batch 374

NONCAL prelinker option 577

NONESTINC compiler option 161

NOOBJECT compiler option 162

NOOE compiler option 165

NOOFFSET compiler option 166

NOOPTFILE compiler option 167

NOOPTIMIZE compiler option 169, 584

NOPPONLY compiler option 175

NOREDİR compiler option 178

NOREGULARNAME option of CXXFILT utility 434

NORENT compiler option 179

NOSEARCH compiler option 184

NOSEQUENCE compiler option 185, 584

NOSERVICE compiler option 186

NOSHOWINC compiler option 187

NOSIDEBYSIDE option of CXXFILT utility 434

NOSOURCE compiler option 188

NOSPECIALNAME option of CXXFILT utility 435

NOSPILL compiler option 189

NOSQL compiler option 191

NOSSCOMM compiler option 192

NOSTART compiler option 193

NOSTATICINLINE compiler option 194

NOSTRICT compiler option 195

NOSTRICT_INDUCTION compiler option 196

NOSUPPRESS compiler option 197

NOSYMMAP option of CXXFILT utility 434

NOTEMPINC compiler option 203

NOTEMPLATERECOMPILE compiler option 204

NOTEMPLATEREGISTRY compiler option 205

NOTERMINAL compiler option 206

NOTEST compiler option 206

Notices 623

NOUPCASE prelinker option 577

NOUPCONV compiler option 216

NOWARN64 compiler option 216

NOWIDTH option of CXXFILT utility 434

NOWSIZEOF compiler option 217

NOXPLINK compiler option 218

NOXREF compiler option 222

O

OBJ parameter for CXXMOD EXEC 561

object

- code 291
- library utility
 - adding object modules 421
 - deleting object modules 421
 - listing the contents 421
 - TSO 423
 - z/OS batch 421
- module
 - additional object modules as input 542
 - creating 559
 - DLL compiler option 95
 - EXPORTALL compiler option 103
 - link-editing multiple modules 543
 - LIST compiler option 143
 - OBJECT compiler option 162
 - optimization 169
 - storing in a load library 566
 - TARGET compiler option 197
 - z/OS C object listing 257
 - z/OS C++ object listing 271

OBJECT

- compiler option 162
- JCL parameter 594

object code, listing 588

object files

- object file browse 315
- working with 315

object files variations

- object file variation identification 316

- Object Library Utility
 - example under z/OS batch 421
 - long name support 421
 - map
 - heading 431
 - member heading 431
 - symbol definition map 432
 - symbol information 432
 - symbol source list 432
 - user comments 431
- OBJECTMODEL compiler option 163
- OE compiler option 165
- OFFSET compiler option 166, 588
- OGET utility 308, 415
- OGETX utility 415
- OMVS
 - OE compiler option 165
- OPARM JCL parameter 594
- OPTFILE compiler option 167
- optimization
 - object module 169
 - OPTIMIZE compiler option 169
 - storage requirements 169
 - TMPLPARSE compiler option 211
 - TUNE compiler option 212
- OPTIMIZE compiler option 169, 584
- optional features 472
- options
 - compiler
 - See compiler options
 - CXXFILT 433
 - linkage editor 542
 - prelinker
 - See prelinker, options
 - run-time 287
- OPUTX utility 415
- OUTFILE parameter 593
- output
 - from the linkage editor 541, 542
 - from the prelinker 538
- OUTPUT DSECT utility option 448

P

- PARM parameter 557
- passing arguments 287
- PCH (precompiled header)
 - See precompiled headers
- PDF documents xxii
- performance
 - C/C++ programs
 - FLOAT 474
 - XPLINK 482
- PHASEID compiler option 172
- PLI message prefix 581
- PLIB parameter CXXMOD EXEC 561
- PLIST compiler option 173
- PMAP parameter CXXMOD EXEC 562
- PMOD parameter CXXMOD EXEC 561
- POPT parameter CXXMOD EXEC 561
- PORT compiler option 174

- PPARM
 - JCL parameter 594
 - parameter 556
- PPCOND DSECT utility option 446
- PPONLY compiler option 175, 584, 588
- pragmas
 - csect 539
 - map 539
 - options 47
 - runopts 287
 - See runtime options
- prelinker
 - building and using DLLs 545
 - error source 586
 - function of 549
 - functions of 536
 - IBM-supplied class libraries 555
 - IMPORT statement and 569
 - INCLUDE statement and 569
 - input 537, 538, 549
 - LIBRARY statement and 570
 - load modules 586
 - map 538, 550
 - mapping long names to short names 539
 - messages from 538
 - options
 - MAPINOMAP 550
 - specifying 556
 - output from 537, 538, 549, 563
 - overview 535
 - RENAME statement and 571
 - resolving undefined symbols 555
 - under z/OS batch 558
 - usage 535
- preprocessor directives
 - effects of PPOONLY compiler option 175
 - include 316
- preprocessor, diagnostic information 588
- preventive service planning (PSP) bucket 582, 589
- primary data set
 - specifying input to the compiler 291
 - specifying input to the linkage editor 541
- primary input
 - compiler 291
 - linkage editor 541
 - to the linkage editor 541
 - to the prelinker 537
- processing a C program
 - z/OS C sample program, under TSO 26
 - z/OS C sample program, under z/OS Batch 25
- Profile-Directed Feedback 127
- programming errors 77
- PSP (preventive service planning) bucket 582, 589
- PTF (Program Temporary Fix) 582

R

- RECFM DSECT utility option 448
- record margins 156
- REDIR compiler option 178
- reentrancy 510

- reentrancy (*continued*)
 - linking 586
 - RENT compiler option 179
- reentrant code
 - linking 586
 - RENT compiler option 179
- region size 409
- regular names used with CXXFILT 433
- REGULARNAME option of CXXFILT utility 434
- RENAME control statement
 - mapping long names to short names 539
 - syntax 571
- RENT compiler option syntax 179
- REXX EXECs
 - supplied by IBM
 - C370LIB 591
 - CC, new syntax 591
 - CC, old syntax 601
 - CDSECT 591
 - CMOD 601, 602
 - CPLINK 562
 - CXXBIND 591
 - CXXMOD 591
 - EDCLDEF 458
 - GENXLT 456, 591
 - ICONV 454, 591
 - LOCALEDEF 591
- ROCONST compiler option 180
- ROSTRING compiler option 181
- ROUND compiler option 182
- RTTI compiler option 183
- run-time
 - errors 587
 - options
 - in the EXEC statement 411
 - recognize at run time 100
 - specifying 287
 - under z/OS batch 410
 - under z/OS UNIX System Services 415
 - specifying run-time environment 197
- running programs
 - TSO
 - example 412
 - specifying run-time options 413
 - with CALL TSO command 412
 - z/OS batch
 - BPXBATCH 415
 - example 411
 - with EXEC JCL statement 410
 - z/OS UNIX System Services application 414

S

- sample program
 - processing z/OS C under TSO 26
 - processing z/OS C under z/OS Batch 25
 - z/OS C source 23
 - z/OS C++ source 29
- SCEECPP library 561
- SCEELKED library 550, 561
- SEARCH compiler option 184

- SEARCH compiler option (*continued*)
 - using to compile z/OS C code 326
 - using to compile z/OS C++ code 327
- search sequence
 - library files 410
 - system include files 184
 - user include files 150
- secondary data set
 - libraries 302
 - secondary input to the linkage editor 542
- secondary input
 - compiler 292, 302
 - linkage editor 542
 - to the linkage editor 541
 - to the prelinker 537
- SECT DSECT utility option 439
- select format of floating-point numbers
 - FLOAT 474
- SEQUENCE compiler option 185, 584
- SEQUENCE DSECT utility option 447
- sequence numbers on input records 185
- SERVICE compiler option 186
- serviceability
 - C/C++ programs
 - GONUMBER 475
- shell
 - compiling and linking using c89 307
 - invoking load modules 415
 - using BPXBATCH to run commands or scripts 467
- short names
 - automatic library call processing 555
 - definition of 536
 - mapping 539
 - unresolved 555
- shortcut keys 621
- SHOWINC compiler option 187, 588
- SIDEBYSIDE option of CXXFILT utility 434
- singlebyte character conversions 455
- source
 - file listing 588
 - program
 - comment (SSCOMM compiler option) 192
 - compiler listing options 187, 188
 - file names in include files 317
 - generating reentrant code 179
 - input data set 291
 - margins 156
 - SEQUENCE compiler option 185
- source code
 - compiling using c89 305
 - z/OS C sample program 23
 - z/OS C++ example program 29
- SOURCE compiler option 188, 588
- source definitions
 - converting for locale categories 459
- special characters, escaping 46, 298, 305
- special names used with CXXFILT 433
- SPECIALNAME option of CXXFILT utility 435
- spill area
 - changing the size of 190
 - definition of 190

- spill area (*continued*)
 - pragma 190
- SPILL compiler option 189
- SQL compiler option 191
- SSCOMM compiler option 192
- standard files, allocation for BPXBATCH 467
- standards
 - ANSI compiler option 131
 - LIBANSI compiler option 142
- START compiler option 193
- statement
 - failure in 588
 - switch 588
- STATICINLINE compiler option 194
- STEPLIB
 - data set 594, 596, 597
 - ddname 536
 - prelinker 537
- storage optimization 169
- STRICT compiler option 195
- STRICT_INDUCTION compiler option 196
- structure and union maps, z/OS C compiler listing 255
- stub routines
 - contents of 543
 - in z/OS Language Environment 543
- SUPPRESS compiler option 197
- switch statement 588
- SYMMAP option of CXXFILT utility 434
- syntax diagrams
 - how to read xv
- SYSCPRT data set 294, 594, 596, 597
- SYSDEFSD data set
 - description 598
 - prelinker and 536, 537
- SYSEVENT data set
 - description of 596
- SYSIN data set for stdin
 - description of 595, 596
 - primary input to prelinker 536, 537, 549
 - primary input to the compiler 301
 - usage 594
- SYSLIB data set
 - description of 595, 597
 - linkage editor and 540, 541, 550
 - prelinker and 536, 537, 549
 - secondary input to linkage editor 542
 - specifying 302
 - usage 594
- SYSLIN data set
 - description of 595, 597
 - linkage editor and 540, 541, 549
 - primary input to linkage editor 541
 - usage 594
 - with OBJECT compiler option 294
- SYSLMOD data set 540, 541, 550, 594
- SYSMOD data set 536, 538, 550, 594
- SYSMSGs data set 536, 537, 594
- SYSOUT data set
 - description of 595, 597, 600
 - prelinker and 536, 538
 - usage 594

- SYSPRINT data set
 - linkage editor and 540, 541
 - prelinker and 536
 - usage 594
- system
 - files and libraries 167, 184
 - programmer, establishing library access 303, 412
- system header files 302
- SYSUT1 data set 540, 541, 595, 596
- SYSUT5-10 data sets 596

T

- TARGET compiler option 197
- tasks
 - avoiding installation problems
 - steps for 589
 - binding each compile unit under TSO
 - steps for 380
 - binding each compile unit under z/OS batch
 - steps for 368
 - binding each compile unit using c89
 - steps for 362
 - building a module in USS using PDF
 - steps for 347
 - building and using a DLL under TSO
 - steps for 380
 - building and using a DLL under z/OS batch
 - steps for 369
 - compiling, binding, and running the C example program using TSO commands
 - steps for 26
 - compiling, binding, and running the C example program using UNIX commands
 - steps for 27
 - compiling, binding, and running the C++ example program using TSO commands
 - steps for 35
 - compiling, binding, and running the C++ example program using UNIX commands
 - steps for 37
 - compiling, binding, and running the C++ template example program under z/OS batch
 - steps for 39
 - compiling, binding, and running the C++ template example program using UNIX commands
 - steps for 42
 - compiling, running, and binding the C++ template example program using TSO commands
 - steps for 41
 - completing the PDF process
 - steps for 345
 - diagnosing errors that occur at compile time
 - steps for 584
 - diagnosing errors that occur at IPA Link time
 - steps for 585
 - diagnosing errors that occur at run time
 - steps for 587
 - generating a reentrant load module in C
 - steps for 573

- tasks *(continued)*
 - generating a reentrant load module in C++
 - steps for 574
 - problem diagnosis using optimization levels
 - steps for 583
 - rebinding a changed compile unit under TSO
 - steps for 381
 - rebinding a changed compile unit under z/OS batch
 - steps for 375
 - rebinding a changed compile unit using c89
 - steps for 364
 - single final bind under TSO
 - steps for 379
 - single final bind under z/OS batch
 - steps for 367
 - single final bind using c89
 - steps for 361
 - steps for building and using a DLL using c89
 - steps for 363
- TEMPINC compiler option 203
- TEMPLATERECOMPILE compiler option 204
- TEMPLATEREGISTRY compiler option 205
- templates
 - create template instantiation output 294
 - program example 37
- TERMINAL compiler option 206
- test case, creating 583, 584
- TEST compiler option 206, 588
- TEXT deck 582
- TMPLPARSE compiler option 211
- trigraph 584
- TSO (Time Sharing Option)
 - compiling under 303
 - creating an object library
 - See Object Library Utility
 - LINK command 565
- TUNE compiler option 212
- type conversion, preserving unsignedness 216
- type conversions 216
- type mismatches 584

U

- UNDEFINE compiler option 214
- UNIQUE DSECT utility option 447
- unknown names input to CXXFILT utility 435
- UNNAMED DSECT utility option 448
- unprintable character 584
- UNROLL compiler option 215
- unsignedness preservation, type conversion 216
- UPCASE prelinker option 577
- UPCONV compiler option 216
- user
 - comments, object library utility map 431
 - include files
 - LSEARCH compiler option 150
 - SEARCH compiler option 184
 - specifying with #include directive 316
 - prefix 25, 34
- USL 7

- utilities
 - CXXFILT 433
 - mangled name filter 433
 - z/OS C 591
 - z/OS C, old syntax 601
 - z/OS C++ 591
 - z/OS UNIX System Services 465

W

- WARN64 compiler option 216
- WIDTH option of CXXFILT utility 434
- work data sets 594
- writable static
 - object library 421
 - prelinker and 538
 - relative offsets 536
- WSIZEOF compiler option 217

X

- xlC/xlC shell command
 - environment variables 515
 - specifying
 - system and operational information to xlC/xlC 515
- XPLINK
 - C/C++ programs 482
 - compiler option 218
 - extra performance linkages 482
 - improved performance 482
- XREF compiler option 222, 588

Z

- z/OS batch
 - compiling under 296
 - link-editing 559
 - running shell scripts and z/OS C/C++ applications 467
 - running your program 410
- z/OS UNIX System Services
 - compiling and binding using c89 307
 - compiling and binding using compiler invocation
 - command names supported by c89 and xlC 309
 - compiling and binding using make 311
 - maintaining objects in an archive library 465
 - maintaining through makefiles 466
 - OE compiler option 165
 - placing z/OS load modules in the HFS 415



Program Number: 5694-A01 and 5655-G52

Printed in the United States of America

SC09-4767-03

