

XL C/C++ for z/VM
1.3

User's Guide



Note:

Before you use this information and the product it supports, read the information in [“Notices” on page 119.](#)

This edition applies to version 7, release 3 of IBM® z/VM® (product number 5741-A09) and to all subsequent releases and modifications until otherwise indicated in new editions.

Last updated: 2022-09-07

© **Copyright International Business Machines Corporation 2003, 2022.**

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

Contents

Figures.....	vii
Tables.....	ix
About This Document.....	xi
Intended Audience.....	xi
Conventions and Terminology.....	xi
Syntax, Message, and Response Conventions.....	xi
Where to Find More Information.....	xiv
Links to Other Documents and Websites.....	xiv
How to Send Your Comments to IBM.....	xv
Summary of Changes for XL C/C++ for z/VM: User's Guide.....	xvii
SC09-7625-73, XL C/C++ for z/VM, 1.3 (September 2022)	xvii
SC09-7625-02, XL C/C++ for z/VM, 1.3 (September 2020)	xvii
SC09-7625-02, XL C/C++ for z/VM, 1.3 (September 2018)	xvii
Chapter 1. About XL C/C++ for z/VM.....	1
Differences between XL C/C++ for z/VM and z/OS XL C/C++.....	1
The C Language.....	1
The C++ Language.....	1
Common Features of the C and C++ Compilers.....	2
Class Library.....	2
Utilities.....	2
Language Environment.....	2
z/VM OpenExtensions.....	3
OpenExtensions Services.....	3
Applications with OpenExtensions Services.....	4
Applications with OpenExtensions Interoperability.....	4
Softcopy Examples.....	5
Chapter 2. C Example.....	7
Example of a C Program.....	7
CCNUAAM.....	7
CCNUAAN.....	8
Compiling, Binding, and Running the C Example.....	8
Non-XPLINK and XPLINK under CMS.....	9
Non-XPLINK and XPLINK under the OpenExtensions Shell.....	9
Chapter 3. C++ Examples.....	11
Example of a C++ Program.....	11
CCNUBRH.....	11
CCNUBRC.....	13
Compiling, Binding, and Running the C++ Example.....	15
Non-XPLINK and XPLINK under CMS.....	15
Non-XPLINK and XPLINK under the OpenExtensions Shell.....	15
Example of a C++ Template Program.....	16
CCNUTMP.....	16

Compiling, Binding, and Running the C++ Template Example.....	17
Under CMS.....	17
Under the OpenExtensions Shell.....	18
Chapter 4. Compiler Options.....	19
Specifying Compiler Options.....	19
Specifying Compiler Options Using #pragma options.....	21
Compiler Option Defaults.....	21
Summary of Compiler Options.....	21
Descriptions of Compiler Options.....	21
Unsupported Compiler Options.....	22
Compiler Options with Operational Differences.....	23
Using the C Compiler Listing.....	33
Using the C++ Compiler Listing.....	33
Chapter 5. Compiler Options under OpenExtensions.....	35
Specifying Compiler Options Using c89/cxx.....	35
c89/cxx Default Compiler Settings.....	35
c89 Selectable Compiler Settings.....	35
Format.....	35
Description.....	36
Feature Test Macros.....	37
Chapter 6. Runtime Options.....	39
Specifying Runtime Options.....	39
Runtime Options Using Language Environment.....	39
Chapter 7. Compiling a C/C++ Program.....	41
Invoking the XL C/C++ Compiler.....	41
GLOBAL Command for Using the Language Environment Library.....	41
Syntax of the CC EXEC.....	41
Specifying the Input File.....	42
Specifying Compiler Options.....	44
Creating Input Source Files.....	45
Specifying Output Files.....	45
Valid Input/Output File Types.....	47
Using Include Files.....	47
Determining If <i>filename</i> Is In Absolute Form.....	49
Using LSEARCH and SEARCH.....	51
Search Sequences for Include Files.....	52
With the NOOE option in effect.....	52
With the OE option in effect.....	53
Chapter 8. Binding and Running a C/C++ Program.....	55
Library Routine Considerations.....	55
Creating an Executable Program.....	55
Language Environment Sidedeck Files and TXTLIBs.....	56
CMOD Options.....	57
Examples.....	59
Using the LOAD and GENMOD Commands.....	59
Using the BIND Command.....	60
Using the LKED Command.....	61
Using FILEDEF to Define Input and Output Files.....	61
Preparing a Reentrant Program.....	61
Linking Modules for Interlanguage Calls.....	62
Running a Program.....	62
Making the Runtime Libraries Available for Execution.....	62

Making the Language Environment Library Available for VM/CMS.....	63
Search Sequence for Library Files.....	63
Specifying Runtime Options.....	63
Message Handling.....	64
Chapter 9. Compiling a C/C++ Program under OpenExtensions.....	65
Compiling with c89/cxx.....	65
Compiler Selection.....	66
Compiling and Building in One Step with c89/cxx.....	66
Using the make Utility.....	67
Chapter 10. Binding and Running a C/C++ Program under OpenExtensions.....	69
Using the c89 Utility to Bind and Create Executable Files.....	69
c89 Binder Options.....	69
Binder Options.....	70
Specifying Runtime Options under OpenExtensions.....	70
Running under OpenExtensions.....	70
OpenExtensions Application Program Environments.....	70
Placing a CMS Application Program Load Module in the File System.....	70
Running a CMS Module from the OpenExtensions Shell.....	71
Running an OpenExtensions XL C/C++ Application Executable File from the OpenExtensions Shell.....	71
Chapter 11. Object Library Utility.....	73
Creating an Object Library under VM/CMS.....	73
LINKLOAD EXEC.....	74
Object Library Utility Map.....	75
Chapter 12. Filter Utility.....	79
CXXFILT Options.....	79
SYMMAP NOSYMMAP.....	79
SIDEBYSIDE NOSIDEBYSIDE.....	80
WIDTH(width) NOWIDTH.....	80
REGULARNAME NOREGULARNAME.....	80
CLASSNAME NOCLASSNAME.....	80
SPECIALNAME NOSPECIALNAME.....	80
Unknown Type of Name.....	80
Running CXXFILT under VM/CMS.....	80
Chapter 13. DSECT Conversion Utility.....	83
DSECT Utility Options.....	83
SECT.....	84
BITFOXL NOBITFOXL.....	84
COMMENT NOCOMMENT.....	85
DEFSUB NODEFSUB.....	85
EQUATE NOEQUATE.....	86
HDRSKIP NOHDRSKIP.....	88
INDENT NOINDENT.....	88
LOCALE NOLOCALE.....	88
LOWERCASE NOLOWERCASE.....	88
OPTFILE NOOPTFILE.....	89
PPCOND NOPPCOND.....	89
SEQUENCE NOSEQUENCE.....	89
UNNAMED NOUNNAMED.....	89
OUTPUT.....	90
RECFM.....	90
LRECL.....	90

BLKSIZE.....	90
Generation of C Structures.....	90
Chapter 14. Code Set and Locale Utilities.....	95
Code Set Conversion Utilities.....	95
iconv Utility.....	95
genxlt Utility.....	96
localedef Utility.....	97
Chapter 15. OpenExtensions ar and make Utilities.....	101
OpenExtensions Archive Libraries.....	101
Creating Archive Libraries.....	101
Creating Makefiles.....	102
Appendix A. IBM-Supplied EXECs.....	103
Appendix B. XL C/C++ Compiler Return Codes and Messages.....	105
Appendix C. EXEC Error Messages.....	107
Appendix D. Runtime Error Messages and Return Codes.....	109
perror Messages.....	109
XL C/C++ Runtime Return Codes.....	109
Appendix E. Utility Messages.....	111
DSECT Utility Messages.....	111
Return Codes.....	111
Messages.....	111
Appendix F. Layout of the Events File.....	115
FILEID Field.....	115
FILEEND Field.....	116
ERROR Field.....	116
Notices.....	119
Programming Interface Information.....	120
Trademarks.....	120
Terms and Conditions for Product Documentation.....	120
IBM Online Privacy Statement.....	121
Bibliography.....	123
Where to Get z/VM Information.....	123
z/VM Base Library.....	123
z/VM Facilities and Features.....	124
Prerequisite Products.....	126
Related Products.....	126
Index.....	129

Figures

1. Celsius to Fahrenheit Conversion.....	7
2. User #include File for Conversion Program.....	8
3. Commands to Compile, Bind, and Run a C Program under VM/CMS.....	9
4. Commands to Compile, Bind, and Run a C Program under OpenExtensions.....	9
5. Header File for the Biorhythm Example (Part 1 of 2).....	11
6. Header File for the Biorhythm Example (Part 2 of 2).....	12
7. z/OS C++ Biorhythm Example Program (Part 1 of 3).....	13
8. z/OS C++ Biorhythm Example Program (Part 2 of 3).....	14
9. z/OS C++ Biorhythm Example Program (Part 3 of 3).....	14
10. Commands to Compile, Bind, and Run a C++ Program under CMS.....	15
11. Commands to Compile, Bind, and Run a C++ Program under OpenExtensions.....	15
12. C++ Template Program (Part 1 of 2).....	16
13. C++ Template Program (Part 2 of 2).....	17
14. Commands to Compile, Bind, and Run a C++ Template Program under CMS.....	17
15. Commands to Compile, Bind, and Run a C++ Template Program under OpenExtensions.....	18
16. Specifying a CMS Record Input File under VM/CMS (Example 1).....	42
17. Specifying a CMS Record Input File under VM/CMS (Example 2).....	43
18. Specifying a BFS Input File under VM/CMS (Example 1).....	44
19. Specifying a BFS Input File under VM/CMS (Example 2).....	44
20. Specifying Compiler Options under VM/CMS (Example 1).....	44
21. Specifying Compiler Options under VM/CMS (Example 2).....	45
22. Specifying Compiler Options for BFS Files.....	45
23. Testing If filename Is In Absolute Form.....	50

24. Determining If LSEARCH/SEARCH opt Is BFS Path.....	51
25. CMS Commands to Bind and Run a C Program.....	55
26. CMS Commands to Bind and Run a C++ Program.....	55
27. Example 1 - Using the CMOD EXEC.....	59
28. Example 2 - Using the CMOD EXEC.....	59
29. Example 3 - Using the CMOD EXEC.....	59
30. Example 4 - Using the CMOD EXEC.....	59
31. Using the LOAD and GENMOD commands.....	60
32. Running under CMS.....	64
33. Object Library Utility Map.....	76
34. Running the DSECT Utility under CMS.....	83
35. SIMPLE C.....	115
36. ERR H.....	115
37. Sample SYSEVENT file.....	115

Tables

1. Examples of Syntax Diagram Conventions.....	xii
2. c89 Option and Corresponding Compiler Option.....	37
3. Default CMS File Types and BFS Suffixes for Output Files.....	46
4. Valid Combinations of Source and Output File Types.....	47
5. Include Directive and Resulting File Names.....	49
6. Examples of Search Order for OpenExtensions.....	54
7. CMOD options.....	57
8. DSECT Utility Options, Abbreviations, and IBM-Supplied Defaults.....	84
9. Return Codes from the DSECT Utility.....	111
10. Explanation of the FILEID Field Layout.....	116
11. Explanation of the FILEEND Field Layout.....	116
12. Explanation of the ERROR Field Layout.....	116

About This Document

This document provides information about using IBM XL C/C++ for z/VM to implement (compile, bind, and run) C and C++ programs with Language Environment®. It contains guidelines for preparing C and C++ programs to run under z/VM.

This document includes information about the following topics:

- Introduction to IBM XL C/C++ for z/VM
- Differences between IBM XL C/C++ for z/VM and z/OS® XL C/C++
- How to compile, bind, and run a C/C++ program with IBM XL C/C++ for z/VM in the CMS environment of z/VM
- How to compile, bind, and run a C/C++ program with IBM XL C/C++ for z/VM in the z/VM OpenExtensions environment

Intended Audience

This information is intended for programmers who want to write C and C++ applications on the z/VM platform. To use this information, you must have a working knowledge of the C and C++ programming languages, Language Environment for z/VM, and z/VM OpenExtensions.

Conventions and Terminology

Throughout this document, the following conventions are used:

- *XL C/C++* is used to represent IBM XL C/C++ for z/VM.
- *z/VM* refers to z/VM 7.1 or later.
- *VM/CMS* is used to represent the CMS environment of z/VM.
- *Language Environment* is used to represent Language Environment for z/VM.
- *OpenExtensions* is used to represent the z/VM OpenExtensions environment.

The term *filename* is used to refer to both files in general, regardless of the specific file system in which they reside, and also more specifically to refer to the name component of a minidisk or shared file system file identifier. The intended usage should be clear from the context.

It is often necessary, however, to make a distinction between files that reside in the byte file system, and those that reside on minidisks or in the shared file system (but not in the byte file system). For convenience, the former will be referred to as *BFS files*, and the latter as *CMS files*.

The term *ddname* is used to refer to a data definition name. The relation of a ddname to one or more CMS files is achieved by using the FILEDEF command or for a BFS file by using the OPENVM PATHDEF CREATE command.

The term *FILEDEF* is used to refer to the data definition created by the use of the FILEDEF command.

The term *PATHDEF* is used to refer to the data definition created by the use of the OPENVM PATHDEF CREATE command.

The term *program module* is defined as the output of the binder. It is a collective term for program object and load module.





Syntax, Message, and Response Conventions

The following topics provide information on the conventions used in syntax diagrams and in examples of messages and responses.

How to Read Syntax Diagrams

Special diagrams (often called *railroad tracks*) are used to show the syntax of external interfaces.

To read a syntax diagram, follow the path of the line. Read from left to right and top to bottom.

- The  symbol indicates the beginning of the syntax diagram.
- The  symbol, at the end of a line, indicates that the syntax diagram is continued on the next line.
- The  symbol, at the beginning of a line, indicates that the syntax diagram is continued from the previous line.
- The  symbol indicates the end of the syntax diagram.

Within the syntax diagram, items on the line are required, items below the line are optional, and items above the line are defaults. See the examples in [Table 1 on page xii](#).





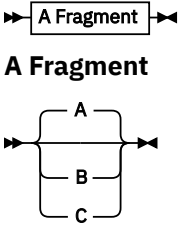
Table 1. Examples of Syntax Diagram Conventions	
Syntax Diagram Convention	Example
Keywords and Constants A keyword or constant appears in uppercase letters. In this example, you must specify the item KEYWORD as shown. In most cases, you can specify a keyword or constant in uppercase letters, lowercase letters, or any combination. However, some applications may have additional conventions for using all-uppercase or all-lowercase.	 KEYWORD 
Abbreviations Uppercase letters denote the shortest acceptable abbreviation of an item, and lowercase letters denote the part that can be omitted. If an item appears entirely in uppercase letters, it cannot be abbreviated. In this example, you can specify KEYWO, KEYWOR, or KEYWORD.	 KEYWO 
Symbols You must specify these symbols exactly as they appear in the syntax diagram.	<div><div>*</div><div>Asterisk</div></div> <div><div>:</div><div>Colon</div></div> <div><div>,</div><div>Comma</div></div> <div><div>=</div><div>Equal Sign</div></div> <div><div>-</div><div>Hyphen</div></div> <div><div>()</div><div>Parentheses</div></div> <div><div>.</div><div>Period</div></div>

Table 1. Examples of Syntax Diagram Conventions (continued)

Syntax Diagram Convention	Example
Variables A variable appears in highlighted lowercase, usually italics. In this example, <i>var_name</i> represents a variable that you must specify following KEYWORD.	<p>→ KEYWOrd — <i>var_name</i> →</p>
Repetitions An arrow returning to the left means that the item can be repeated. A character within the arrow means that you must separate each repetition of the item with that character. A number (1) by the arrow references a syntax note at the bottom of the diagram. The syntax note tells you how many times the item can be repeated. Syntax notes may also be used to explain other special aspects of the syntax.	<p>→ repeat →</p> <p>→ , repeat →</p> <p>→ repeat 1 →</p> <p>Notes: ¹ Specify <i>repeat</i> up to 5 times.</p>
Required Item or Choice When an item is on the line, it is required. In this example, you must specify A. When two or more items are in a stack and one of them is on the line, you must specify one item. In this example, you must choose A, B, or C.	<p>→ A →</p> <p>→ A B C →</p>
Optional Item or Choice When an item is below the line, it is optional. In this example, you can choose A or nothing at all. When two or more items are in a stack below the line, all of them are optional. In this example, you can choose A, B, C, or nothing at all.	<p>→ A →</p> <p>→ A B C →</p>
Defaults When an item is above the line, it is the default. The system will use the default unless you override it. You can override the default by specifying an option from the stack below the line. In this example, A is the default. You can override A by choosing B or C.	<p>→ A B C →</p>
Repeatable Choice A stack of items followed by an arrow returning to the left means that you can select more than one item or, in some cases, repeat a single item. In this example, you can choose any combination of A, B, or C.	<p>→ A B C →</p>

Table 1. Examples of Syntax Diagram Conventions (continued)	
Syntax Diagram Convention	Example
Syntax Fragment Some diagrams, because of their length, must fragment the syntax. The fragment name appears between vertical bars in the diagram. The expanded fragment appears in the diagram after a heading with the same fragment name. In this example, the fragment is named "A Fragment."	

Examples of Messages and Responses

Although most examples of messages and responses are shown exactly as they would appear, some content might depend on the specific situation. The following notation is used to show variable, optional, or alternative content:

- xxx**
Highlighted text (usually italics) indicates a variable that represents the data that will be displayed.
- []**
Brackets enclose optional text that might be displayed.
- { }**
Braces enclose alternative versions of text, one of which will be displayed.
- |**
The vertical bar separates items within brackets or braces.
- ...**
The ellipsis indicates that the preceding item might be repeated. A vertical ellipsis indicates that the preceding line, or a variation of that line, might be repeated.

Where to Find More Information

- This document is intended to be used in conjunction with the following documents:
- z/OS XL C/C++ documents (included in the IBM XL C/C++ for z/VM library)
 - Other IBM C/C++ programming documents (included in the IBM XL C/C++ for z/VM library)
 - z/VM and z/OS Language Environment documents (included in the z/VM library)
 - z/VM OpenExtensions documents (included in the z/VM library)
 - z/VM and z/OS Program Management Binder documents (included in the z/VM library)

For more information, see [“Bibliography” on page 123](#).

Links to Other Documents and Websites

The PDF version of this document contains links to other documents and websites. A link from this document to another document works only when both documents are in the same directory or database, and a link to a website works only if you have access to the Internet. A document link is to a specific edition. If a new edition of a linked document has been published since the publication of this document, the linked document might not be the latest edition.

How to Send Your Comments to IBM

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

To send us your comments, go to z/VM Reader's Comment Form (<https://www.ibm.com/systems/campaignmail/z/zvm/zvm-comments>) and complete the form.

If You Have a Technical Problem

Do not use the feedback method. Instead, do one of the following:

- Contact your IBM service representative.
- Contact IBM technical support.
- See [IBM: z/VM Support Resources \(https://www.ibm.com/vm/service\)](https://www.ibm.com/vm/service).
- Go to [IBM Support Portal \(https://www.ibm.com/support/entry/portal/Overview\)](https://www.ibm.com/support/entry/portal/Overview).

Summary of Changes for XL C/C++ for z/VM: User's Guide

This information includes terminology, maintenance, and editorial changes. Technical changes or additions to the text and illustrations for the current edition are indicated by a vertical line (|) to the left of the change.

SC09-7625-73, XL C/C++ for z/VM, 1.3 (September 2022)

This edition supports the general availability of z/VM 7.3. Note that the publication number suffix (-73) indicates the z/VM release to which this edition applies.

Language Environment upgrade

The z/VM Language Environment runtime libraries have been upgraded to z/OS 2.5 equivalence.

The following topic is updated:

- [“Object Library Utility Map” on page 75](#)

Miscellaneous updates for z/VM 7.3

The following topics are updated:

- [“Differences between XL C/C++ for z/VM and z/OS XL C/C++” on page 1](#)
- [“Unsupported Compiler Options” on page 22](#)

SC09-7625-02, XL C/C++ for z/VM, 1.3 (September 2020)

This edition supports the general availability of z/VM 7.2.

SC09-7625-02, XL C/C++ for z/VM, 1.3 (September 2018)

This edition supports the general availability of z/VM 7.1.

Chapter 1. About XL C/C++ for z/VM

XL C/C++ for z/VM is the language-centered C/C++ application development environment on the z/VM platform. It is a z/VM-enabled version of z/OS XL C/C++. IBM XL C/C++ for z/VM includes a C/C++ compiler component (referred to as the XL C/C++ compiler) and some C/C++ application development utilities.

Differences between XL C/C++ for z/VM and z/OS XL C/C++

>XL C/C++ for z/VM does *not* support the following z/OS XL C/C++ compiler features:

- ASCII support

The associated compiler option is ASCII.

- Assembler code generation

The associated compiler options are EPILOG, GENASM, METAL, and PROLOG.

- Host Performance Analyzer

- IEEE floating-point arithmetic

The associated compiler option is FLOAT(IEEE).

- Interprocedural analysis

The associated compiler option is IPA.

For the complete list of unsupported compiler options, see “Unsupported Compiler Options” on page 22.

Some supported z/OS XL C/C++ compiler options operate differently in XL C/C++ for z/VM. See “Compiler Options with Operational Differences” on page 23.

The C Language

The C language is a general-purpose, function-oriented 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, and it also provides many of the benefits of a low-level language.

The C++ Language

The C++ language is based on the C language, but incorporates support for object-oriented concepts. For a detailed description of the differences between C++ and C, see the *z/OS: XL C/C++ Language Reference* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147308/\\$file/cbclx01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147308/$file/cbclx01_v2r5.pdf)).

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 C and C++ Compilers

The C and C++ compilers offer many features to help your work:

- Optimization support.

The OPTIMIZE compiler option instructs the compiler to optimize the machine instructions it generates to produce faster-running object code to improve application performance at run time.

- Dynamic link libraries (DLLs) 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, XL 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 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.
- Locale-based internationalization support derived from the IEEE POSIX 1003.2-1992 standard. Also derived from the 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.
- Support for the Euro currency.

Class Library

IBM XL C/C++ for z/VM uses the following thread-safe class library:

- Standard C++ Library, including the Standard Template Library (STL), and other library features of Programming languages - C++ (ISO/IEC 14882:1998) and Programming languages - C++ (ISO/IEC 14882:2003(E))

The Standard C++ Library includes the following:

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

Utilities

IBM XL C/C++ for z/VM provides the following utilities:

- The CXXFILT utility to map C++ mangled names to the original source.
- The DSECT Conversion Utility to convert descriptive assembler DSECTs into C/C++ data structures.
- The localedef utility to read the locale definition file and produce a locale object that the locale-specific library functions can use.

Language Environment

IBM XL C/C++ for z/VM exploits the runtime library and common library of base routines available with z/VM and the C/C++ Language Environment for z/VM (referred to as Language Environment in this document).

Language Environment establishes a common runtime environment and common runtime services for language products, user programs, and other products.

The common execution environment is made up of data items and services performed by library routines available to a particular application running in the environment. The services that Language Environment can provide to your application include:

- Services that satisfy basic requirements common to most applications. These include support for the initialization and termination of applications, allocation of storage, support for interlanguage communication (ILC) and condition handling.
- Extended services often needed by applications. These functions are contained within a library of callable routines, and include interfaces to operating system functions and a variety of other commonly used functions.
- Runtime options that help the execution, performance tuning, performance, and diagnosis of your application.
- Access to operating system services. OpenExtensions services are available through the XL C/C++ language bindings.
- Access to language-specific library routines, such as the XL C/C++ functions.

z/VM OpenExtensions

z/VM OpenExtensions (referred to as OpenExtensions in this document) provides capabilities under z/VM to make it easier to implement or port applications in an open, distributed environment.

OpenExtensions Services

OpenExtensions services are available to XL C/C++ application programs through the C language bindings available with Language Environment.

Together, OpenExtensions, Language Environment, and XL C/C++ provide an application programming interface that supports industry standards.

Support for POSIX and UNIX-like interfaces includes:

- OpenExtensions services that include C-language programming interfaces defined by the IEEE POSIX 1003.1 standard (although the `fork()` function is only partially implemented) and subsets of the draft 1003.1a and 1003.1c standards, as well as OpenExtensions-unique extensions
- OpenExtensions shell and utilities, which provide a UNIX-like user interface and an application development environment for creating XL C/C++ programs for OpenExtensions, including the following utilities:

c89

Compile and link-edit XL C/C++ applications

make

Software build and maintenance tool

ar

Create and maintain library archives

- Byte file system (BFS), which provides the POSIX file system

This support offers:

- Program portability (with support for POSIX.1 and POSIX.1a) across multivendor operating systems
- A byte file system (BFS) in z/VM (with support for POSIX.1) including access to data in either CMS record files or the BFS
- A UNIX-like user interface (with support for POSIX.2) including access to services provided through the OpenExtensions shell and utilities
- Application threads (with support for a subset of POSIX.1c)
- OpenExtensions extensions which provide z/VM-specific support beyond the defined standards

This support is integrated with z/VM and Language Environment for use by both existing VM applications and for new OpenExtensions applications.

Application developers familiar with UNIX-like environments will find the OpenExtensions shell to be a familiar C application development environment. Those familiar with existing VM development environments may find that the OpenExtensions environment can enhance their productivity. For more information about the OpenExtensions shell and utilities, see [z/VM: OpenExtensions User's Guide](#).

Applications with OpenExtensions Services

To make use of OpenExtensions services, an XL C/C++ program must be an OpenExtensions POSIX XL C/C++ program with Language Environment runtime option POSIX(ON), or it must use the interoperability support for OpenExtensions. (See “Applications with OpenExtensions Interoperability” on page 4.)

An XL C/C++ program can make use of OpenExtensions services in one of the following ways:

- The program is invoked from another program, or from the OpenExtensions shell, using `spawn()` or one of the `exec` functions.
- The program is invoked using the POSIX `system()` call.
- The program is invoked from the CMS command line with the POSIX(ON) override option or through the OPENVM RUN command.

Functions with dependencies on the OpenExtensions kernel, such as `spawn()`, or the threading functions, such as `pthread_create()` are strictly limited to use within the OpenExtensions POSIX environment.

Applications with OpenExtensions Interoperability

OpenExtensions interoperability is used to describe the fact that:

- OpenExtensions applications running under POSIX(ON) can access traditional VM services and data.
- Traditional VM applications running under POSIX(OFF) can access the OpenExtensions services that permit access to BFS data.

For example, for functions such as `fopen()` and `freopen()`, the following statement will open a BFS file named `parts.instock`:

```
fopen("./parts.instock","r")
```

The next statement will open a CMS record file named PARTS INSTOCK:

```
fopen("//parts.instock","r")
```

Changing the runtime option POSIX(OFF) to POSIX(ON) will affect the environment in which these functions execute. For example, the following statement will open the BFS file named, if POSIX(ON) is in effect, and will open the CMS record file if POSIX(OFF) is in effect:

```
fopen("parts.instock","r")
```

Some of the C language functions that use OpenExtensions services can be invoked from applications running in the non-POSIX CMS environment, as specified with the runtime option POSIX(OFF).

For example, the following statement will open a BFS file named `parts.instock` whether the application is running under POSIX(OFF) or POSIX(ON):

```
open("parts.instock",O_RDONLY)
```

For more information on the C language functions available under the OpenExtensions environment that can be invoked by applications running in the non-POSIX VM environment, see the [XL C/C++ for z/VM: Runtime Library Reference](#).

Softcopy Examples

Most of the larger examples in this document and in these documents:

- *z/OS: XL C/C++ Language Reference* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147308/\\$file/cbclx01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147308/$file/cbclx01_v2r5.pdf))
- *z/OS: XL C/C++ Programming Guide* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147315/\\$file/cbcpx01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147315/$file/cbcpx01_v2r5.pdf))

are available in machine-readable form.

In the following documents, a label on an example indicates that the example is distributed in softcopy. The label is a file name on the IBM XL C/C++ for z/VM product disk. The labels have the form CCNxyyy or CLBxyyy, where x refers to a publication:

- R and X refer to the *z/OS: XL C/C++ Language Reference* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147308/\\$file/cbclx01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147308/$file/cbclx01_v2r5.pdf)).
- G refers to the *z/OS: XL C/C++ Programming Guide* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147315/\\$file/cbcpx01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147315/$file/cbcpx01_v2r5.pdf)).
- U refers to the *XL C/C++ for z/VM: User's Guide*.

Chapter 2. C Example

This chapter contains an example of the basic steps for compiling, binding, and running a C program.

If you have not yet compiled an XL C/C++ program or read the other chapters in this book, some concepts in this chapter may be unfamiliar. This chapter outlines the steps to compile, bind, and run your program under VM/CMS. Refer to relevant sections of the book for clarification as you read the examples of compiling, binding and running.

Example of a C Program

The following example shows a simple program that converts temperatures in Celsius to Fahrenheit. You can either enter the temperatures on the command line or be prompted for the temperature.

In this example, the main program calls the `convert` function to perform the conversion of 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 */
        int ch;

        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
        }
    }
    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
        }
    }
}

void convert(double c_temp) {                      8
    double f_temp = (c_temp * CONV + OFFSET);
    printf("%.2f Celsius is %.2f Fahrenheit\n", c_temp, f_temp);
}
```

Figure 1. Celsius to Fahrenheit Conversion

```
/******  
 * User include file: ccnuaan.h *  
*****/  
  
#define CONV (9./5.)  
#define OFFSET 32
```

9

Figure 2. User #include File for Conversion Program

1

This preprocessor directive includes the system file that contains the declarations of standard library functions, such as the `printf()` function used by this program.

The compiler searches for the file named `STDIO H` or for the member `STDIO` of the `VM/CMS MACLIBs`, depending on the options that are set. For a description of the file modes used in the search, see [“Search Sequences for Include Files”](#) on page 52.

2

This preprocessor directive includes a user file that defines constants that are used by the program.

The compiler searches for a file called `CCNUAAN`. See [“Search Sequences for Include Files”](#) on page 52 for a description of the file modes used in the search.

If the compiler cannot locate the file in the user libraries, the system libraries are searched.

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

7

This statement contains a call to the function `convert()`. It 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 C 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 C language, see the [XL C/C++ for z/VM: Runtime Library Reference](#).

Compiling, Binding, and Running the C Example

In general, you can compile, bind, and run C programs under CMS or the OpenExtensions shell. For more information, see [Chapter 7, “Compiling a C/C++ Program,”](#) on page 41 and [Chapter 8, “Binding and Running a C/C++ Program,”](#) on page 55.

Non-XPLINK and XPLINK under CMS

If the sample C program (CCNUAAM) was stored in CCNUAAM C L, and the sample include file (CCNUAAN) was stored in CCNUAAN H L, the following set of commands would compile, bind, and run the source code, using the Language Environment:

GLOBAL LOADLIB SCEERUN	1
CC CCNUAAM C L	2
-- or, for XPLINK --	
CC CCNUAAM C L (XPLINK	2
CMOD CCNUAAM	3
-- or, for XPLINK --	
CMOD CCNUAAM (XPLINK	3
GLOBAL LOADLIB SCEERUN	4
CCNUAAM	5

Figure 3. Commands to Compile, Bind, and Run a C Program under VM/CMS

- 1** Makes the library available to the compiler.
- 2** Compiles CCNUAAM C L and stores the object module in CCNUAAM TEXT A.
- 3** Using CCNUAAM TEXT A, created by the CC EXEC, generates an executable module called CCNUAAM MODULE using default options.
- 4** Makes the runtime library available to the executable module.
- 5** Runs CCNUAAM MODULE A using default options.

Non-XPLINK and XPLINK under the OpenExtensions Shell

If the sample C program (CCNUAAM) was stored in ./ccnuaam.c, and the sample include file (CCNUAAN) was stored in ./ccnuaan.h, the following set of commands would compile, bind, and run the source code, using the Language Environment:

Note: 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.

c89 -o //conv.module ccnuaam.c	1
-- or, for XPLINK --	
c89 -o //conv.module -Wc,xplink -Wb,x ccnuaam.c	1
cms global loadlib sceerun	2
conv	3

Figure 4. Commands to Compile, Bind, and Run a C Program under OpenExtensions

- 1** Compiles and binds ccnuaam.c, and generates an executable module called conv.
- 2** Makes the runtime library available to the executable module.
- 3** Runs conv using default options.

Chapter 3. C++ Examples

This chapter contains two examples that show the basic steps for compiling, binding, and running a C++ program.

If you have not yet compiled a XL C/C++ program or read the other chapters in this book, some concepts in this chapter may be unfamiliar. This chapter outlines the steps to compile, bind, and run your program under VM/CMS. Refer to relevant sections of the book for clarification as you read the examples of compiling, linking and running.

Example of a C++ Program

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

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

The example contains 2 files. File CCNUBRH contains the classes that are used in the main program. File CCNUBRC contains the remaining source code.

If you need more details on the constructs of the C++ language, see the [*XL C/C++ for z/VM: Runtime 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[];  
};
```

Figure 5. Header File for the Biorhythm Example (Part 1 of 2)

```

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

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

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)

```
//
// 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);
    }

    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;
}

```

Figure 8. z/OS C++ Biorhythm Example Program (Part 2 of 3)

```

// 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 9. z/OS C++ Biorhythm Example Program (Part 3 of 3)

Compiling, Binding, and Running the C++ Example

In general, you can compile, bind, and run C++ programs under CMS or the OpenExtensions shell. For more information, see Chapter 7, “Compiling a C/C++ Program,” on page 41 and Chapter 8, “Binding and Running a C/C++ Program,” on page 55.

Non-XPLINK and XPLINK under CMS

If the sample C++ program (CCNUBRC) was stored in CCNUBRC CXX L, and the sample include file (CCNUBRH) was stored in CCNUBRH H L, the following set of commands would compile, bind, and run the source code, using the Language Environment:

```
GLOBAL LOADLIB SCEERUN      1
CC CCNUBRC CXX L            2
-- or, for XPLINK --
CC CCNUBRC CXX L (XPLINK    2
CMOD CCNUBRC (C++           3
-- or, for XPLINK --
CMOD CCNUBRC (XPLINK C++    3
GLOBAL LOADLIB SCEERUN      4
CCNUBRC                     5
```

Figure 10. Commands to Compile, Bind, and Run a C++ Program under CMS

- 1** Makes the runtime library available to the compiler.
- 2** Compiles CCNUBRC CXX L and stores the object module in CCNUBRC TEXT A.
- 3** Using CCNUBRC TEXT A, created by the CC EXEC, generates an executable module called CCNUBRC MODULE using default options.
- 4** Makes the runtime library available to the executable module.
- 5** Runs CCNUAAM MODULE A using default options.

Non-XPLINK and XPLINK under the OpenExtensions Shell

If the sample C++ program (CCNUBRC) was stored in ./ccnubrc.cpp, and the sample include file (CCNUBRH) was stored in ./ccnubrh.h, the following set of commands would compile, bind, and run the source code, using the Language Environment:

Note: 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.

```
cxx -o //bio.module ccnubrc.cpp      1
-- or, for XPLINK --
cxa -o //bio.module -Wc,xplink -Wb,x ccnubrc.cpp  1
cms global loadlib sceerun           2
bio                                  3
```

Figure 11. Commands to Compile, Bind, and Run a C++ Program under OpenExtensions

- 1** Compiles and binds ccnubrc.cpp, and generates an executable module called bio.

2

Makes the runtime library available to the executable module.

3

Runs bio using default options.

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 C++ program that uses templates to perform simple operations on linked lists.

The main program, CCNUTMP (see “CCNUTMP” on page 16), uses three header files that are from the Standard C++ Library: list, string, and iostream. It has one class template: list.

CCNUTMP

```
#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';
}
```

Figure 12. C++ Template Program (Part 1 of 2)

```

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;

    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 13. 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 CMS and the OpenExtensions shell.

Under CMS

If the sample C++ template program (CCNUTMP) was stored in CCNUTMP CXX L, the following set of commands would compile, bind, and run the source code, using the Language Environment:

GLOBAL LOADLIB SCEERUN	1
CC CCNUTMP CXX L (TEMPL	2
-- or, for XPLINK --	
CC CCNUTMP CXX L (TEMPL XPLINK	2
CMOD CCNUTMP (C++	3
-- or, for XPLINK --	
CMOD CCNUTMP (XPLINK C++	3
GLOBAL LOADLIB SCEERUN	4
CCNUTMP	5

Figure 14. Commands to Compile, Bind, and Run a C++ Template Program under CMS

1

Makes the runtime library available to the compiler.

- 2** Compiles CCNUTMP CXX L and stores the object module in CCNUTMP TEXT A.
- 3** Using CCNUTMP TEXT A, created by the CC EXEC, generates an executable module called CCNUTMP MODULE using default options.
- 4** Makes the runtime library available to the executable module.
- 5** Runs CCNUTMP MODULE A using default options.

Under the OpenExtensions Shell

If the sample C++ template program (CCNUTMP) was stored in ./ccnutmp.cpp, the following set of commands would compile, bind, and run the source code, using the Language Environment:

cxx -o //ccnutmp.module -Wc,templ ccnutmp.cpp	1
-- or, for XPLINK --	
cxx -o //ccnutmp.module -Wc,templ,xplink -Wb,x ccnutmp.cpp	1
cms global loadlib sceerun	2
ccnutmp	3

Figure 15. Commands to Compile, Bind, and Run a C++ Template Program under OpenExtensions

- 1** Compiles ccnutmp.cpp, binds the created object module, and stores the load module in ccnutmp.
- 2** Makes the runtime library available to the executable module.
- 3** Runs ccnutmp using default options.

Chapter 4. Compiler Options

This chapter describes the options that you can use to alter the compilation of your program. For information on compiler options when compiling under OpenExtensions, see [Chapter 5, “Compiler Options under OpenExtensions,”](#) on page 35.

Specifying Compiler Options

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

- In the option list when you invoke the IBM-supplied CC EXEC. See [“Syntax of the CC EXEC”](#) on page 41 for details.
- In an options file. See [“OPTFILE | NOOPTFILE”](#) on page 30 for details.
- In a `#pragma` options preprocessor directive within your source file. See [“Specifying Compiler Options Using #pragma options”](#) on page 21 for details.

Compiler options specified on the command line can override compiler options used in `#pragma` options.

If two contradictory options are specified, the last one specified is accepted and the first ignored.

If you use one of the following compiler options, the option name is inserted at the bottom of your object module indicating that it was used:

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

Compiler Option	Usage Note
GONUMBER	
HOT	
IGNERRNO	
ILP32	
INITAUTO	
INLINE	
LANGLVL	
LIBANSI	
LOCALE	
LONGNAME	
MAXMEM	
NAMEMANGLING	(C++ compile only)
OBJECTMODEL	
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)
TUNE	
UNROLL	
UPCONV	(C compile only)
WSIZEOF	
XPLINK	

Specifying Compiler Options Using #pragma options

You can use the #pragma options preprocessor directive to override the default values for compiler options. Remember that compiler options specified on the command line can override compiler options used in #pragma options. For complete details on the #pragma options preprocessor directive, see the *z/OS: XL C/C++ Language Reference* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147308/\\$file/cbclx01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147308/$file/cbclx01_v2r5.pdf)).

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

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

Notes:

1. When you specify conflicting attributes either explicitly or implicitly by the specification of other options, the last explicit option is accepted. No diagnostic message is issued to indicate that any options are overridden.
2. When you specify the SOURCE compiler option on the command line, your listing will contain an options list indicating the options in effect at invocation. The values in the list are the options specified on the command line or the default options specified at installation. These values do not reflect any options that are specified in the #pragma options directive.

Compiler Option Defaults

You can alter the compilation of your program by specifying compiler options when you invoke the compiler or when you use the preprocessor directive, #pragma options, in your source program. Options that you specify when you invoke the compiler override installation defaults or compiler options specified through a #pragma options directive.

The defaults of the compiler options supplied by IBM can be changed to other selected defaults when XL C/C++ is installed. To determine the current defaults, compile a program with only the SOURCE compiler option specified. In the listing generated, you can view the options that are in effect at invocation; that is, the settings that result from the interaction of the command-line options and the defaults that were specified at installation. The listing does not reflect options specified in #pragma options in the source file being compiled.

Summary of Compiler Options

See the corresponding section in the *z/OS: XL C/C++ User's Guide* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147307/\\$file/cbcux01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147307/$file/cbcux01_v2r5.pdf)).

Descriptions of Compiler Options

For details of specific compiler options, see the corresponding section in the *z/OS: XL C/C++ User's Guide* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147307/\\$file/cbcux01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147307/$file/cbcux01_v2r5.pdf)). For the most part, the compiler options will work as described. However, some

compiler options are not supported, and other compiler options have operational differences, as identified in the following sections.

Unsupported Compiler Options

XL C/C++ for z/VM does *not* support the following z/OS XL C/C++ compiler options:

ARMODE

All functions compiled in access-register mode

ASCII

ASCII support

CICS®

CICS support

DBRMLIB

Database request module for SQL option

DFP

Decimal floating-point support

EPILOG

Supports user-supplied epilog code

FLOAT(IEEE)

Supports IEEE floating-point arithmetic

GENASM

Generates HLASM source code

HGPR

64-bit General Purpose Register support

IPA

Interprocedural analysis

LP64

AMODE 64 support

MAKEDEP

Creates dependency files for the make utility

METAL

Generates HLASM code with no Language Environment runtime dependencies

PREFETCH

Inserts prefetch instructions automatically

PROLOG

Supports user-supplied prolog code

REPORT

Produces pseudo-C code listing files for IPA

RTCHECK

Generates compare-and-trap instructions

SQL

Supports embedded SQL statements

SPLITLIST

Splits a listing into multiple files for IPA

TEMPINC

Specifies a location for C++ template instantiation files

Note: Use the `TEMPLATEREGISTRY` option instead.

WARN64

Supports diagnostic messages for 32-bit to 64-bit conversions

Compiler Options with Operational Differences

The following z/OS XL C/C++ compiler options are supported but operate differently in IBM XL C/C++ for z/VM.

ARCHITECTURE

The default for this option is ARCH(4). Note that code generated for groups 5 and above (z/Architecture® mode) might not execute on CMS.

CSECT | NOCSECT



This option does not accept a qualifier. If a qualifier is specified it is ignored.

If CSECT is specified, it will name the code, static and test sections of the object module as *basename#suffix*, where:

basename

is one of the following:

- File name of the primary source file, if it is a CMS record file
- Source file name, with path and extension information removed, if it is a BFS file

suffix

is one of the following:

C

For code CSECT

S

For static CSECT

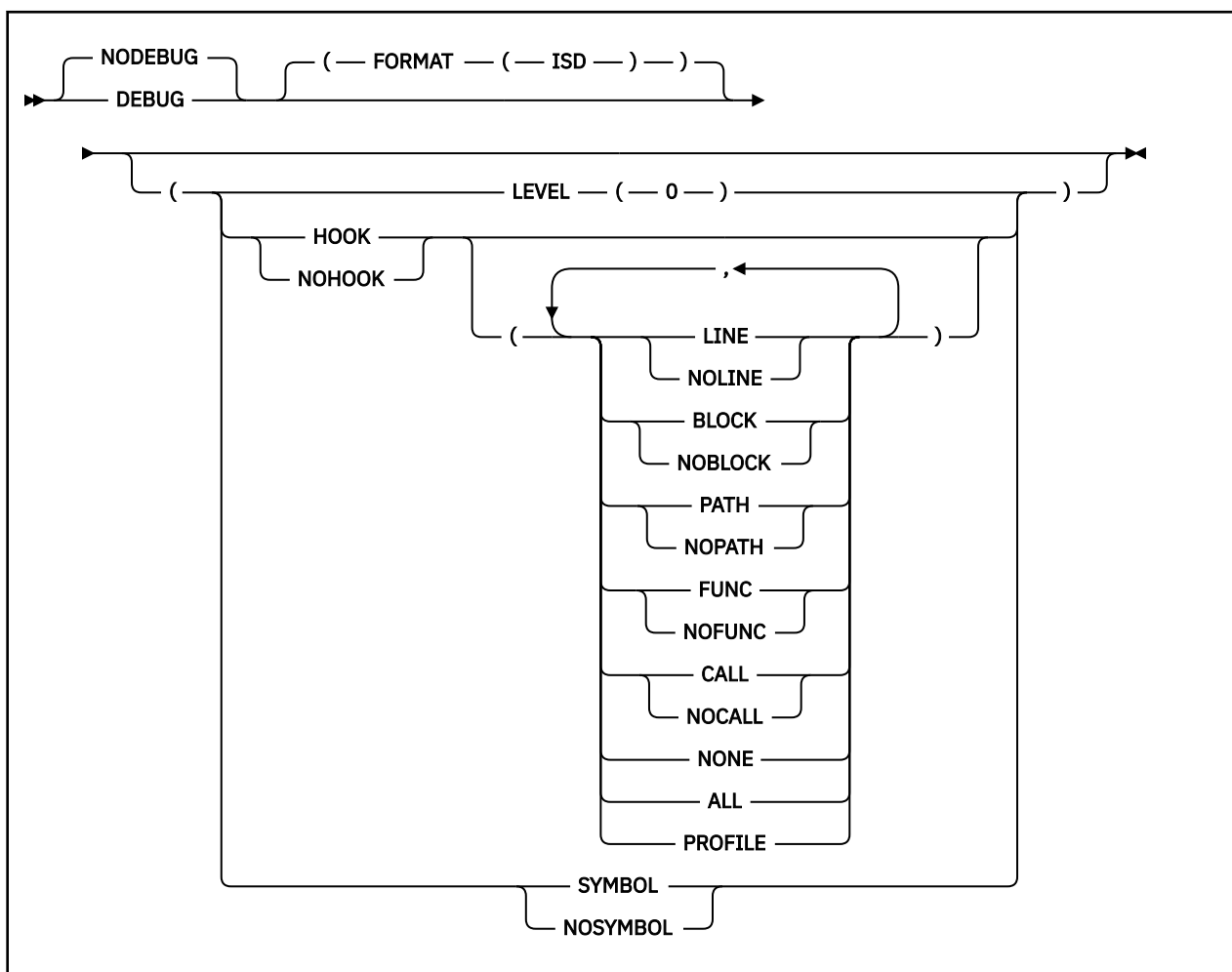
T

For test CSECT

Specifying CSECT allows the compiler to generate long CSECT names. For NOGOFF, if the compiler option LONGNAME is not in effect when CSECT is specified, the compiler turns it on, and issues an informational message. For GOFF, both NOLONGNAME and LONGNAME options are supported.

When CSECT is specified, the code, data and test CSECTs are always generated. The test CSECT has content only when the TEST option is also specified.

DEBUG | NODEBUG



Defaults:

- NODEBUG
- For FORMAT, the default is ISD.
- For LEVEL, the default is LEVEL(0).
- For HOOK, the defaults are HOOK(ALL) for NOOPTIMIZE and HOOK(NONE,PROFILE) for OPTIMIZE.
- For SYMBOL, the defaults are SYMBOL for NOOPTIMIZE and NOSYMBOL for OPTIMIZE.

FORMAT(ISD)

produces the same debug information as the TEST option.

LEVEL(0)

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

HOOK

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]
```

Note: If the OPTIMIZE compiler option is specified, the only valid suboptions for HOOK are CALL and FUNC. If other suboptions are specified, they will be ignored.

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

- 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

The same as specifying `NOLINE`, `NOBLOCK`, `NOPATH`, `NOCALL`, and `NOFUNC`. It instructs the compiler to suppress all hook instructions.

ALL

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

The same as specifying CALL and FUNC.

SYMBOL

generates symbol information that gives you access to variable and other symbol information.

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

ENUMSIZE



Default: ENUM(SMALL)

SMALL

specifies that enumerations occupy a minimum amount of storage, which is either 1, 2, or 4 bytes of storage, depending on the range of the enum constants.

INT

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

1

specifies that enumerations occupy 1 byte of storage.

2

specifies that enumerations occupy 2 bytes of storage

4

specifies that enumerations occupy 4 bytes of storage.

EVENTS | NOEVENTS



The EVENTS option creates an events file that contains error information and source file statistics.

EVENTS(*filename*) places the events information in the specified file. *filename* can be a CMS record or BFS file. If you do not specify a file name for the EVENTS option, the compiler generates a file name as follows:

- For CMS source files, the events information is written to a file that has the name of the source file and a file type of SYSEVENT.
- For BFS source files, the events information is written to a file that has the name of the source file and a `.err` extension.

The compiler ignores `#line` directives when the EVENTS option is active, and issues a warning message.

INLRPT | NOINLRPT



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. *filename* can be a CMS record or BFS file. If you do not specify a file name for the INLR option, the compiler generates a file name as follows:

- For CMS record source files, the report is created in a file that has the source file name, file type LISTING, and file mode A.
- For BFS source files, the report is created in a BFS file that has the source file name with a `.lst` extension.

The NOINLR option can optionally take a *filename* suboption. This file name then becomes the default. If you subsequently use the INLR option without *filename*, the compiler uses the file name that you specified in the earlier specification or NOINLR. For example,

```
CC HELLO (NOINLR(/hello.lst) INLR OPT
```

is the same as specifying:

```
CC HELLO (INLR(/hello.lis) OPT
```

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:

```
CC HELLO (NOINLR(/hello.lis) INLR(/n1.lis) NOINLR(/test.lis) INLR
CC HELLO (INLR(/test.lis)
```

If you specify file names with the SOURCE, LIST or INLRPT options, all the listing sections are combined into the last file name specified.

LIST | NOLIST



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. *filename* can be a CMS record or BFS file. If you do not specify a file name for the LIST option, the compiler generates a file name as follows:

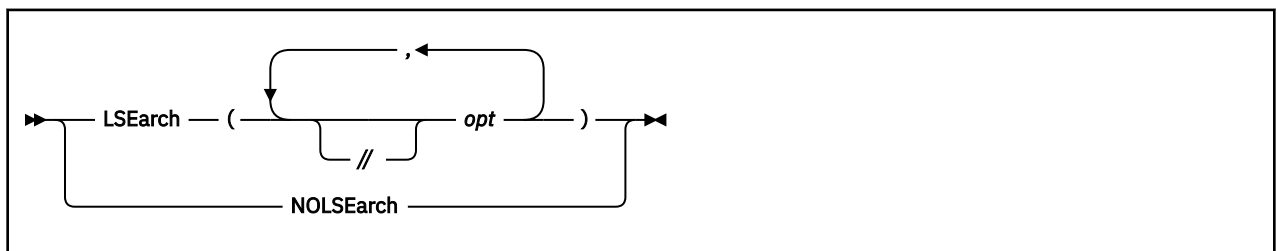
- For CMS record source files, the listing is created in a file that has the source file name, file type LISTING, and file mode A.
- For BFS source files, the listing is created in a BFS file that has the source file name with a .lst extension.

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

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

If you specify file names with the SOURCE, LIST or INLRPT options, all the listing sections are combined into the last file name specified.

LSEARCH | NOLSEARCH

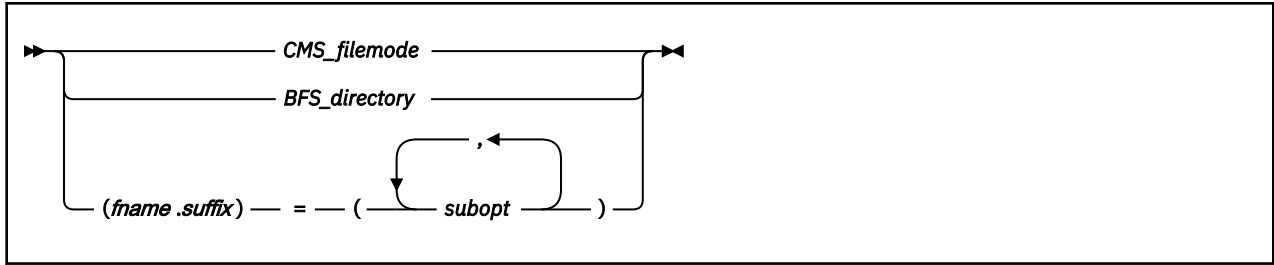


The LSEARCH option directs the preprocessor to look for user include files in the specified libraries in the VM/CMS MACLIBs, on the specified minidisks, or in the specified BFS directories. User include files are files associated with the #include "*filename*" format of the #include preprocessor directive. See [“Using Include Files” on page 47](#) for a description of the #include preprocessor directive.

For further information on library search sequences, see [“Search Sequences for Include Files” on page 52](#).

You must use the double slashes (//) to specify non-BFS searches when the OE compiler option is specified. (You may use them regardless of the OE option.)

Parts of the `#include filename` are appended to each LSEARCH *opt* to search for the include file. *opt* has the format:



CMS_filemode

is the file mode where the sequential disk search for the user include file begins.

BFS_directory

is the BFS path name indicating the directory that should be searched for the include file.

(fname.suffix) = (subopt, subopt, ...)

is a specification where:

fname

is the name of the include file or *.

suffix

is the suffix of the include file or *.

subopt

indicates a sub-path to be used in the search for the include files that match the pattern of *fname.suffix* and should appear at least once. The possible values are:

LIB([mac,...])

Each *mac* is a MACLIB name that should be searched in the same order as in the list. The format of the name is either that of a ddname (form DD: *name*) or *fn.ft.fm*, where the *ft* must be MACLIB and the default *fm* is *.

NOLIB

Specifies that all LIB(. . .) previously specified for this pattern should be ignored at this point. For example, (*.h) = (LIB(n1.MACLIB), NOLIB, LIB(n4.MACLIB)) is equivalent to (*.h) = (LIB(n4.MACLIB)).

When the `#include filename` matches the pattern of *fname.suffix*, the search continues according to the *subopts* in the order given. An asterisk (*) in *fname* or *suffix* matches anything. If the file is not found, other searches are attempted according to the remaining options in LSEARCH.

The MACLIBs are searched in the same order for the include file with * (asterisk) matching anything.

If a file mode is also specified using the SEARCH option, the disks specified by the LSEARCH option are searched first. If the user include file is not located on any of the LSEARCH disks, the disks in the SEARCH option are scanned, in the standard CMS search order, for the user include file.

If no disk is specified, the file mode A will be added to the end of the LSEARCH options.

For more information on the search paths, see [“Search Sequences for Include Files”](#) on page 52.

Under CMS, the NOLSEARCH option instructs the preprocessor to perform the standard CMS search for user include files.

Note: If the *filename* in the `#include` directive is in absolute form, searching is not performed. See [“Determining If filename Is In Absolute Form”](#) on page 49 for more details on absolute `#include filename`.

Specifying Byte File System (BFS) Files

When specifying BFS library searches, do not put double slashes at the beginning of the LSEARCH *opt*. Use path names separated by slashes (/) in the LSEARCH *opt* for a BFS library. When the LSEARCH *opt* does not start with double slashes, any single slash in the name indicates a BFS library. If you do not have path separators (/), then setting the OE compiler option on indicates that this is a BFS library; otherwise the library is interpreted to be a CMS library.

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

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

The resulting BFS file name is:

```
/u/mike/myfiles/new/headers.h
```

OBJECT | NOOBJECT



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.

OBJECT(*filename*) places the object module in the specified file. *filename* can be:

- CMS record file
- Single-letter mode to which the object module is stored as *filename* TEXT *fm*
- BFS file
- Fully qualified path name
- Path name relative to the current working directory

If you do not specify a file name for the OBJECT option, the compiler generates a file name as follows:

- For CMS record source files, the listing is created in a file that has the source file name, file type TEXT, and file mode A.
- For BFS source files, the listing is created in a BFS file that has the source file name with a .o extension.

The NOOBJ option can optionally take a *filename* suboption. This file name then becomes the default. If you subsequently use the OBJ option without a *filename* suboption, the compiler uses the file name that you specified in the earlier NOOBJ. For example, the following specifications have the same result:

```
CC HELLO (NOOBJ(/hello.obj) OBJ
CC 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:

```
CC HELLO (NOOBJ(/hello.obj) OBJ(/n1.obj) NOOBJ(/test.obj) OBJ
CC HELLO (OBJ(/test.obj)
```

If you request a listing by using the SOURCE, INLRPT, or LIST option, and you also specify OBJECT, the name of the object module is printed in the listing prolog.

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

OPTFILE | NOOPTFILE



The OPTFILE option directs the compiler to look for compiler options in the file specified.

OPTFILE(*filename*) specifies the name of the options file where your compiler options are defined. *filename* can be a CMS record or BFS file. The compiler opens *filename* as it is specified. If *filename* is not a valid name, or if the file does not exist, the compiler does not issue an error message. For example, specifying:

```
CC cpgma (OPTFILE(myopts))
```

does not cause an error, but file *myopts* is not opened.

Specifying:

```
CC cpgma (OPTFILE(myopts optfile))
```

opens options file *myopts optfile*. Under the OpenExtensions shell, *filename* is a BFS file. If *filename* is not specified, DD:SYSOPTF is used.

The NOOPTF option can optionally take a *filename* suboption. This file name then becomes the default. If NOOPTF(*filename*) is specified and a subsequent OPTF option is used without a *filename* suboption, the file name specified in the previous NOOPTF is used. For example,

```
CC HELLO (NOOPTF(hello.opt) OPTF
```

is equivalent to specifying:

```
CC HELLO (OPTF(hello.opt))
```

The options are specified in a free format with the same syntax as they would have on the command line. Everything specified in the file is taken to be part of a compiler option (except for the continuation character) and unrecognized entries are flagged. Nothing on a line is ignored.

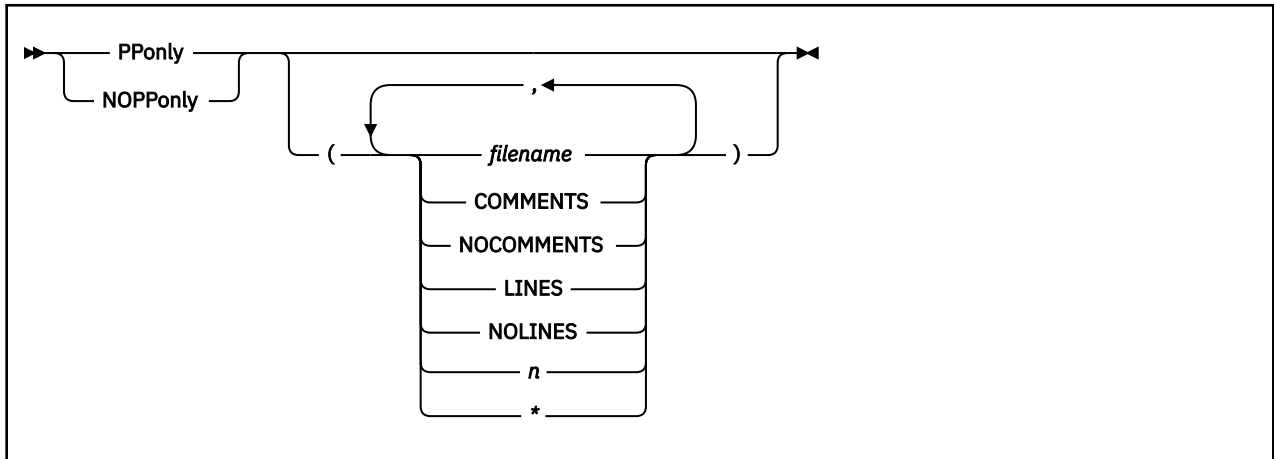
If the record format of the options file is fixed and the record length is greater than 72, columns 73 to the end-of-line are treated as sequence numbers and are ignored.

Notes:

1. You cannot nest the OPTFILE option. If the OPTFILE option is also used in the file specified by another OPTFILE option, it is ignored.
2. If NOOPTFILE is specified after a valid OPTFILE, it does not undo the effect of the previous OPTFILE.
3. If the file cannot be opened or cannot be read, **NO** warning message will be issued and the OPTFILE option will be ignored.
4. The options file can be an empty file.

The OPTFILE option is added to the options section of the compiler-generated listing file.

PPONLY | NOPPONLY



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. The suboptions are:

filename

is the file name for the preprocessed output file. *filename* can be a CMS record or BFS file. If a *filename* is not specified for the PPOONLY option, the compiler writes the preprocessed output as follows:

- For CMS record source files, the preprocessed output is written to a file that has the source file name and file type EXPAND.
- For BFS source files, the preprocessed output is written to a BFS file that has the source file name with a .i extension.

NOCOMMENTS

COMMENTS

specifies whether comments should be preserved in the preprocessed output. The default is NOCOMMENTS.

NOLINES

LINES

specifies whether #line directives should be issued at include file boundaries, block boundaries, and where there are more than 3 blank lines. The default is NOLINES.

n

is an integer between 2 and 32760 inclusive that specifies the column number at which all lines are folded.

specifies that all lines are folded at the maximum record length of 32760. Otherwise, all lines are folded to fit in 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 NOPPONLY, all the suboptions are combined and used for the last occurrence of the option. For example, the following three specifications have the same result:

```
CC HELLO (NOPPONLY(/aa.exp) PPOONLY(LINES) PPOONLY(NOLINES)
CC HELLO (PPOONLY(/aa.exp,LINES,NOLINES)
CC 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")
```

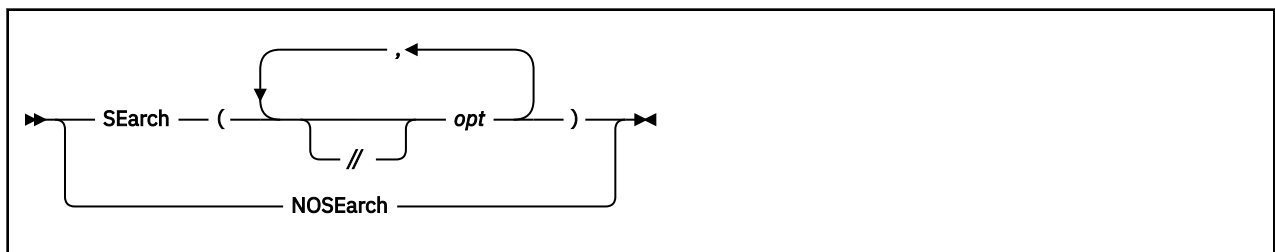
In the above, `??=` is a trigraph representation of the `#` character.

The code page in the pragma is the code set that is specified in the LOCALE option. For more information on locales, see the *z/OS: XL C/C++ Programming Guide* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147315/\\$file/cbcpx01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147315/$file/cbcpx01_v2r5.pdf)).

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.

SEARCH | NOSEARCH



The SEARCH option directs the preprocessor to look for system include files in the specified libraries in the VM/CMS MACLIBs, on the specified minidisks, or in the specified BFS directories. System include files are those files associated with the `#include <filename>` format of the `#include` preprocessor directive. See [“Using Include Files” on page 47](#) for a description of the `#include` preprocessor directive.

For further information on library search sequences, see [“Search Sequences for Include Files” on page 52](#).

The suboptions for the SEARCH option are identical to those for the LSEARCH option, as described in [“LSEARCH | NOLSEARCH” on page 27](#).

Any NOSEARCH option cancels all previous SEARCH specifications and any SEARCH options following it will be used. When several SEARCH compiler options are specified, all the libraries in these SEARCH options are used to find the user include files.

The NOSEARCH option instructs the preprocessor to perform the standard CMS search for system include files.

Note: If the file name in the `#include` directive is in absolute form, searching is not performed. See [“Determining If filename Is In Absolute Form” on page 49](#) for more details on absolute `#include filename`.

SOURCE | NOSOURCE



The SOURCE option generates a listing that shows the original source input statements plus any diagnostic messages.

SOURCE(*filename*) places the listing in the specified file. *filename* can be a CMS record or BFS file. If you do not specify a file name for the SOURCE option, the compiler constructs the file name as follows:

- For CMS record source files, the listing is created in a file that has the source file name, file type LISTING, and file mode A.
- For BFS source files, the listing is created in a BFS file that has the source file name with a .lst extension.

The NOSOURCE option can optionally take a *filename* suboption. This file name then becomes the default. If you subsequently use the SOURCE option without a *filename* suboption, the compiler uses the file name that you specified in the earlier NOSOURCE. For example, the following specifications have the same result:

```
CC HELLO (NOSO(/hello.lst) SO
CC HELLO (SO(/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:

```
CC HELLO (NOSO(/hello.lst) SO(/n1.lst) NOSO(/test.lst) SO
CC HELLO (SO(/test.lst)
```

If you specify file names with the SOURCE, LIST or INLRPT options, all the listing sections are combined into the last file name specified.

Using the C Compiler Listing

See the corresponding section in the *z/OS: XL C/C++ User's Guide* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147307/\\$file/cbcux01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147307/$file/cbcux01_v2r5.pdf)).

Using the C++ Compiler Listing

See the corresponding section in the *z/OS: XL C/C++ User's Guide* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147307/\\$file/cbcux01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147307/$file/cbcux01_v2r5.pdf)).

Chapter 5. Compiler Options under OpenExtensions

This chapter discusses the compiler options that you can use when compiling under OpenExtensions. For information about compiler options under VM/CMS, see [Chapter 4, “Compiler Options,”](#) on page 19.

Specifying Compiler Options Using c89/cxx

The c89 and cxx utilities are the OpenExtensions interface to the XL C/C++ compiler. When you issue c89 or cxx for a C or C++ application program, the utility passes information about the application program and the compiler options to the XL C/C++ compiler for processing.

The c89 and cxx utilities select specific values for most compiler options. You can cause them to change the settings of those C/C++ compiler options that they have corresponding flags (options) for. If you want to pass other C/C++ compiler options to the XL C/C++ compiler, use the -W option. If you used the options -E, -g, -s, or -O you cannot override the compiler options forced by the c89 or cxx utility. This holds true even when using the -W option to explicitly pass XL C/C++ compiler options.

XL C/C++ compiler options are summarized in [“Compiler Option Defaults”](#) on page 21 and described in detail in [“Descriptions of Compiler Options”](#) on page 21. For more information on OpenExtensions commands, see the [z/VM: OpenExtensions Commands Reference](#).

c89/cxx Default Compiler Settings

c89 overrides the default settings for the XL C/C++ compiler options. The overridden defaults are:

- `DEFINE(errno=(*_errno()))`
- `DEFINE(_POSIX_SOURCE=1)`
- `DEFINE(_POSIX1_SOURCE=2)`
- `DEFINE(_POSIX_C_SOURCE=2)`
- `LANGVLV(ANSI)`
- `OE`
- `RENT`

c89 Selectable Compiler Settings

Format

```
c89 [-cgsEOV]
    [-D name[=value]] .... [-U name]...
    [-W c,opt[,opt]... ]...
    [-o outfile]
    [-I directory]... [-L directory]...
    [file.c]... [file.a]... [file.o]...
    [-l libname]...
```

```
cxx [-+cgsEOV]
    [-D name[=value]] .... [-U name]...
    [-W c,opt[,opt]... ]...
    [-o outfile]
    [-I directory]... [-L directory]...
    [file.c]... [file.a]... [file.o]...
    [-l libname]...
```

Description

c89/cxx Option

Compiler Option

-+ (cxx only)

All source files are to be recognized as C++ source files.

-c

Compilation only

-D

Define preprocessor macros.

-E

Run the C preprocessor only (do not generate an object file or run the linkage editor) and copy output source to stdout.

-g

Generate symbolic information with the compiled object. The c89 -s option, the default, indicates that no debugging information or line number tables be generated.

-I

Specify where to search for C include files. The search path is supplied as a value on the option. For example:

```
-I /usr/hankvp/bin/headers
```

-L

Specify where to search for archive files specified by the -I option.

-O

Set an optimization level and place functions at their point of call.

o

Write the executable file to outfile.

-s

Do not generate symbolic information with the compiled object.

-U

Undefine preprocessor macros (including c89 default macro definitions).

-V

Write a "verbose" listing to stdout. Listings are generated by the compiler and binder.

The information in the compiler listing corresponds to those compiler options set by the c89 -V option. For a complete description of the effect of each compiler option, see [“Descriptions of Compiler Options”](#) on page 21.

-W

Pass compiler or module build options. Phase 0 or c specifies the compile phase, and phase b specifies the module build phase. The module build phase is binder processing to create the module file. To pass options to the BIND command, the module build option must be b. For example, to pass the LANTLR option to the compiler, specify:

```
c89 -W 0,langlvl(extended)
```

and to write the binder map to stdout, specify:

```
c89 -W b,b,map file.c
```

For a detailed description of the c89 options, see the [z/VM: OpenExtensions Commands Reference](#).

c89 uses the following compiler option settings if the c89 option listed is specified (more than one compiler option may be specified by a particular c89 option):

Table 2. c89 Option and Corresponding Compiler Option	
c89 Option	Compiler Options(s)
D <i>value</i>	DEFINE(<i>value</i>)
-E	PPONLY(1024)
-g	TEST(ALL) GONUMBER
-I <i>value</i>	SEARCH(<i>value</i>)
-O	INLINE(NOAUTO,NOREPORT,250,1000) NOMEMORY OPTIMIZE(2)
-S	NOGONUMBER NOTEST
-V	AGGREGATE CHECKOUT(ALL,NOEXTERN,NOPPCHECK,NOPPTRACE) FLAG(I) LIST OFFSET SHOWINC SOURCE XREF

Notes:

1. The c89 -U *value* option causes c89 not to specify a corresponding DEFINE(*value*) compiler option.
2. Use of the c89 -V option *may* result in a return code of 4 from the compile step when the return code should be 0. This is because of the specification of the CHECKOUT option. Also, the specification of FLAG(I) may cause additional informational messages to be directed to stderr.

Feature Test Macros

For information on how to use the feature test macros, see the [XL C/C++ for z/VM: Runtime Library Reference](#).

Chapter 6. Runtime Options

This chapter describes runtime options and the `#pragma runopts` preprocessor directives available to you with XL C/C++ and Language Environment. For information on runtime options under OpenExtensions, refer to “Specifying Runtime Options under OpenExtensions” on page 70.

Specifying Runtime Options

To allow your application to recognize runtime 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 runtime options as follows:

- On the command line when you invoke your program under VM/CMS
- At compile time, on a `#pragma runopts` directive in your main program

If EXECOPS is in effect, use a slash (/) to separate runtime options from arguments that you pass to the application. For example:

```
PGMX STORAGE(FE,FE,FE)/PARM1 PARM2 PARM3
```

If EXECOPS is in effect, Language Environment interprets the character string that precedes the slash as runtime 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. Runtime 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 runtime options for applications that are compiled with the Language Environment, see the *z/VM: Language Environment User's Guide* and the *z/OS: Language Environment Programming Reference* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380683/\\$file/ceea300_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380683/$file/ceea300_v2r5.pdf)).

Runtime Options Using Language Environment

You can use the `#pragma runopts` preprocessor directive to specify Language Environment runtime options, including ARGPARSE, ENV, PLIST, REDIR, and EXECOPS, which have matching compiler options. If you specify the compiler option, it has precedence over the `#pragma runopts` directive.

When the runtime option EXECOPS is in effect, you can specify runtime options at execution time, as previously described. These options override runtime options that you compiled into the program by using the `#pragma runopts` directive.

The `#pragma runopts` directive can appear in any file: main, include, or source. You can specify multiple runtime 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()`.

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 bind multiple compilation units that specify `#pragma runopts`, the binder takes only the first CSECT, thereby ignoring your other option statements. Therefore, you should always specify your `#pragma runopts` directive in the same source file that contains the function `main()`.

For more information on the `#pragma runopts` preprocessor directive, see the *z/OS: XL C/C++ Language Reference* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147308/\\$file/cbclx01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147308/$file/cbclx01_v2r5.pdf)).

Chapter 7. Compiling a C/C++ Program

This chapter describes how to compile your program using the XL C/C++ compiler and Language Environment under VM/CMS. For information on compiling your program under OpenExtensions, refer to Chapter 9, “Compiling a C/C++ Program under OpenExtensions,” on page 65.

The XL C/C++ compiler analyzes the C/C++ source program and translates the source code into machine instructions known as *object code*. You must have access to the Language Environment C/C++ runtime library, because the compiler calls functions in the library to compile the code.

Invoking the XL C/C++ Compiler

When you invoke the XL C/C++ compiler, the operating system automatically tries to locate and execute the compiler. The location of the compiler is determined by the system programmer who installed the product. The compiler may be in a nucleus extension, in a *discontiguous saved segment (DCSS)*, or on a minidisk. In either instance, you only need to ensure that you have access to the C compiler version that you want to use.

The XL C/C++ compiler can be invoked under VM/CMS using the IBM supplied CC EXEC.

The XL C/C++ compiler compiles source code using the Language Environment. You must ensure that the load libraries that contain XL C/C++, Language Environment, and VM/CMS library routines are available. The runtime libraries are needed for compilation, because the compiler calls functions from the libraries. The GLOBAL command is used to link to the libraries. The libraries may be in a nucleus extension, a DCSS, or in the GLOBAL LOADLIB list. For more information on how to make libraries available for execution, refer to “Making the Runtime Libraries Available for Execution” on page 62. The following examples assume that the default names (which can be changed by the system programmer during installation) are used.

GLOBAL Command for Using the Language Environment Library

The GLOBAL commands to make the library available to compile, bind, and run a program are as follows:

- To run the compiler:

```
GLOBAL LOADLIB USERLIB SCEERUN
```

- To bind your C object code:

```
GLOBAL TXTLIB USERLIB SCEELKD
```

- To bind your C++ object code:

```
GLOBAL TXTLIB USERLIB SCEELKD SCEECPP
```

- To run your C or C++ module:

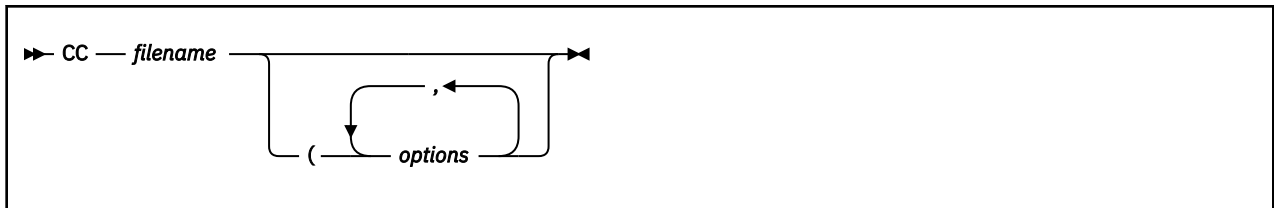
```
GLOBAL LOADLIB USERLIB SCEERUN
```

where USERLIB represents any user load or text libraries.

Note: The SCEECPP text library is part of Language Environment and contains the base C++ link-edit routines.

Syntax of the CC EXEC

The syntax of the CC EXEC is:



filename

is the name of the source file to be compiled. The source file can be a CMS record or BFS file.

options

specifies the compiler options to use during compilation. If no compiler options are specified, the default settings are used.

For a description of the compiler options that you can specify when invoking the CC EXEC, refer to [“Descriptions of Compiler Options”](#) on page 21.

Specifying the Input File

Input for the compiler consists of:

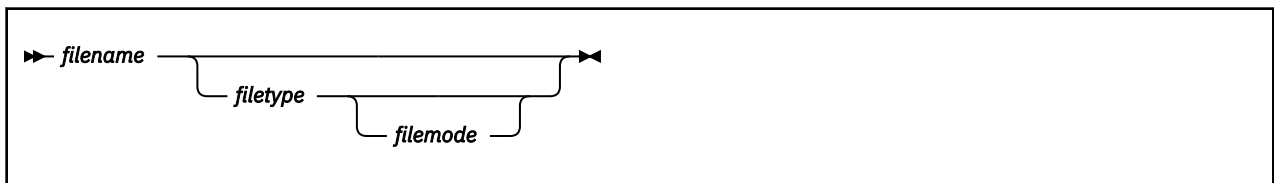
- Your C/C++ source program
- C/C++ standard header files
- Your header files

The primary input to the compiler is the first argument passed to the CC exec. Your C/C++ source may be in a CMS record or BFS file. The secondary input to the compiler consists of files identified by `#include` preprocessor directives in the input. For more information on `#include` files, see [“Using Include Files”](#) on page 47.

The output that the compiler generates is based on the primary source file input.

CMS Record Files

To specify a CMS record file as your primary source file, use the following syntax:



You must always specify the source file name following the CC keyword. If the file type is not C, the file type must also be specified on the CC EXEC. If you do not specify the file mode, the currently accessed minidisks are searched in the standard VM/CMS search order. The file that is compiled is the first one encountered in the disk search. For example, if you have a file called `TWICE C` on both your B and Y minidisks, and the Y minidisk is not accessed as an extension of the A disk, `TWICE C B` is compiled if you do not specify the file mode. Note also that if you specify the file mode, you must also specify the file type.

KNOWN:

- The file name is SALARY.
- The file type is C.
- The file mode is A.

USE THE FOLLOWING COMMAND:

```
CC SALARY
```

Result: The object module generated will have file name SALARY, a file type of TEXT, and file mode A.

Figure 16. Specifying a CMS Record Input File under VM/CMS (Example 1)

```

KNOWN:   - The file name is INCOME
          - The file type is NET
          - The file mode is Y

USE THE FOLLOWING COMMAND:
CC INCOME NET Y
Result:  The object module generated will have file name INCOME,
         a file type of NET, and file mode A.

```

Figure 17. Specifying a CMS Record Input File under VM/CMS (Example 2)

BFS Files

You can also use the CC EXEC to compile source that is in BFS files. To specify a BFS file as your primary source file, use the following syntax:



- ./**
specifies the current directory.
- ../**
specifies the previous directory.
- /**
specifies the beginning of an absolute path name.

pathname
specifies all directories leading to the file.

filename
is the name of the source file.

When you use the CC EXEC, you must use unambiguous BFS source file names. For example, the compiler treats the following input files as BFS files:

```

CC ./test/hello.c
CC /u/david/test/hello.c
CC test/hello.c
CC ///hello.c
CC ../test/hello.c

```

If *filename* is not in the *pathname* format with single slashes, the compiler treats the file as non-BFS input. The following input files are treated as non-BFS files:

```

CC hello.c
CC //hello.c

```

For complete information on working with BFS files, see the [z/VM: OpenExtensions User's Guide](#).

```
KNOWN:   - The current working directory is /u/proga.
          - The file name is myprog.c.

USE THE FOLLOWING COMMAND:
CC ./myprog.c
Result:  The object module generated will be in the current
working directory and have a file name
myprog.o.
```

Figure 18. Specifying a BFS Input File under VM/CMS (Example 1)

```
KNOWN:   - The file name is myprog.c in directory
          /u/boris/progs.
          - The current working directory is /u/proga.

USE THE FOLLOWING COMMAND:
CC /u/boris/progs/myprog.c
Result:  The object module generated will be myprog.o
in the current working directory /u/proga.
```

Figure 19. Specifying a BFS Input File under VM/CMS (Example 2)

For information on compiling programs under OpenExtensions, see [Chapter 9, “Compiling a C/C++ Program under OpenExtensions,”](#) on page 65.

Specifying Compiler Options

There are many compiler options that you can specify when you compile using the CC EXEC. They are described in [“Descriptions of Compiler Options”](#) on page 21.

The following examples show you how to override the default options when compiling under VM/CMS. When you specify the options, separate them by at least one blank; you can have any number of extra blanks. The order is unimportant. If two contradictory options are specified, the last option specified is accepted and the first ignored.

CMS Record File Examples

```
KNOWN:   - The file being compiled is FINANCE EXPAND A.
          - A listing of the source file is required.

USE THE FOLLOWING COMMAND:
CC FINANCE EXPAND (SOURCE

Result:  A listing with the same file name as your source and
a file type of LISTING is generated.
When an error occurs, the compiler sends an error
message to your terminal screen and to your source
```

Figure 20. Specifying Compiler Options under VM/CMS (Example 1)

```

KNOWN:  - The file being compiled is BASEBALL C X.
         - The following disks are to be scanned for
           user include files:
           - V,W,X,Y, and Z (using the LSEARCH option)
           - S,T, and U (using the SEARCH option)
         - The following disks are to be scanned for
           system include files:
           - S,T,U,V,W,X,Y, and Z (using the SEARCH option)

USE THE FOLLOWING COMMAND:
  CC BASEBALL C X (LSEARCH(V) SEARCH(S)

Result:  If the user include file is not found
         on one of the disks specified by the LSEARCH option,
         then the disks specified by the SEARCH option are
         searched for the user include file.

         The disks S through Z are scanned in the standard CMS search
         order for the system include file(s).

```

Figure 21. Specifying Compiler Options under VM/CMS (Example 2)

BFS File Example

```

KNOWN:  - The file being compiled is myprog.c in the
         current working directory /u/progs.
         - A listing of the source file is required.

USE THE FOLLOWING COMMAND:
  CC ./myprog.c (source

Result:  A listing with the same file name as your source and
         a file extension of lst is generated in your
         current working directory.

```

Figure 22. Specifying Compiler Options for BFS Files

Creating Input Source Files

For CMS record files, the C/C++ compiler accepts both F-format and V-format records. The primary and secondary input can have different formats. For information on mixing formats, see the #pragma sections, margins and sequence in the *z/OS: XL C/C++ Language Reference* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147308/\\$file/cbclx01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147308/$file/cbclx01_v2r5.pdf)).

To assist you in migrating existing applications from other operating systems to VM/CMS, file name conversions (described in the following sections) are performed automatically by XL C/C++ These conversions affect file names specified on #include preprocessor directives, and in file I/O library functions such as fopen. See the *z/OS: XL C/C++ Language Reference* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147308/\\$file/cbclx01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147308/$file/cbclx01_v2r5.pdf)) for general information on the #include directive and the available I/O functions.

For complete information on working with BFS files, see the *z/VM: OpenExtensions User's Guide*.

Specifying Output Files

The compiler can generate the following kinds of output files:

- Object file
- Listing file
- Preprocessor output
- Events file

- Error message file

When you compile source, you can specify the resultant output file type by using the following compiler options:

Output File Type	Compiler Option
Object File	OBJECT(<i>filename</i>)
Listing File	INLRPT(<i>filename</i>)
	LIST(<i>filename</i>)
	SOURCE(<i>filename</i>)
Preprocessor Output	PPONLY(<i>filename</i>)
Events File	EVENTS(<i>filename</i>)

When you specify any of these compiler options and do not use suboptions to identify the output file names, the compiler generates default output file names based on the type of source file being compiled. Output file names are the same as the source file names. The default output CMS file types and BFS suffixes that the compiler uses are summarized in [Table 3 on page 46](#).

Table 3. Default CMS File Types and BFS Suffixes for Output Files		
Output File	CMS filename Type	BFS filename Suffix
Object File	TEXT	o
Listing File	LISTING	lst
Preprocessor Output	EXPAND	i
Events File	SYSEVENT	err

If you compile source in a CMS record file without specifying output file names in the compiler options, output files are generated on the A disk with the file type shown in [Table 3 on page 46](#). For example,

```
cc hello c
```

generates object file

```
hello text
```

If you compile source in a BFS file without specifying output file names in the compiler options, output files are generated in the current directory with the suffix shown in [Table 3 on page 46](#). For example,

```
cc /user/fred/hello.c
```

generates object file

```
./hello.o
```

Events file output is generated using the same file name as the source file and stored on the user's A disk using a file type of SYSEVENT.

Error messages are redirected to `stderr` if the `TERMINAL` option is in effect. Error messages can be redirected to a file using the redirection technique, for example:

```
CC A (>ERROR.LOG
```

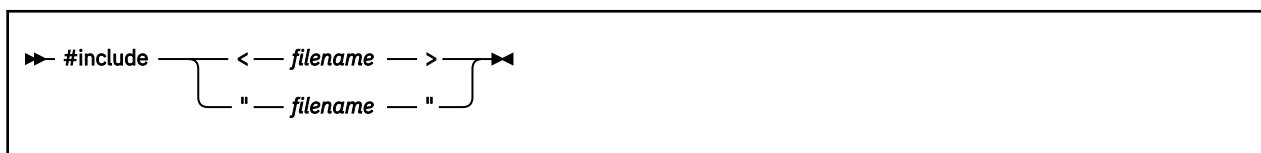
Valid Input/Output File Types

Depending on the type of file that is used as primary input, certain output file types are allowed. The following table describes these input and output file combinations:

Input Source File	Output File Specified Not In filename Format, for example A B C	Output specified as BFS file, for example a/b/c.o
CMS Native File, for example A B	<ol style="list-style-type: none"> 1. If output file exists, overwrites it 2. If output file does not exist, creates the file 	<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.
BFS file, for example /a/b/d.c	<ol style="list-style-type: none"> 1. If file exists as a CMS record file, overwrites it 2. If file does not exist, creates output file 	<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.

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. A description of the `#include` directive is provided in the *z/OS: XL C/C++ Language Reference* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147308/\\$file/cbclx01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147308/$file/cbclx01_v2r5.pdf)). Its syntax is:



Note: On previous compilers, the double slash at the beginning of *filename* indicated a CMS file. This is not so for XL C/C++. If you specify it, the CMS minidisks will **NOT** be searched.

Angle brackets (< >) are used to specify system include files, and double quotation marks (" ") are used to specify user include files.

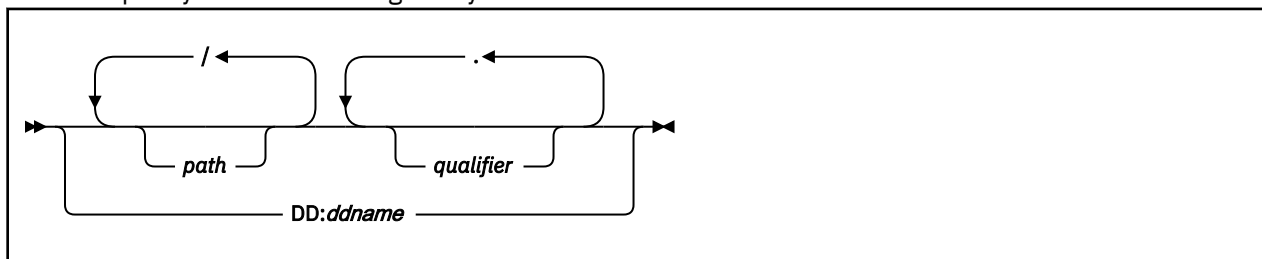
When you use the `#include` directive, you must be aware of:

1. The file-naming conversions performed by XL C/C++ See [“Specifying #include File Names” on page 48](#) for more information on file name conversions performed by XL C/C++.
2. The search order used by XL C/C++ to locate the file (known as the *library search sequence*). See [“Search Sequences for Include Files” on page 52](#) for more information on the library search sequence.
3. The area of the input record containing sequence numbers when including files with different record formats. See the *z/OS: XL C/C++ Language Reference* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147308/\\$file/cbclx01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147308/$file/cbclx01_v2r5.pdf)) 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 and user include files. For more information on these options, see [“LSEARCH | NOSEARCH”](#) on page 27 and [“SEARCH | NOSEARCH”](#) on page 32.

You can specify a file name using the syntax:



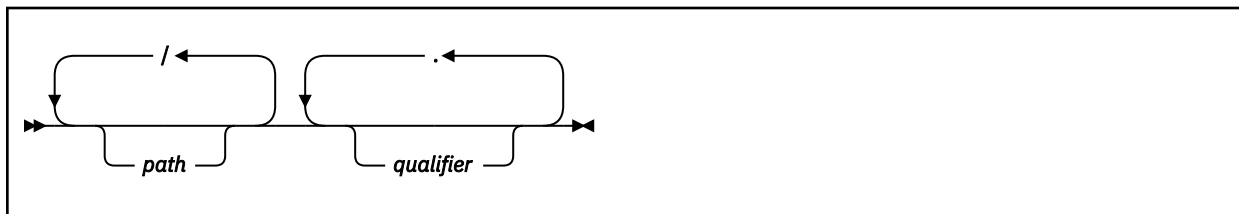
Notes:

1. Absolute CMS file names contain a file mode or are ddnames.
2. Absolute BFS file names begin with a leading slash (`/`) as the first character in *filename*.

For more information on absolute file names, see [“Determining If filename Is In Absolute Form”](#) on page 49.

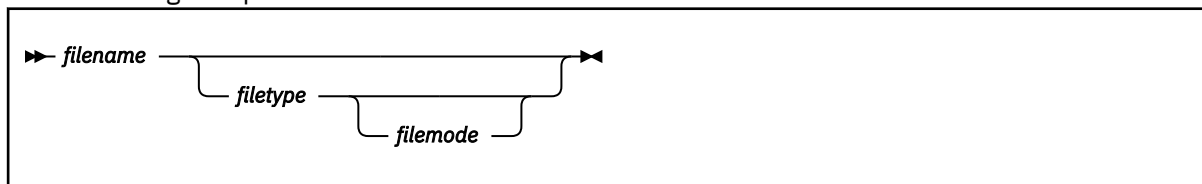
When the compiler performs a library search, *filename* may be treated as a BFS file name or a CMS file name. This depends on whether a CMS library or a BFS directory is being searched. If *filename* is treated as a BFS file name, then no conversions are performed on *filename*. If, on the other hand, *filename* is to be treated as a CMS file name, then the following conversions are performed:

- For the first format:



The compiler performs name conversions in the following order:

1. All periods (`.`) are replaced with blank spaces.
2. Characters up to and including the rightmost slash (`/`) (if any slashes are present) are deleted from the file specification.
3. The remaining file specification must be of the form:



If there are more than three qualifiers, only the first three are used as the file name, file type and file mode, beginning with the leftmost qualifier. The remaining ones are ignored. If you specify the CHECKOUT (PPTRACE) compiler option, a message states what include files the preprocessor is looking for.

4. All file names and file types are truncated to a maximum of eight characters. File modes are truncated to two characters.
5. The file mode must be a valid CMS file mode, or an asterisk (`*`).
6. If a file mode is not specified, the currently accessed disks are searched in the order described under [“Search Sequences for Include Files”](#) on page 52.

7. If a file type is not specified, the default is H.

- For the second format:

➡ DD:ddname →

1. DD: and *ddname* are uppercased and truncated to eight characters.

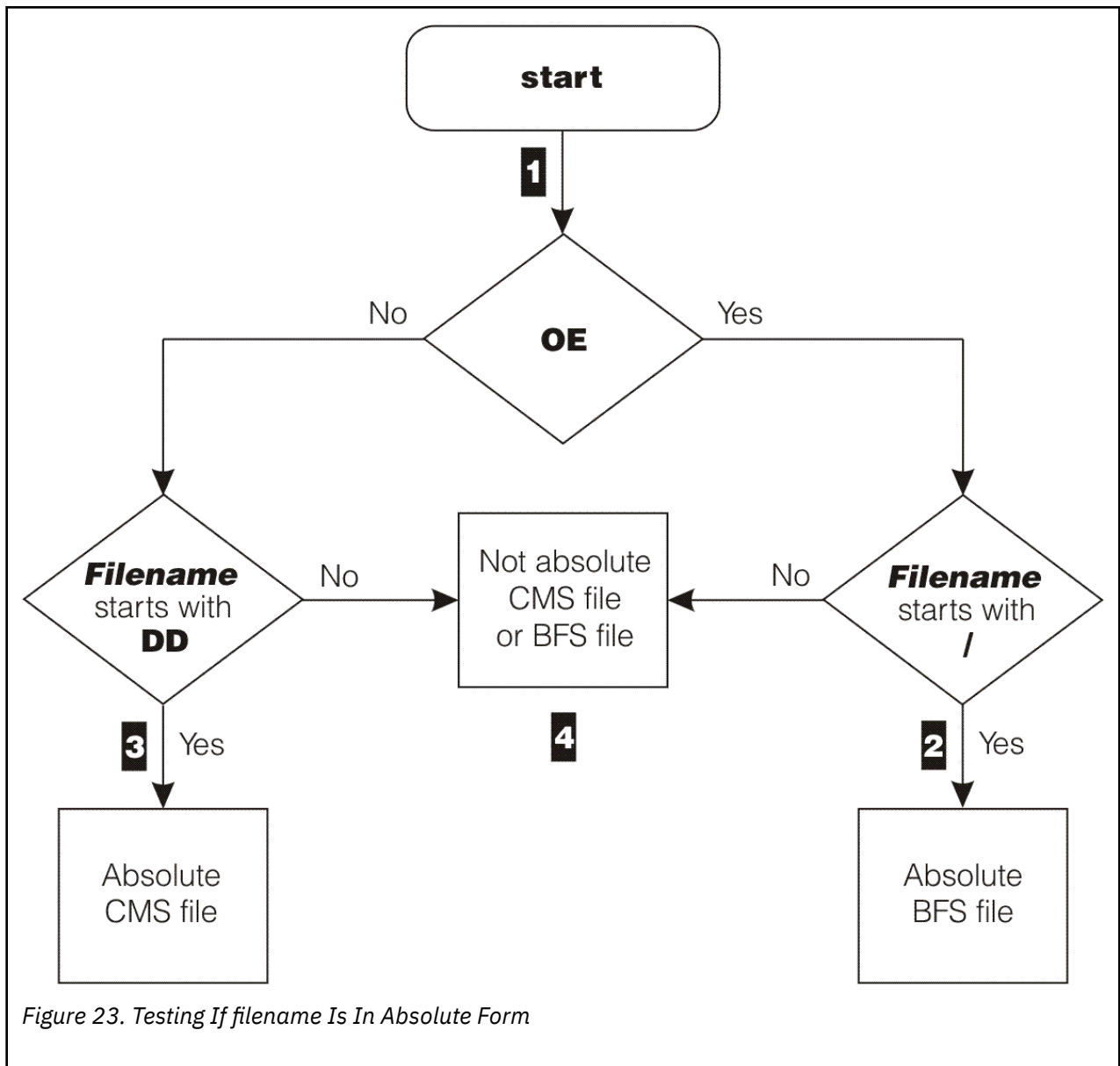
2. Invalid characters are not converted to at signs(@, hex 7c).

Table 5 on page 49 gives the original format of the file name as specified on a `#include` directive in a source file, and the actual file name used when XL C/C++ attempts to locate and open the file.

Table 5. Include Directive and Resulting File Names	
#include Directive	Resulting File Name
#include <stdio.h>	STDIO H
#include <Shoe/Sale/Fall.D>	FALL D
#include "cprog"	CPROG H
#include "utility.h.a"	UTILITY H A Note: If the file is not found on disk A, no further search is made.
#include "DD:MYSYS"	file name on MYSYS DD Note: The file name associated with the ddname MYSYS will be used.
#include <DD:PLANLIB>	file name on PLANLIB DD Note: The file name associated with the ddname PLANLIB will be used.

Determining If *filename* Is In Absolute Form

The compiler determines if the *filename* specified in `#include` is in absolute form as shown in [Figure 23](#) on page 50.



- 1** The compiler first checks whether the OE option is specified.
- 2** If OE is specified, and *filename* starts with a slash (/), then *filename* is in absolute form. The compiler opens the file directly as a BFS file.
- 3** If OE is not specified, and the *ddname* format of the `#include` directive has been used, the compiler uses the file associated with the given *ddname* and directly opens the file. The *ddname* can point to a BFS file. The libraries specified in the LSEARCH or SEARCH options are ignored.
- 4** If none of the above conditions are true, then the *filename* is not in absolute format and each opt in the LSEARCH or SEARCH compiler option determines if the file is a BFS or CMS native file.

For example:

```
Options specified:
  OE
Include Directive:
```

#include "apath/afile.h"	NOT absolute, BFS/CMS (no starting slash)
#include "/apath/afile.h"	absolute BFS, (starts with 1 slash)
#include "a.b.c"	NOT absolute, BFS/CMS (no starting slash)
#include "DD:SYSLIB"	NOT absolute, BFS/CMS (no starting slash)
#include "a.b"	NOT absolute, BFS/CMS (no starting slash)

Using LSEARCH and SEARCH

When the filename in a `#include` directive is not in absolute format, 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 a BFS or CMS path as shown in [Figure 24 on page 51](#).

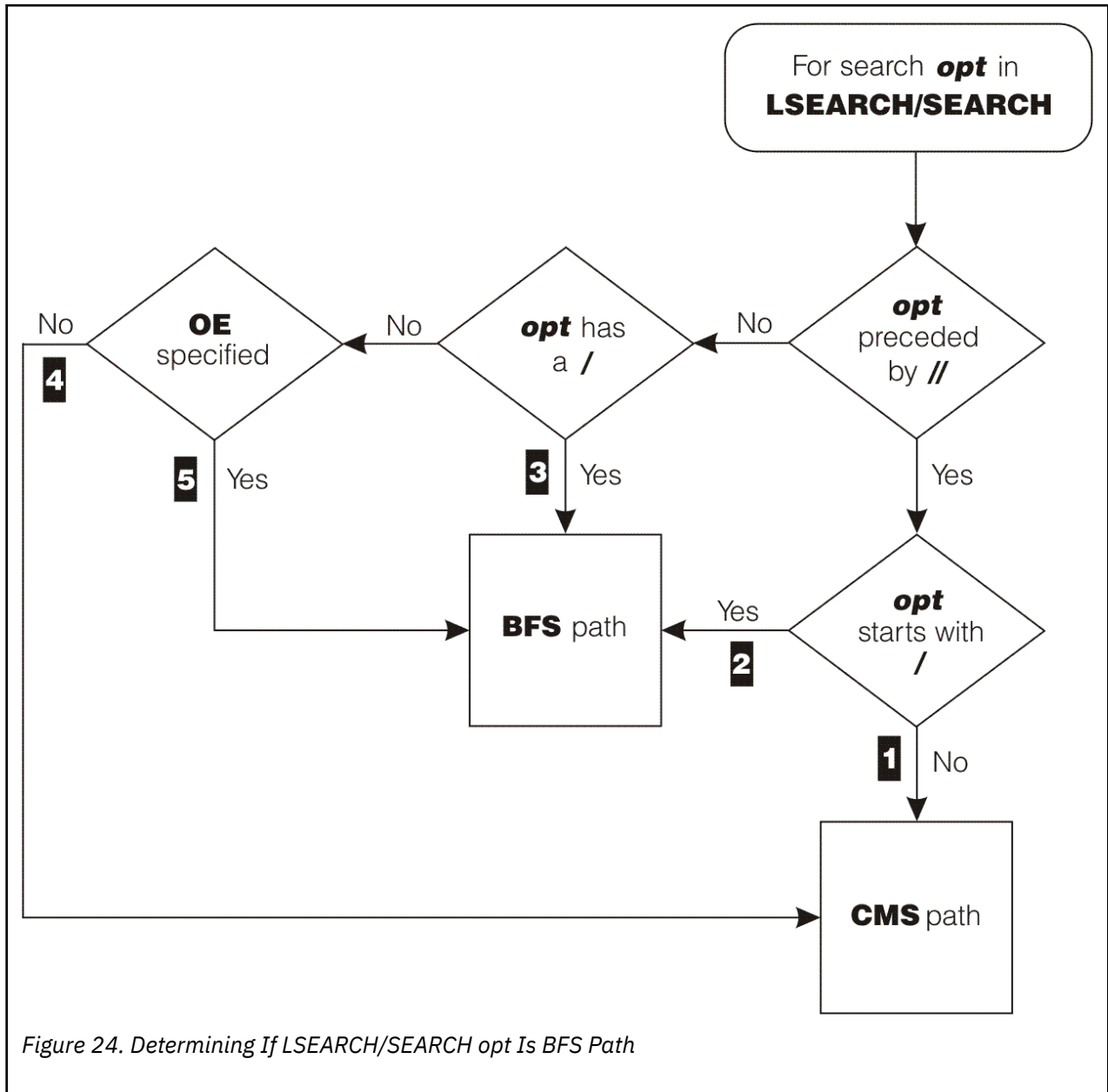


Figure 24. Determining If LSEARCH/SEARCH opt Is BFS Path

1

If *opt* is preceded by double slashes (`//`) and *opt* does not start with a slash (`/`), then this path is a CMS path.

2

If *opt* is preceded by double slashes (`//`) and *opt* starts with a slash (`/`), then this path is a BFS path.

3

If *opt* is **not** preceded by double slashes (`//`) and *opt* contains a slash (`/`), then this path is a BFS path.

4

If *opt* is **not** preceded by double slashes (//) and does not contain a slash (/) and NOOE is specified, then this path is a non-BFS path.

For example:

Syntax	Path
SEARCH(./PATH)	Explicit BFS path
OE SEARCH(PATH)	Treated as a BFS path
NOOE SEARCH(PATH)	Treated as a non-BFS path
OE SEARCH(//PATH)	Explicit non-BFS path.

When combining the library with the *filename* specified on the #include directive, it is the form of the library that determines how the include filename is to be transformed. For example:

Options specified:

```
NOOE LSEARCH(Z, /u/myincs)
```

Include Directive:

```
#include "apath/afile.h"
```

Resulting fully qualified include names:

1. AFILE H Z (Z is non-BFS so filename is treated as non-BFS)
2. /u/myincs/apath/afile.h (/u/myincs is BFS so filename is treated as BFS)

The order of specification of the options on the LSEARCH/SEARCH option is the order that is searched.

If no disk is specified, the file mode A will be added to the end of the LSEARCH/SEARCH option.

See “LSEARCH | NOSEARCH” on page 27 and “SEARCH | NOSEARCH” on page 32 for more information on these compiler options.

Search Sequences for Include Files

With XL C/C++, you can specify a search path for locating secondary input files. To specify the search path, you use the LSEARCH and SEARCH compiler options. For details on these compiler options, refer to “LSEARCH | NOSEARCH” on page 27 and “SEARCH | NOSEARCH” on page 32.

You can search any currently accessed disk or any MACLIB or BFS directory in any order. By default, if there is no LSEARCH or SEARCH option specified, the disks will be searched in the standard VM/CMS order.

If a user include file is not found on the disks or in the MACLIBs or BFS directories specified by the LSEARCH option, the disks and MACLIBs named in the SEARCH option are also scanned in the standard VM/CMS order. Only the disks and MACLIBs specified in the SEARCH option are searched for system include files.

With the NOOE option in effect

Search Sequences for include files are used when the include file is **not** in absolute form. See “Determining If filename Is In Absolute Form” on page 49 for a description of the absolute form of an include file.

If the include filename is not absolute, then the compiler performs the library search as follows:

The search order for **system** include files is:

1. The search order as specified on the SEARCH option, if any
2. The standard CMS disk search, as long as no file mode was specified on the SEARCH option.

The search order for **user** include files is:

1. The search order as specified on the LSEARCH option, if any
2. The standard CMS disk search, as long as no file mode was specified on the LSEARCH option.
3. The search order for system include files.

For example:

```
CC ECONOMY (LSEARCH(X,(*.H)=(LIB(ALPHA.MACLIB))) SEARCH(V)
```

would result in the following search:

Order of Search	For System Include Files	For User Include Files
First	V	X
Second	W	Y
Third	X	Z
Fourth	Y	ALPHA MACLIB (for *.H files)
Fifth	Z	V
Sixth		W

With the OE option in effect

Search Sequences for include files are used when the include file is not in absolute form. See [“Determining If filename Is In Absolute Form” on page 49](#) for a description of the absolute form of an include file.

If the include filename is not absolute then 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 standard CMS disk search, as long as no file mode was specified on the SEARCH option.
- For **user** include files:
 1. If the current source file is a BFS file, the directory of the current source file
 2. The search order as specified on the LSEARCH option, if any
 3. The standard CMS disk search, as long as no file mode was specified on the LSEARCH option
 4. The search order for system include files.

For example, given a file `/r/you/cproc` containing the following `#include` directives:

```
#include "/u/usr/header1.h"  
#include "common/header2.h"  
#include <header3.h>
```

And the following options:

```
OE(/u/crossi/myincs/cproc)  
SEARCH(/V, "/new/inc1", "/new/inc2")  
LSEARCH("/c/c1", "/c/c2")
```

Then the include files would be searched as follows:

Table 6. Examples of Search Order for OpenExtensions	
#include Directive Filename	Files in Search Order
Example 1. This is an absolute path name, so no search is performed.	
#include "/u/usr/header1.h"	1. /u/usr/header1.h
Example 2. This is an OpenExtensions 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/header2.h"	1. /u/crossi/myincs/common/header2.h 2. /c/c1/common/header2.h 3. /c/c2/common/header2.h 4. HEADER2 H * 5. HEADER2 H V 6. HEADER2 H W 7. HEADER2 H X 8. HEADER2 H Y 9. HEADER2 H Z 10. /new/inc1/common/header2.h 11. /new/inc2/common/header2.h
Example 3. This is an OpenExtensions system include file with a relative path name. The search follows the order of suboptions of the SEARCH option.	
<header3.h>	1. HEADER3 H V 2. HEADER3 H W 3. HEADER3 H X 4. HEADER3 H Y 5. HEADER3 H Z 6. /new/inc1/common/header3.h 7. /new/inc2/common/header3.h

Chapter 8. Binding and Running a C/C++ Program

This chapter gives an overview of how to bind and run C/C++ applications using Language Environment under VM/CMS. If you are using OpenExtensions, see [Chapter 10, “Binding and Running a C/C++ Program under OpenExtensions,”](#) on page 69.

Language Environment provides a common runtime environment for C, COBOL, and PL/I. For detailed instructions on binding and running existing and new C/C++ programs under Language Environment, see the *z/VM: Language Environment User's Guide* and the *z/OS: Language Environment Programming Reference* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380683/\\$file/ceea300_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380683/$file/ceea300_v2r5.pdf)).

The following examples describe how to bind and run a program under VM/CMS in Language Environment. Use the following series of commands to:

1. Bind the C and/or C++ text files.
2. Make the Language Environment library available.
3. Run the load module.

To bind and run a C program:

```
CMOD MYPROG
GLOBAL LOADLIB SCEERUN
MYPROG
```

Figure 25. CMS Commands to Bind and Run a C Program

To bind and run a C++ program:

```
CMOD MYPROG (C++
GLOBAL LOADLIB SCEERUN
MYPROG
```

Figure 26. CMS Commands to Bind and Run a C++ Program

Note: Information on Language Environment is reproduced here for convenience only. For detailed information on Language Environment, see your Language Environment documentation.

Library Routine Considerations

The Language Environment consists of one component that contains all Language Environment enabled languages, such as C, COBOL, and PL/I.

The Language Environment is *dynamic*. That is, many of the functions available in XL C/C++ are not physically stored as a part of your executable program. Instead, only a small portion of code known as a *stub routine* is actually stored with your executable program, and this results in a smaller executable module size. The stub routines contain code that branches to the dynamically loaded Language Environment routine.

Creating an Executable Program

Compilation using the CC EXEC produces an object module with file type TEXT. Further processing is required to produce an executable module. The simplest way to do this is to use the IBM-supplied CMOD EXEC.

The CMOD EXEC uses either of the following methods to load one or more object modules (file type TEXT) into virtual storage, resolve external references, and create an executable module (file type MODULE) on disk:

- Invoke the Binder.
- Invoke the LOAD and GENMOD CMS commands (and optionally the prelinker).

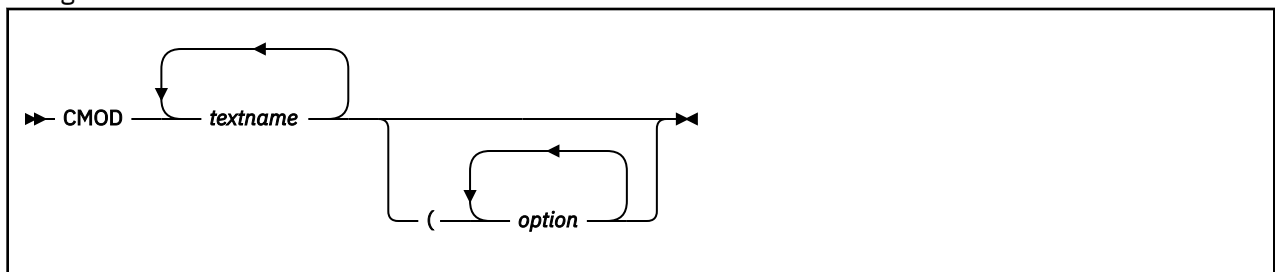
Note: The Prelinker is not supported for use with XL C/C++.

The method used will depend first of all on the options specified on the CMOD EXEC. Some CMOD options are Binder specific and some are LOAD/GENMOD/Prelinker specific. If any Binder specific options are specified, CMOD will use the Binder. If any LOAD/GENMOD/Prelinker specific options are specified, CMOD will use LOAD/GENMOD/Prelinker. If both types of options are specified, the type specified first will determine what CMOD uses. Warning messages will then be issued for the other type. If no Binder specific or LOAD/GENMOD/Prelinker specific options are specified, CMOD will check the value of the `_CNAME` environment variable in the CENV group of GLOBALV. If this is set to CBXFINIT, which indicates that XL C/C++ is being used, CMOD will use the Binder. Otherwise, it will use LOAD/GENMOD/Prelinker. Refer to Table 7 on page 57 for more information on CMOD options.

Before using the CMOD EXEC, you should issue a GLOBAL TXTLIB for any user libraries that contain objects that you want to include.

Note: If your application performs long double arithmetic, you must have the CMSLIB TXTLIB available.

The general form of the CMOD EXEC is:



textname

is the name of an object module generated by the CC EXEC.

Note: The file containing the function main should be the first file named in the CMOD EXEC. The compiler verifies that main exists by creating a special CSECT that references main.

To specify the name of the executable module, use the MODNAME option of the CMOD EXEC. The CMOD EXEC stores the executable module in a file specified on the MODNAME option. If you do not explicitly name the file in which you want the executable module to be stored, the name of the first object module specified on the CMOD EXEC will be used as the default.

Language Environment Sidedeck Files and TXTLIBs

The CMOD EXEC automatically sets up the appropriate GLOBAL TXTLIB commands and accesses the appropriate sidedeck files to properly create both non-XPLINK and XPLINK C and C++ programs. If you wish to create these without using the CMOD EXEC, that is invoking the Binder yourself, you must also execute the necessary GLOBAL TXTLIB commands prior to invoking the Binder and make the necessary sidedeck files available as primary input to the Binder, as follows:

- For non-XPLINK C programs:

```
GLOBAL TXTLIB SCEELKED
Sidedeck file(s): none
```

- For XPLINK C programs:

```
GLOBAL TXTLIB SCEEBND2
Sidedeck file(s): CELHS003 CELHS001
```

- For non-XPLINK C++ programs:

```
GLOBAL TXTLIB SCEELKD SCEECPP
Sidedeck file(s): C128
```

- For XPLINK C++ programs:

```
GLOBAL TXTLIB SCEEBND2
Sidedeck file(s): CELHSCPP CELHS003 CELHS001 C128
```

The sidedeck files are on the Language Environment disk with a file type of TEXT.

CMOD Options

Table 7. CMOD options	
Option	Description
Binder specific options	
BINDOPTS(<i>options</i>)	Specifies options for the Binder. These options may be any of the options supported by the Binder. For complete descriptions of these options, see the z/VM: Program Management Binder for CMS .
C++	Specifies that at least one of the text decks is C++. This must be specified for C++ code to be correctly linked.
DLL(<i>side_file_names</i>)	<p>If a side file name is not specified, this just passes the DYNAM DLL option to the Binder. It is the same as specifying BINDOPTS (DYNAM DLL), which enables the module for dynamic linking. A definition side file will be produced with the same name as the first text deck name, and a file type of SYSDEFSD.</p> <p>If a side file name is specified, the DYNAM DLL option is still passed to the Binder, but also the Binder will process the definition side file specified. An 8-character CMS file name is specified. CMOD will look for that file name with a file type of SYSDEFSD. Multiple names can be specified, separated by blanks.</p> <p>For information about this option, see the z/VM: Program Management Binder for CMS.</p> <p>For more information about DLLs (Dynamic Link Libraries), see the section about building and using DLLs in the z/OS: XL C/C++ Programming Guide (https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147315/\$file/cbcpx01_v2r5.pdf) and the sections about binder processing in the z/OS: XL C/C++ User's Guide (https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147307/\$file/cbcux01_v2r5.pdf).</p>
XPLINK	Specifies that the text deck(s) has been compiled with the XPLINK option. Generally speaking, XPLINK text decks cannot be bound with non-XPLINK text decks.
LOAD/GENMOD/Prelinker specific options	
AMODE	Specifies the addressing mode in which the program will be entered in a virtual machine. For a complete description of AMODE, see the LOAD command in the z/VM: CMS Commands and Utilities Reference .
AUTO NOAUTO	Specifies that your disks are to be searched for TEXT files for use in resolving undefined references.

Table 7. CMOD options (continued)

Option	Description
CPLINK(<i>options</i>)	Specifies options for the Prelinker. Note: The Prelinker is not supported for use with XL C/C++.
DUP NODUP	Specifies that an error message is to be generated if duplicate CSECT names are encountered. If you want to ensure that only one copy of a object module is loaded, use the NODUP option.
GENMOD(<i>options</i>)	Passes any options to the GENMOD command.
INV NOINV	Specifies that invalid card images are not to be included in the load map.
LET NOLET	Specifies that all LOAD errors for the load module are to be ignored and an attempt to generate a module will be made.
ORIGIN	Specifies where CMS loads the program. This location must be in the CMS transient area or in any free CMS storage.
RLD NORLD	Specifies that relocation directory information is to be saved in the load module.
STR NOSTR	Specifies that storage is to be initialized during the generation of the executable module.
RMODE	Specifies where the program is to reside in a virtual machine with greater than 16MB of storage. For a complete description of RMODE, see the LOAD command in the z/VM: CMS Commands and Utilities Reference .
Common options	
MAP NOMAP	The specified option is passed to the Binder or the LOAD command. For MAP (which is the default), the Binder will incorporate a module map into the SYSPRINT output (see the z/VM: Program Management Binder for CMS , for more information); the LOAD command will generate a load map file on your A disk with the name LOAD MAP A.
MODNAME(<i>modulename</i>)	The default is to generate an executable module having the same file name as the first object module specified, a file type of MODULE, and a file mode of A. The MODNAME option allows you to give a specific name to the executable module. If you specify this option, CMOD creates an executable module named <i>modulename</i> MODULE A.

Examples

```
KNOWN:  - Only one object module is to be loaded.
        - The object module to be loaded has file name
          PRODUCE, file type TEXT, and file mode A.
        - Default options and file names are to be used.
```

```
USE THE FOLLOWING COMMAND:
    CMOD PRODUCE
```

Note: File type and file mode are not specified on the CMOD EXEC.

Figure 27. Example 1 - Using the CMOD EXEC

```
KNOWN:  - The two object modules to be loaded are:
        - GRAPHING TEXT A
        - TRIG TEXT A
        - GRAPHING TEXT A contains the main().
        - A load module map is to be generated.
        - The load module produced is to be called MATH MODULE A.
```

```
USE THE FOLLOWING COMMAND:
    CMOD GRAPHING TRIG (MODNAME(MATH) MAP
```

Figure 28. Example 2 - Using the CMOD EXEC

```
KNOWN:  - Only one object module is to be loaded.
        - The object module to be loaded has file name
          DLLA07, file type TEXT, and file mode A.
        - The load module is to be a DLL
        - Default options and file names are to be used.
```

```
USE THE FOLLOWING COMMAND:
    CMOD DLLA07 (DLL
```

Note: A definition side file with name DLLA07 and type SYSDEFSD will be produced on the A disk.

Figure 29. Example 3 - Using the CMOD EXEC

```
KNOWN:  - Only one object module is to be loaded.
        - The object module to be loaded has file name
          C955A07, file type TEXT, and file mode A.
        - C955A07 calls functions in the DLLA07 DLL.
        - There is a definition side file called DLLA07 SYSDEFSD.
        - Default options and file names are to be used.
```

```
USE THE FOLLOWING COMMAND:
    CMOD C955A07 (DLL(DLLA07)
```

Figure 30. Example 4 - Using the CMOD EXEC

Using the LOAD and GENMOD Commands

Note: This method of creating an executable program cannot be used for C text files that were compiled with either the LONGNAME or RENT options, or for C++ text files. These text files need to be processed by the binder to resolve writable static references and/or map long internal names to short external names.

The loader can also be invoked under VM/CMS by using the LOAD command processor. For complete information about the LOAD, INCLUDE, and GENMOD commands, see the [z/VM: CMS Commands and Utilities Reference](#).

Compilation using the CC command produces an object module with the file type TEXT. To run the program, you must load the object module to form a load module before you can execute it.

The LOAD command invokes the loader, which loads one or more object modules and creates an executable module in virtual storage.

Note that the object modules you are loading with the LOAD command must have a file type of TEXT. If you are loading several object modules, the file names must be separated by at least one blank. You can also specify load options following the input file names. If you want to specify more than one load option, the options must be separated by blanks.

Default options for the LOAD command are described in the [*z/VM: CMS Commands and Utilities Reference*](#).

The general form of the LOAD command is:

```
LOAD filename1 filename2 ... (options
```

Note: If the main program is C, you should include RESET CEESTART under the options for the LOAD command. The object module that contains the main must be the first one specified.

To store the executable module that was created by the loader in a file, use the GENMOD command. The GENMOD command will take a copy of the executable module in virtual storage and store it with the file name specified on the GENMOD command. Only the file name is required on the GENMOD command.

The general form of the GENMOD command is:

```
GENMOD filename (options
```

Notes:

1. If you specify a file type, it must be MODULE.
2. If the main program is C, then under the options for the GENMOD command, you should include FROM CEESTART.

If you do not explicitly name the file in which you want the load module to be stored, the GENMOD command processor defaults to the name of the first entry point in the load map. The following example shows you how to override the default to produce a load module with a user-specified file name.

```
KNOWN:  - Three object modules are to be loaded:
        - IMPORTS TEXT A
        - EXPORTS TEXT A
        - FORMULA TEXT A
        - The load module is to be called GNP MODULE A.
        - The main procedure is in IMPORTS.
        - Default options are to be used.

USE THE FOLLOWING COMMAND:
GLOBAL TXTLIB SCEELKED CMSLIB
LOAD IMPORTS EXPORTS FORMULA (RESET CEESTART
GENMOD GNP (FROM CEESTART
```

Figure 31. Using the LOAD and GENMOD commands

For more information on linking modules for interlanguage calls, see the [z/OS: Language Environment Programming Guide \(https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380682/\\$file/ceea200_v2r5.pdf\)](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380682/$file/ceea200_v2r5.pdf).

Using the BIND Command

The BIND command invokes the z/VM Program Management Binder for CMS, which encompasses the functionality of the Prelinker, the LKED command, and the LOAD and GENMOD commands. In addition, it supports the Program Object format which is required for some compiler options such as XPLINK.

The Prelinker is not supported for use with XL C/C++. Any C programs which use the RENT or LONGNAME options, or any C++ programs must use the Binder to create an executable module.

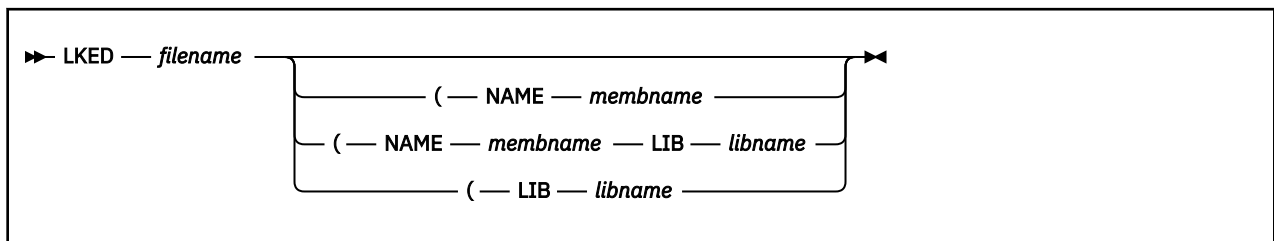
For more information about the Program Management Binder for CMS, see the *z/VM: Program Management Binder for CMS* and the sections about binder processing in the *z/OS: XL C/C++ User's Guide* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147307/\\$file/cbcux01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147307/$file/cbcux01_v2r5.pdf)).

Using the LKED Command

The LKED command is used to create a member of a CMS load library. CMS load libraries, like text libraries, are in CMS partitioned data set formats. Text libraries contain applications that contain unresolved external references to other routines. Load libraries, on the other hand, contain applications with external references that have already been resolved, thus saving overhead every time the application is loaded.

Your TEXT file is input to the LKED command. If your application calls a subroutine with object code stored as a separate TEXT file or as a member of a text library, you must define the files that contain the subroutines used by your application with a FILEDEF command.

After you issue the appropriate FILEDEF commands, issue the LKED command as follows:



filename

is the name of the TEXT file that contains your object code and/or linkage editor control cards.

NAME memname

specifies the member name to be used for the load module that is created.

LIB libname

specifies the name of the LOADLIB file where the resulting load module is placed.

The following example causes the automatic call library to search SCEELKED to resolve external references, creates a load library member named PROGRAM1, and stores it in a CMS load library with the name USERLOAD.

```
FILEDEF SYSLIB DISK SCEELKED TXTLIB E
LKED PROGRAM1 (NAME PROGRAM1 LIB USERLOAD
```

Using FILEDEF to Define Input and Output Files

If your program opens files by ddname (`fopen("DD:INFILE", "r")`), you must issue a corresponding FILEDEF prior to executing your program. The FILEDEF command relates the ddname of the input or output file specified in your application with an I/O device. For example, if PROGRAM1 contains a ddname of an input file stored on your A disk as MYDATA INPUT, issue the following command:

```
FILEDEF infile DISK MYDATA INPUT A
```

where *infile* is the ddname of the input file specified in PROGRAM1.

Preparing a Reentrant Program

Reentrancy allows more than one user to share a single copy of a load module or to repeatedly use a load module without reloading it.

Reentrant programs can be categorized by their reentrancy type as follows:

- Natural reentrancy - Programs that contain no writable static and do not require additional processing to make them reentrant.
- Constructed reentrancy - Programs that contain writable static and require additional processing to make them reentrant.

Note: All C++ programs use constructed reentrancy. They cannot be compiled with the NORENT option.

Writable static is storage that changes and is maintained throughout program execution. It is made up of:

- All program variables with the static storage class.
- All program variables receiving the extern storage class
- All writable strings

Note: If your program contains no writable strings and none of your static or extern variables are updated in your application (that is, they are read only), your program is naturally reentrant.

To generate a reentrant load module, you must follow these steps:

1. If your program contains writable static, you must compile all your C source files using the RENT compiler option.

If you are unsure about whether your program contains writable static, compile it with the RENT option. Invoking the Binder with the MAP option and the object module as input produces a module map. Any writable static data in the object module appears in the writable static section of the map.

2. Use the Binder to combine all input object modules into a single output object module.

3. Optionally, do one of the following:

- Have your system programmer link/install your program into a discontinuous saved segment (DCSS). For information about using saved segments, see the [*z/VM: Saved Segments Planning and Administration*](#).
- Install your program as a nucleus extension by using the VM/CMS NUCXLOAD command. For more information about the NUCXLOAD command, see the [*z/VM: CMS Commands and Utilities Reference*](#).

You do not need to install your program to run but if you do not, you will not gain all the benefits of reentrancy.

Linking Modules for Interlanguage Calls

For information on link-editing modules for interlanguage calls, see the *z/OS: Language Environment Programming Guide* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380682/\\$file/ceea200_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380682/$file/ceea200_v2r5.pdf)).

Running a Program

Once you have compiled and loaded your C/C++ program, you can run it one of two ways:

1. Using the file name of the load module. For example:

```
TESTRUN
```

2. Using the START command immediately after a LOAD or LOADMOD command. For example:

```
LOADMOD TESTRUN  
START
```

Making the Runtime Libraries Available for Execution

The Language Environment must be available at run time for your application to use the dynamic library routines. The following sections describe how to make these libraries available to your programs.

Making the Language Environment Library Available for VM/CMS

The C specific portions of the Language Environment are in modules CEEEV003, CELHV003 and EDCZ24. CEEEV003 is the main C runtime module for non-XPLINK programs, CELHV003 is the main C runtime module for XPLINK programs, and EDCZ24 is the I/O routine module.

These can be loaded as nucleus extensions, discontinuous saved segments (DCSSs), or directly from the Language Environment minidisk or directory. Nucleus extensions and DCSSs offer improved performance.

Other portions of the Language Environment dynamic environment are on the Language Environment minidisk or directory in the form of separate modules and the SCEERUN loadlib. The modules can also be loaded as nucleus extensions or DCSSs. The loadlib needs to be accessed with the GLOBAL LOADLIB command. For example:

```
GLOBAL LOADLIB SCEERUN
```

Search Sequence for Library Files

The search order for the library files is:

1. Nucleus extension
2. Saved segment
3. LOADLIB

For best performance, the library should be loaded as a nucleus extension.

Specifying Runtime Options

Each time a C/C++ program is run, values must be established for a set of C/C++ runtime options. These options affect many of the properties of a C/C++ program's execution, including its performance, its error handling characteristics, and its production of debugging and tuning information.

If the EXECOPS runtime option is in effect and if you want to specify additional runtime options on the command line, specify the options, followed by a slash (/), followed by the parameters you want to pass to the main function.

If the NOEXECOPS runtime option is in effect, any arguments and options that you specify on the command line (including the slash, if present) are passed as arguments to the main function. Runtime options are described in [“Specifying Runtime Options” on page 39](#).

Each time the program is run, the default runtime options selected during C/C++ installation apply unless overridden by options specified in a `#pragma runopts` directive in your source program or by command line options specified at the time of program execution.

Runtime options are specified using the `runopts` pragma or in the *parameter-string* on the command line when you invoke your C/C++ program. The *parameter-string* contains two fields separated by a slash(/), and takes the form:

```
[runtime options/][parameter string]
```

The first field is passed to the program initialization routine as a runtime option list; the second passes to the main function. If you do not specify any runtime options but want to pass arguments, you must precede the arguments with a slash.

The following example shows you how to specify runtime options and pass arguments when you invoke your program under VM/CMS.

```
KNOWN:  - The load module to be executed is called SURVEY MODULE A.
         - You want to pass the words THIS IS A TEST to
           the program.
         - The messages generated by the runtime library
           will be received in Japanese.
```

```
USE THE FOLLOWING COMMAND:
    SURVEY LANG(JA)/THIS IS A TEST
```

Figure 32. Running under CMS

Message Handling

By default, all C/C++ programs (including the compiler) set `emsg` off so that VM/CMS messages generated during normal execution of C library functions are not output to the terminal along with `stdout` and `stderr`. The C system function restores the `emsg` setting, issues the given command in the system call, and sets `emsg` off again.

Use the `setenv()` function to set `emsg` via the C environment variable `_EDC_KEEP_EMMSG`, as follows:

```
setenv("_EDC_KEEP_EMMSG", "Y", 1);
```

This environment variable restores the `emsg` setting to its value prior to the execution of the C program, and keeps that value while the program is running.

There are four ways to get XL C/C++ to leave the `emsg` setting on. This allows any messages produced by VM/CMS during execution of your program to be displayed.

- Issue the CMS command `GLOBALV SELECT CENV SET _EDC_KEEP_EMMSG Y`.
- Modify the user exit `CEEBCINT` to issue a `setenv("_EDC_KEEP_EMMSG", "Y", 1)` and link this with your executable module.
- Create a variable length file with a line of the following format. (Spaces are not permitted.)

```
_EDC_KEEP_EMMSG=Y
```

Create a `FILEDEF EDCENV DISK fn ft fm` for the same file. During initialization of the root main program, XL C/C++ opens the file associated with the `ddname` `EDCENV` and sets the appropriate environment variables.

- Issue a `setenv("_EDC_KEEP_EMMSG", "Y", 1)` in your program. This restores `emsg` to the value in effect when your program was invoked.

The environment variable may be set any time in a C program, or may be set in the runtime user exit `CEEBCINT`.

If the `emsg` setting is changed via a `system()` call once `_EDC_KEEP_EMMSG` has been set, then the new `emsg` setting will be maintained even after the C program terminates.

For additional information on the `setenv()` library function, see the [XL C/C++ for z/VM: Runtime Library Reference](#). For more information on environment variables, see the [z/OS: XL C/C++ Programming Guide \(https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147315/\\$file/cbcpx01_v2r5.pdf\)](#). Additional information on runtime user exits in XL C/C++ is also available in the [z/OS: XL C/C++ Programming Guide \(https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147315/\\$file/cbcpx01_v2r5.pdf\)](#).

Chapter 9. Compiling a C/C++ Program under OpenExtensions

This chapter describes how to compile C/C++ programs under OpenExtensions using the OpenExtensions c89 and cxx utilities. For detailed information regarding the c89/cxx utility options and operands, see the [z/VM: OpenExtensions Commands Reference](#).

The c89/cxx utilities call the XL C/C++ compiler. You must have access to the Language Environment C/C++ runtime library, because the compiler calls functions in the library to compile the code.

Compiling with c89/cxx

An OpenExtensions C/C++ application program with source code in BFS files or CMS native files must be compiled to create output object files residing either in BFS files or z/VM record files.

Application source code can be compiled and built at one time or compiled and then bound at another time with other application source files or compiled objects.

To compile and build an OpenExtensions application program from the OpenExtensions shell, use the c89 or cxx utility.

Note: All references to c89 also apply to cxx unless otherwise specified.

The syntax for cxx is the same as for c89. The syntax is:

```
c89 [-options ...] [file.c ...] [file.a ...] [file.o ...] [-l libname]
```

options

specifies one or more of the c89 options described in [“c89 Selectable Compiler Settings”](#) on page 35.

file.c

specifies the name of the source file.

file.o

specifies the name of the object file.

file.a

specifies the name of the archive file.

libname

is name of the archive library.

Note: You can use the c89 utility to compile and build application program source and objects from within the shell or directly from CMS. If you use c89, you must keep track of and maintain all the source and object files for the application program. However, you can take advantage of the make utility and create makefiles to maintain your OpenExtensions application source and object files automatically when you update individual modules. The make utility runs c89 for you. However, make must be run from the shell.

For more information on using the make utility, see [Chapter 15, “OpenExtensions ar and make Utilities,”](#) on page 101 and [z/VM: OpenExtensions Advanced Application Programming Tools](#).

To compile source files without binding them, enter the c89 command with the -c option to create object file output. Use the -o option to specify placement of the application program executable file to be generated. The placement of the intermediate object file output depends on the location of the source file:

- If the C/C++ source module is a BFS file, the object file is created in the working directory.
- If the C/C++ source module is a CMS native file, the object file is created as a CMS native file. The object file is placed in the CMS minidisk or SFS directory accessed as file mode A.

For example, if the C/C++ source is in a minidisk file named `USERSRC C B`, the object is placed in the file `USERSRC TEXT A`. Because the CMS file ID is always converted to uppercase, you can specify it in lowercase or mixed case.

- Compiling application source to produce only object files.
 - To compile C/C++ source to create the default object file `usersource.o` in your working BFS directory, specify:

```
c89 -c usersource.c
```

- To compile C/C++ source to create an object file as a file on the A disk, specify:

```
c89 -c //approg.c
```

- Compiling and binding application source to produce an application executable file.
 - To compile an application source file to create the object file `usersource.o` in the BFS working directory and the executable file `mymod.out` in the `/app/bin` directory, specify:

```
c89 -o /app/bin/mymod.out usersource.c
```

- To compile the C source file `MAINBAL C` on the B disk and build it to produce the application executable file `/u/parker/myappls/bin/mainbal.out`, specify:

```
c89 -o /u/parker/myappls/bin/mainbal.out //mainbal.c.b
```

Compiler Selection

By default, the `c89` utility calls the XL C/C++ compiler (or the IBM C/C++ for z/VM compiler, whichever is installed). If you had previously set `c89` to call the IBM C for VM/ESA compiler and want to change to the XL C/C++ compiler, issue the following command to specify the C/C++ compiler module (`CBXFINIT`) on the `_CNAME` environment variable in the `CENV` group of `GLOBALV`:

```
globalv select cenv setlp_cname cbxfinit
```

This will also cause `c89` to call the Binder instead of the Prelinker.

To use the IBM C for VM/ESA compiler instead of the XL C/C++ compiler, you can specify the C compiler module (`CBC310`) by issuing the following command:

```
globalv select cenv setlp_cname cbc310
```

The `cxx` utility ignores the setting of the `_CNAME` environment variable and always calls the `CBXFINIT` module.

Compiling and Building in One Step with `c89/cxx`

To compile and build an OpenExtensions C/C++ application program in one step to produce an executable file, specify the `c89/cxx` utility without specifying the `-c` option.

Note: To compile source files without building them, use the `c89 -c` option. This will create object files only.

You can use the `-o` option with the command to specify the name and location of the application program executable file to be created.

- To compile and build an application program source file to create the default executable file `a.out` in the BFS working directory, specify:

```
c89 usersource.c
```

- To compile and build 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
```

- To compile and build several application source files to create the `mymod.out` executable file in your `/app/bin` directory, specify:

```
c89 -o /app/bin/mymod.out usersource.c ottrsrc.c //pwapp.c
```

- To compile and build an application source file to create the `MYLOADMD` module file on your A disk specify:

```
c89 -o //myloadmd.module usersource.c
```

- To compile and build an application source file with a previously compiled object file to create the executable file `zinfo` in your `/approg/lib` BFS directory, specify:

```
c89 -o /approg/lib/zinfo usersource.c existobj.o //pwapp.c
```

Using the make Utility

You can use the OpenExtensions shell `make` utility to control your OpenExtensions C/C++ application's parts. The `make` utility calls the `c89` utility by default to compile and bind the programs specified in the previously created `makefile`.

The `/etc/startup.mk` file contains the `make` default rules.

For example, if you have the file `/u/jake/appwrk/makefile.c` that contains the dependencies for your C application program `primappl` and you make changes to the source file `subordpgm.c`, you can recompile the application by entering:

```
cd appwrk
make -f makefile.c
```

The result is the same as if you had entered:

```
c89 -O -o primappl ./appwrk/subordpgm.c
```

Note: The OpenExtensions `make` utility requires that any application program source files to be "maintained" through use of a `makefile` reside in BFS files. To compile and build C/C++ source files that are in CMS native files you must use the `c89` utility directly.

For a description of the `make` utility, see the [z/VM: OpenExtensions Commands Reference](#). For a detailed discussion on how to create and use `makefiles` to manage application parts, see the [z/VM: OpenExtensions Advanced Application Programming Tools](#).

Chapter 10. Binding and Running a C/C++ Program under OpenExtensions

This chapter describes how to bind and run C/C++ programs under OpenExtensions.

The interfaces to the CMS module build facilities for OpenExtensions C/C++ applications are the OpenExtensions `c89` and `cxx` utilities. You can use `c89/cxx` to compile and build an OpenExtensions C/C++ application program in one step or bind application object files after compilation. For more information on compiling with the `c89/cxx` utility, refer to [Chapter 9, “Compiling a C/C++ Program under OpenExtensions,”](#) on page 65.

Note: All references to `c89` in the following sections also apply to `cxx` unless otherwise specified.

Using the `c89` Utility to Bind and Create Executable Files

To bind an OpenExtensions C/C++ application program's object files to produce an executable file, specify the `c89` utility and pass it object files (*file.o* BFS files or CMS native files). The `c89` utility recognizes that these are object files produced by previous C/C++ compilations and does not invoke the compiler for them.

To compile source files without binding them, use the `c89 -c` option to create object files only.

You can use the `-o` option with the command to specify the name and location of the application program executable file to be created.

- To bind an application program object file to create the default executable file `a.out` in the working directory, specify:

```
c89 usersource.o
```

- To bind an application object file to create the `mymod.out` executable file in the `app/bin` directory, relative to your working directory, specify:

```
c89 -o ./app/bin/mymod.out usersource.o
```

where `usersource.o` is the object file created by compilation with `c89`.

- To bind several application object files to create the `mymod.out` executable file in the `app/bin` directory, relative to your working directory, specify:

```
c89 -o ./app/bin/mymod.out usersrc.o othersrc.o
```

- To bind an application object file to create the `MYLOADMD` module file on the A disk specify:

```
c89 -o //myloadmd.module usersource.o
```

- To compile and bind an application source file with several previously compiled object files to create the executable file `zinfo` in the `approg/lib` subdirectory, relative to your working directory, specify:

```
c89 -o ./approg/lib/zinfo usersrc.c existobj.o //pgmobj.text
```

`c89` Binder Options

The `c89` and `cxx` utilities specify default values for some Binder options. They also pass Binder options by using the `-W` option. For more information on using the `c89` options, see [Chapter 5, “Compiler Options under OpenExtensions,”](#) on page 35.

Binder Options

c89 uses the following Binder options, all of which can be overridden using the -W option:

```
CASE MIXED TERM DISK
```

cxx uses the following Binder options:

```
CASE MIXED TERM DISK RENT DYNAM DLL
```

The following example shows how to use the -W option to pass a Binder option.

```
c89 -Wb,b,map,case,upper hello.c
```

For more information about Binder options, see [z/VM: Program Management Binder for CMS](#).

Specifying Runtime Options under OpenExtensions

If you have an OpenExtensions C/C++ application program executable file in the byte file system (BFS), you cannot run the executable file by simply entering its name on the CMS command line, as you would a traditional CMS application program. Instead, you can execute the application by specifying its name on the CMS command OPENVM RUN. However, OPENVM RUN does not support passing of runtime options to the application.

Runtime options, needed for the OpenExtensions application program residing in the BFS, can be passed from a `#pragma runopts` preprocessor directive at compile time. When runtime options are specified in this way a CEEUOPT control section (CSECT) is created and is linked with the application program by the c89 utility. Because only one CEEUOPT CSECT can be linked with an application program, you should code a `#pragma runopts` directive in the compilation unit for the `main()` function. For more information about `#pragma runopts`, refer to [“Runtime Options Using Language Environment”](#) on page 39.

Note: Also, you can create a CEEUOPT CSECT as a separate step using the CEEXOPT macro and bind the CSECT with the application program object files using c89.

Running under OpenExtensions

This section discusses how to run your OpenExtensions C/C++ application program executable files on the z/VM system.

OpenExtensions Application Program Environments

OpenExtensions supports the following environments, from which you can run your OpenExtensions C/C++ application programs:

- OpenExtensions shell
- CMS

Placing a CMS Application Program Load Module in the File System

If you have an OpenExtensions C/C++ application program executable file as a CMS native file and want to place it in the BFS, you can use the following OpenExtensions CMS commands to copy the file into a BFS file:

- OPENVM PUTBFS

For a description of this command, see the [z/VM: OpenExtensions Commands Reference](#). For examples of using this command to copy CMS files into BFS, see the [z/VM: OpenExtensions User's Guide](#).

Running a CMS Module from the OpenExtensions Shell

If your OpenExtensions C/C++ program is a CMS module file on a minidisk or in the shared file system, you can invoke it from the shell by using the `cms` command. For example, to run `PROG1 MODULE A`, execute the following command:

```
cms prog1
```

If you want to make the module file transparent to the shell, you need to create an external link in the BFS that points to the file. For example, to run `PROG1 MODULE A`, you can create a file in the BFS that represents the module by using the following command:

```
openvm create extlink /u/mydir/prog1 cmsexec PROG1 MODULE A
```

You can run the module transparently from the shell by using the following command:

```
prog1
```

For more information on creating external links, see the [z/VM: OpenExtensions Commands Reference](#).

Running an OpenExtensions XL C/C++ Application Executable File from the OpenExtensions Shell

If the application executable file is a BFS file, you must either run it from the shell interactively or invoke it indirectly through the CMS command `OPENVM RUN`.

Issuing the Executable Filename from the Shell

Before a BFS program can be run in the OpenExtensions shell, it must be given the appropriate mode authority for a user or group of users to run it. You can update the mode authority for an executable program file by using the `chmod` command. See the [z/VM: OpenExtensions Commands Reference](#) for the format and description of `chmod`.

After you have updated the mode authority, enter the program name from the OpenExtensions shell command line. For example, if you want to run the program `datcrnch` from your working directory, you have the directory where the program resides defined in your search path, and you are authorized to run the program, enter:

```
datcrnch
```

Issuing a Setup Shell Script Filename from the Shell

To run an OpenExtensions shell script that sets up an OpenExtensions executable file and then runs the program, give the appropriate mode authority for a user or group of users to run it. You can update the mode authority for a shell script file by using the `chmod` command. See the [z/VM: OpenExtensions Commands Reference](#) for the format and description of `chmod`. After mode authority has been given, enter the script filename from the OpenExtensions shell command line.

Chapter 11. Object Library Utility

This chapter describes how to use the Object Library Utility to update libraries of object files. On VM/CMS, a library is a text library (TXTLIB) with object files as members.

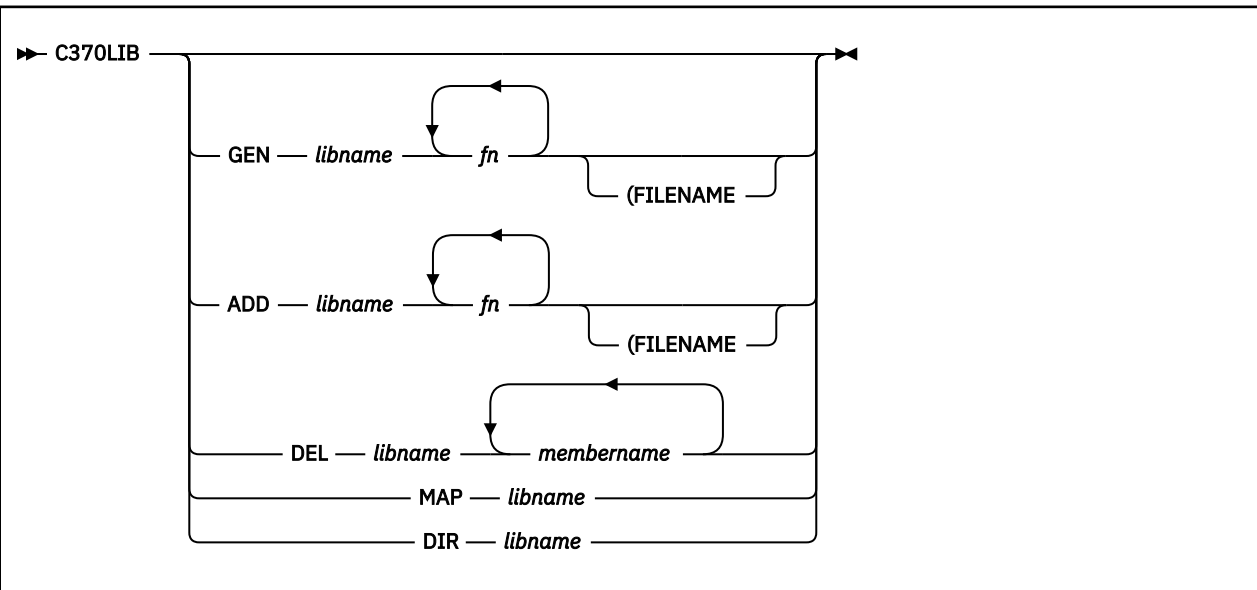
Object libraries provide convenient packaging of object files. With the Object Library Utility, a library can contain objects files compiled with long names, short names, writable static data, or XPLINK. The Object Library Utility stores source member symbol information with different attributes. This information is stored in two special members of the library, the Basic Directory Member (@@DC370\$) and the Enhanced Directory Member (@@DC390\$). Both are referred to in this chapter as the C370LIB directory.

Note: The TXTLIB command under VM/CMS also creates object libraries but you cannot include external names longer than 8 characters. The syntax for the Object Library Utility is similar to the TXTLIB command.

Commands to add object files to a library, to delete object files from a library, or to build the C370LIB directory for a library are available. Use the DIR command to build the C370LIB directory for a library of object files. Use the MAP command to list the contents of the C370LIB directory.

Creating an Object Library under VM/CMS

You use the C370LIB EXEC to create an object library.



GEN

creates a TXTLIB on your A disk. If a TXTLIB with the same name already exists, it is replaced.

ADD

adds TEXT files as members to an existing TXTLIB on a read/write disk. No checking is done for duplicate names, entry points, or CSECTs.

DEL

deletes members from a TXTLIB on a read/write disk and compresses the TXTLIB to remove unused space. If more than one member exists with the same name, only the first entry is deleted.

MAP

lists the names (entry points) of TXTLIB members.

MAP produces a file, *libname* MAP, on your A disk. See [“Object Library Utility Map” on page 75](#) for more information on the map.

DIR

builds the C370LIB directory. The C370LIB directory contains the names (entry points) of library members.

The DIR function is only necessary if TEXT files were previously added or deleted from the TXTLIB without using C370LIB.

libname

specifies the file name of a file with a file type of TXTLIB, which can be one of the following:

- Library to be created or listed
- Library to which members are to be added
- Library from which members are to be deleted
- Library for which a C370LIB directory is to be built

fn

specifies one or more names of files with file types of TEXT, that you want to add to a TXTLIB.

membername

specifies one or more names of TXTLIB members that you want to delete.

FILENAME

indicates that all the specified file names (*fn ...*) will be used as the member names for their respective entries in the TXTLIB file.

C370LIB must be used to update a TXTLIB with TEXT files produced by compiling C programs with the LONGNAME option, or compiling C++ programs. The VM/CMS TXTLIB command cannot be used to do this directly, and an error can result if this is attempted.

When a TEXT file is added to a library, its member name is selected according to the following hierarchy:

1. From the file name, if the FILENAME option is specified
2. From the NAME control statement, if present, in the TEXT file
3. From the file name.

The CMS TXTLIB command GEN, ADD, and DEL functions are used as part of the C370LIB GEN, ADD and DEL functions. Thus, any TXTLIB restrictions apply also to C370LIB unless otherwise stated. For information about the TXTLIB command, see the [z/VM: CMS Commands and Utilities Reference](#).

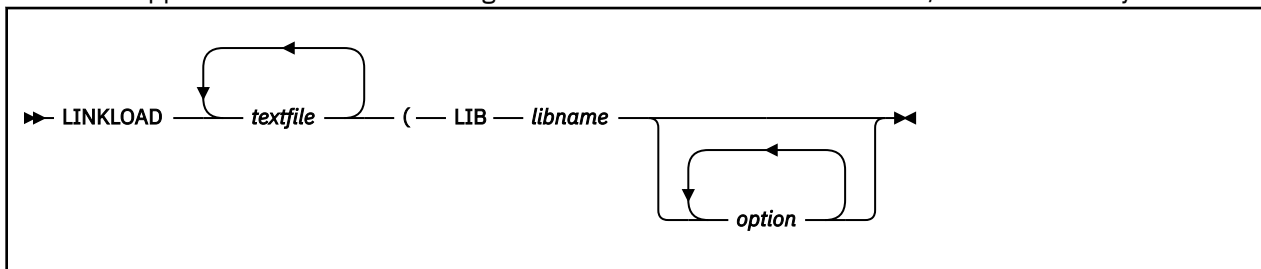
Members must be deleted by their member name. Any attempt to delete a member using a name other than the member name will result in a warning message.

In the following example, the C programs SUB1 C and SUB2 C are compiled with the LONGNAME option. The function library, SUBLIB TXTLIB A, is created with SUB1 TEXT using the GEN command of C370LIB, the Object Library Utility. SUB2 TEXT is added to the library using the ADD command.

```
CC SUB1 (LO
CC SUB2 (LO
C370LIB GEN SUBLIB SUB1
C370LIB ADD SUBLIB SUB2
```

LINKLOAD EXEC

The IBM-supplied LINKLOAD EXEC will generate a fetchable member of a VM/CMS load library.



textfile

specifies one or more names of the input text files. The file type of the object files must be TEXT, and the source programs must have contained a `#pragma linkage(name, FETCHABLE)` preprocessor directive. Note that you do not specify the file type or the file mode when using the LINKLOAD EXEC.

libname

specifies the name of the library where the load member is to be stored.

option

specifies the options you want to apply when you are generating the fetchable load library member:

MBR

Specifies that the next argument, *memname*, is the name of the member within the load library that is to be generated. If you do not specify a member name, the name of the text file containing the fetchable code is used.

CPLINK options

Passes options to the Prelinker. CPLINK is called if it is required by the text file or if a CPLINK option is given.

Note: The Prelinker is not supported for use with XL C/C++.

LKED

Specifies that the options following it are to be passed to LKED. If you do not use this option, default options are used.

Only one of the following options can be specified on a given invocation of LINKLOAD:

ADD

Specifies that the load member generated by the LINKLOAD EXEC is to be added to the load library. If a member by the same name already exists, the new member will not be added.

REPLACE

Specifies that the load member generated by the LINKLOAD EXEC is to replace the member having the same name in the load library. If a member by the same name does not exist, the new member is added.

NEW

Specifies that if a load library of the given name exists, then it is erased, and a new load library containing the new member is created.

Object Library Utility Map

The Object Library Utility produces a listing for a given library when the MAP command is specified. The listing contains information on each member of the library. A representative example is shown in [Figure 33 on page 76](#).

```

=====
|                               Object Library Utility Map                               | 1
| C370LIB:5741A09 V7 R3 M00 IBM z/VM                               2022/09/28 21:19:26 |
|=====

Library Name: TS41949.A.OBJECT                               2022/09/28 21:19:26

*-----*
* Member Name: ASMSTUFF                               (D) 2022/09/28 21:19:26 * 2
*                               569623400      R01 M01 *
*-----*

(S) External Name: CSECT1
(S) External Name: ENTRY1

*-----*
* Member Name: CSTUFF                               (D) 2022/09/28 21:19:26 * 2
*                               5694A01      V1 R02 *
*-----*

(L) Function Name: foo
(WL) External Name: this_int_is_in_writable_static_and_its_name_will
                    _wrap_because_it_is_too_long

*-----*
* Member Name: CXXSTUFF                               (D) 2022/09/28 21:19:26 * 2
*                               5694A01      V1 R02 *
*-----*

3 User Comment: This is a user comment in CXXSTUFF

4 (L) Function Name: testeh()
  (L) Function Name: f1()
  (L) Function Name: operator++(U&)
  (WL) External Name: i1
  (WL) External Name: i2

===== END OF OBJECT LIBRARY MAP =====

```

Figure 33. Object Library Utility Map

1 Map Heading

The heading contains the product number, the compiler release number, the compiler version number, and the date and time the Object Library Utility step commenced. The name of the library immediately follows the heading. To the right of the name of the library 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 ID of the processor that produced the object module. The processor ID is based on the presence of an END record in the object module having the processor information in the appropriate format. If this information is not present, the Processor ID field is not listed.

The Timestamp field is presented in *yyyy/mm/dd* format. The meaning of the timestamp is enclosed in parentheses. That is, the Object Library Utility retains a timestamp for each member and selects the time according to the following hierarchy:

(P)

Indicates that the timestamp is extracted from the object module from the date form of `#pragma comment` or from the timestamp form of `#pragma comment`, whichever comes first.

(D)

Indicates that the timestamp is based on the time that the Object Library Utility DIR command was last issued.

(F)

Indicates that the timestamp is the date of the object module file at the time the ADD or GEN command was issued for the member. This is applicable to z/VM only.

(T)

Indicates that the timestamp is the time that the ADD command was issued for the member. This is applicable to MVS only.

3 User Comments

The user form of comments generated by `#pragma comment` is displayed. These comments are extracted from the END record. It is possible to manually add such comments on multiple END records and have them displayed in the listing. For more information on the END record, see the *z/OS: XL C/C++ Language Reference* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147308/\\$file/cbclx01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147308/$file/cbclx01_v2r5.pdf)).

4 Symbol Information

Immediately following the Member Heading (and user comments, if any) is a list of the defined objects contained within that member. Each symbol is prefixed by Type information enclosed in parentheses and either External Name or Function Name. Function Name appears provided the object module was compiled with the LONGNAME option and the symbol is the name of a defined external function. In all other cases External Name is displayed. The Type field gives additional information on each symbol. That is:

(L)

Indicates that the name is an L-name.

(S)

Indicates that the name is an S-name.

(W)

Indicates that this is a writable static object. If no W is present, this is not a writable static object.

(WL)

Indicates that this is an L-name and in writable static.

Chapter 12. Filter Utility

This chapter describes how to use the CXXFILT utility to convert mangled names to demangled names.

When XL C/C++ compiles a C++ program, it has the ability to encode function names. It also has the ability to encode other identifiers to include type and scoping information. This encoding process is called mangling. Mangled names ensure type-safe linking.

Use the CXXFILT utility to convert these mangled names to demangled names. The utility copies the characters from either a given file or from standard input, to standard output. It replaces all mangled names with their corresponding demangled names.

The CXXFILT utility demangles any of the following classes of mangled names when the appropriate options are specified.

regular names

Names that appear within the context of a function name or a member variable. For example, the mangled name `__1s__7ostreamFPCc` is demangled as `ostream::operator<<(const char*)`.

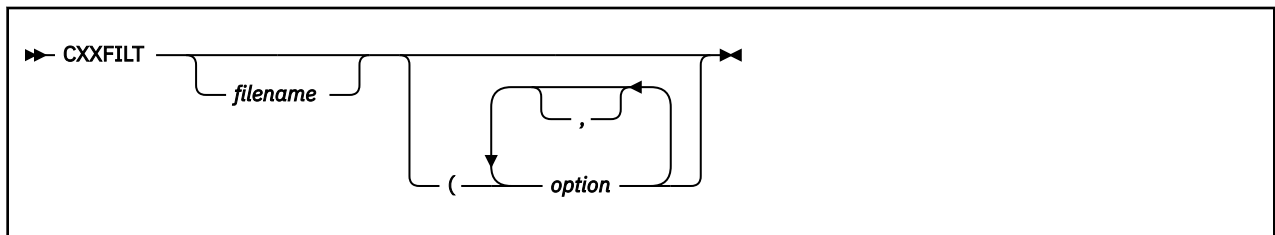
class names

Includes stand-alone class names that do not appear within the context of a function name or a member variable. For example, the stand-alone class name `Q2_1X1Y` is demangled as `X::Y`.

special names

Special compiler-generated class objects. For example, the compiler-generated symbol name `__vft1X` is demangled as `X::virtual-fn-table-ptr`.

The CXXFILT utility is run under VM/CMS by using the CXXFILT EXEC. The syntax of the CXXFILT command is:



filename

is the name of the file that contain the mangled names to be demangled. If you specify no file name, CXXFILT reads from stdin.

option

is the name of a CXXFILT option to be used. If you specify no options, NOSYMMAP, NOSIDEBYSIDE, NOWIDTH, REGULARNAME, NOCLASSNAME, and NOSPECIALNAME are used by default.

CXXFILT Options

SYMMAP | NOSYMMAP

DEFAULT: NOSYMMAP

The SYMMAP option produces a symbol map on standard output. The map contains a list of the mangled names and their corresponding demangled names. The map displays only the first 40 bytes of each demangled name and 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

The SIDEBYSIDE option displays each mangled name that is encountered in the input stream 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

The WIDTH option 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

The REGULARNAME option demangles regular names such as `pr__3F00Fv`. 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 standalone class name.

CLASSNAME | NOCLASSNAME

DEFAULT: NOCLASSNAME

The CLASSNAME option demangles standalone class names such as `Q2_1X1Y`.

To request that the mangled names be treated as standalone class names only, and never as a regular name, use both CLASSNAME and NOREGULARNAME.

SPECIALNAME | NOSPECIALNAME

DEFAULT: NOSPECIALNAME

The SPECIALNAME option demangles special names, such as compiler-generated symbol names, for example `__vft1X`.

To request that the mangled names be treated as special names only, and never as regular names, use CXXFILT (SPECIALNAME NOREGULARNAME).

Unknown Type of Name

If you cannot specify the type of name, use CXXFILT (SPECIALNAME CLASSNAME. This causes CXXFILT to attempt to demangle the name in the following order:

1. Regular name
2. Standalone class name
3. Special name

Running CXXFILT under VM/CMS

The CXXFILT EXEC accepts input by two methods: from stdin or from a file.

With the first method, enter names after invoking CXXFILT. You can specify one or more names on one or more lines. The output is displayed after you press Enter. Names that are successfully demangled, as well as those which are not demangled, are displayed in the same order as they were entered. To indicate end of input, enter /*.

In the following example, CXXFILT treats mangled names as regular names, produces a symbol mapping, and uses a field width 32 characters wide.

```

user> CXXFILT (SYMMAP WIDTH(32)
user> pr__3F00Fvxxxx
reply< F00::pr()                                xxxx
user> __ls__7ostreamFPCc
reply> ostream::operator<<(const char*)
user> __vft1X
reply> X::virtual-fn-table-ptr
user> /*

reply> C++ Symbol Mapping
reply> demangled                                mangled
reply> -----                                -----
reply>
reply> F00::pr()                                pr__3F00Fv
reply> ostream::operator<<(const char*)        __ls__7ostreamFPCc
reply> X::virtual-fn-table-ptr                  __vft1X

```

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 symbol mapping is displayed only after /* requests CXXFILT termination.

The second method of giving input to CXXFILT is to supply it in a file. CXXFILT supports fixed and variable file record formats. Each line of the 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.

NAMES FILE contains the following two mangled names:

```

pr__3F00Fv
__vft1X

```

Entering the following command:

```
CXXFILT NAMES FILE (SPECIALNAME WIDTH(35) SIDEBYSIDE
```

produces the following output:

```

F00::pr()                                pr__3F00Fv
X::virtual-fn-table-ptr                  __vft1X

```

CXXFILT terminates when it reads the end-of-file.

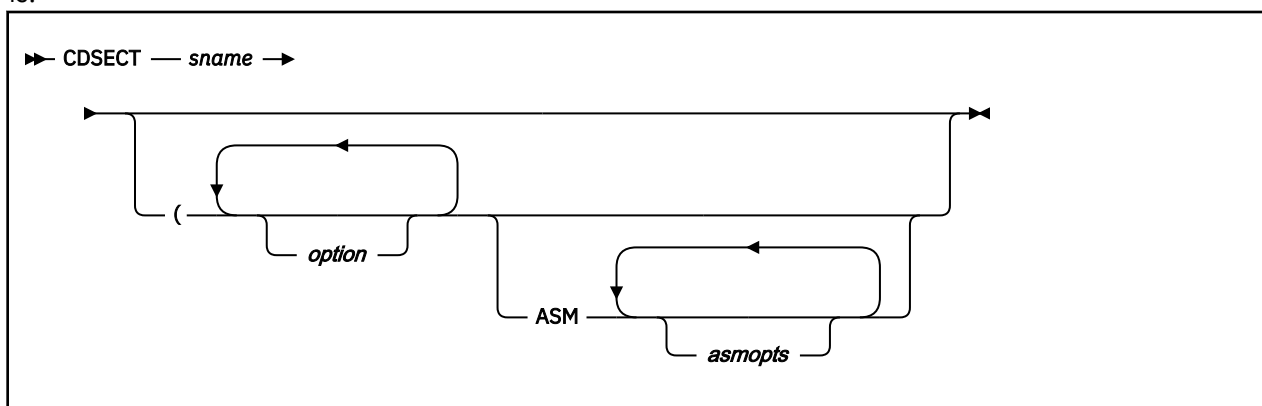
Chapter 13. DSECT Conversion Utility

This chapter describes how to use the DSECT conversion utility.

The DSECT conversion utility generates a C structure to map an assembler DSECT. This utility is used when a C program calls or is called by an Assembler program and a C structure is required to map the area passed.

The source for the assembler DSECT is assembled using the High-Level Assembler specifying the ADATA option. (For a description of the ADATA option, see the *High Level Assembler for z/OS & z/VM & z/VSE: Programmer's Guide* (https://www.ibm.com/docs/en/SSENW6_1.6.0/pdf/asmp1024_pdf.pdf.) The DSECT utility then reads the SYSADATA file produced by the High Level Assembler and produces a file containing the C structure according to the options specified.

The DSECT utility is run under VM/CMS by using the CDSECT EXEC. The syntax of the CDSECT command is:



sname

is the file name of the assembler source program containing the required section.

options

are any valid DSECT utility options.

ASM asmopts

specifies High Level Assembler options. The ADATA option is specified by default.

KNOWN: - The assembler source name is TESTASM ASSEMBLE A.
 - The required DSECT Utility options are EQU(BIT).

USE THE FOLLOWING COMMAND:
CDSECT TESTASM (EQU(BIT)

Figure 34. Running the DSECT Utility under CMS

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 is produced in file `sname DMAP A1`. The C structure produced is written to a file `sname STRUCT A1` unless the OUTPUT option is specified.

If the assembler source requires macros or copy members from a MACLIB, issue the GLOBAL MACLIB command to set up the required MACLIBs before issuing the CDSECT command.

DSECT Utility Options

The options that you can use to control the generation of the C structure are as follows. You can specify them in uppercase or lowercase, separating them by spaces or commas.

Table 8. 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
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
UNNAMED NOUNNAMED	UNN NOUNN	NOUNNAMED
OUTPUT[(<i>filename</i>)]	OUT[(<i>filename</i>)]	OUTPUT(DD:EDCDSECT)
RECFM[(<i>recfm</i>)]	None	C Library defaults
LRECL[(<i>lrecl</i>)]	None	C Library defaults
BLKSIZE[(<i>blksize</i>)]	None	C Library defaults

SECT

DEFAULT: SECT(ALL)

The SECT option specifies the section names for which C structures are to be produced. The section names can be either CSECT or DSECT names. They must exist in the SYSADATA file produced by the Assembler. If you do not specify the SECT option or if you specify SECT(ALL), C 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, the bit fields can be tested using the following code:

TM	FLAGFLD,L'B1FLG	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 is used to provide 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:

- Does not have a duplication factor of zero.
- Has a length between 1 and 4 bytes and does not have a bit length.
- Does not have more than 1 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 1 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, any EQU statements following the field are not used to define the bit fields.

The following fields are used to define the bit fields as long as they map consecutive bits. If two consecutive fields are equivalent, the second field is skipped.

COMMENT | NOCOMMENT

DEFAULT: COMMENT

The COMMENT option specifies whether the comments on the line where the field is defined will be placed in the C structure produced.

If you specify the COMMENT option without a delimiter, the entire comment is placed in the C structure.

If you specify a delimiter, any comments following the delimiter are skipped and are not placed in the C 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 comment text is not significant. You can specify up to 10 delimiters, and they can contain up to 10 characters each.

DEFSUB | NODEFSUB

DEFAULT: DEFSUB

The DEFSUB option specifies whether #define directives will be built for fields that are part of a union or substructure.

If the DEFSUB option is in effect, fields within a substructure or union have the field names prefixed by an underscore. A #define directive is written at the end of the structure to allow the field name to be specified directly as in the following example:

```
_Packed struct dsect_name {
    int      field1;
_Packed 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 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 following 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 following 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:

- Does not have a duplication factor of zero.
- Has a length between 1 and 4 bytes and has a bit length that is a multiple of 8.
- Does not have more than 1 nominal value.

and the EQU statements following the field conform to the following rules:

- The value for the EQU statements following the field mask consecutive bits (for example, X'80' followed by X'40').
- The value for an EQU statement masks one bit or consecutive bits, for example, B'10000000' or B'11000000', but not B'10100000'.
- Where the length of the field is greater than 1 byte, the bits for the remaining bytes can be defined by providing the EQU statements for the second byte after the EQU statement for the first byte.
- The value for the EQU statement is not a relocatable value.

When you specify EQUATE (BIT), the EQU statements are converted as in the following example:

```

FLAGFLD  DS   H
FLAG21   EQU  X'80'
FLAG22   EQU  X'40'
FLAG23   EQU  X'20'
FLAG24   EQU  X'10'
FLAG25   EQU  X'08'
FLAG26   EQU  X'04'
FLAG27   EQU  X'02'
FLAG28   EQU  X'01'
FLAG2A   EQU  X'80'
FLAG2B   EQU  X'40'
_Packed 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:

- Does not have a duplication factor of zero.
- Has a length between 1 and 4 bytes and has a bit length that is a multiple of 8.
- Does not have more than 1 nominal value.

and the EQU statements following the field conform to the following rules:

- The value specified for the EQU statement has the same or overlapping offset as the field.
- The length attribute for the EQU statement is between 1 and 255.
- The length attribute for the EQU statement masks one bit or consecutive bits, for example, B'10000000' or B'11000000', but not B'10100000'.
- The value for the EQU statement is a relocatable value.

When you specify EQUATE (BITL), the EQU statements are converted as in the following example:

```

BYTEFLD DS F
B1FLG1 EQU BYTEFLD+0,B'10000000'
B1FLG2 EQU BYTEFLD+0,B'01000000'
B1FLG3 EQU BYTEFLD+0,B'00100000'
B1FLG4 EQU BYTEFLD+0,B'00010000'
B1FLG5 EQU BYTEFLD+0,B'00001000'
B1FLG6 EQU BYTEFLD+0,B'00000100'
B1FLG7 EQU BYTEFLD+0,B'00000010'
B1FLG8 EQU BYTEFLD+0,B'00000001'
B2FLG1 EQU BYTEFLD+1,B'10000000'
B2FLG2 EQU BYTEFLD+1,B'01000000'
B2FLG3 EQU BYTEFLD+1,B'00100000'
B2FLG4 EQU BYTEFLD+1,B'00010000'
_Packed 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 C structure. The EQU statements should not specify a relocatable value.

When you specify EQUATE (DEF), the EQU statements are converted as in the following example:

```

FLAGBYTE DS X
FLAG1 EQU X'80'
FLAG2 EQU X'20'
FLAG3 EQU X'10'
FLAG4 EQU X'08'
FLAG5 EQU X'06'
FLAG6 EQU X'01'
_Packed 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 C structure produced.

The value specified on the HDRSKIP option indicates the number of bytes at the start of the section that are to be skipped. HDRSKIP(0) is equivalent to NOHDRSKIP.

In the following example, if you specify HDRSKIP(8), the first two fields are skipped and only the remaining two fields are built into the structure.

```
SECTIONNAME DSECT
PREFIX1 DS CL4
PREFIX2 DS CL4
FIELD1 DS CL4
FIELD2 DS CL4
_Packed 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 C 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*) option specifies the name of a locale to be passed to the setlocale() function. Specifying LOCALE without the *name* parameter is equivalent to passing the NULL string to the setlocale() function.

The structure produced contains the left and right brace, and left and right square bracket, backslash, and number sign which have different code point values for the different code pages. When the LOCALE option is specified, and these characters are written to the output file, the code point from the LC_SYNTAX category for the specified locale is used.

The default is NOLOCALE.

You can abbreviate the option to LOC(*name*) or NOLOC.

LOWERCASE | NOLOWERCASE

DEFAULT: LOWERCASE

The LOWERCASE option specifies whether the field names within the C structure are to be converted to lowercase or left as entered. If you specify LOWERCASE, all the field names are converted to lowercase. If you specify NOLOWERCASE, the field names are built into the structure in the case in which they were entered in the assembler section.

OPTFILE | NOOPTFILE

The OPTFILE(*filename*) option specifies the file name containing 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, with only one section name specified. The options following determine how the structure is produced for the specified section. The section name must only be specified once.
- The lines may contain the options BITF0XL, COMMENT, DEFSUB, EQUATE, HDRSKIP, INDENT, LOWERCASE, PPCOND, and UNNAMED, separated by spaces or commas. These override the options specified on the command line for the section.

The OPTFILE option is ignored if the SECT option is also specified on the command line.

The default is NOOPTFILE.

You can abbreviate the option to OPTF(*filename*) or NOOPTF.

PPCOND | NOPPCOND

DEFAULT: NOPPCOND

The PPCOND option specifies whether preprocessor directives will be built around the structure definition to prevent duplicate definitions.

If you specify PPCOND, the following are built around the structure definition.

```
#ifndef switch
#define switch
.
.
.
    structure definition for section
.
.
.
#endif
```

where *switch* is the switch specified on the PPCOND option or the section name prefixed and suffixed by two underscores, for example, `__name__`.

If you specify a switch, the `#ifndef` and `#endif` directives are placed around all structures that are produced. If you do not specify a switch, the `#ifndef` and `#endif` directives are placed around each structure produced.

SEQUENCE | NOSEQUENCE

DEFAULT: NOSEQUENCE

The SEQUENCE option specifies whether sequence numbers will be placed in columns 73 to 80 of the output record. If you specify the SEQUENCE option, the C 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 C 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.

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 C structures 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 C structure. You can specify any valid file name up to 60 characters in length. The file name specified will be passed to fopen() as entered.

RECFM

DEFAULT: C 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 library defaults are used.

LRECL

DEFAULT: C Library default

The LRECL option specifies the logical record length for the file to be produced. The logical record length specified must not be greater than 32767. If it is not specified, the C library defaults will be used.

BLKSIZE

DEFAULT: C Library default

The BLKSIZE option specifies the block size for the file to be produced. The block size specified must not be greater than 32767. If it is not specified, the C library defaults will be used.

Generation of C Structures

The C structure is produced as follows according to the options in effect:

- The section name is used as the structure name. The structure is generated with the `_Packed` attribute to ensure it matches the assembler section.

Whenever you specify the structure name, you should also specify the `_Packed` attribute.

- 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 structure and unions names are generated by the DSECT utility.
- 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. A filler name is generated by the DSECT utility.
- Where there is no direct equivalent for an assembler definition within the 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 following the definition of the field is placed in the C 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 C structure as shown in the following examples:

- 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 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 C 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;
```

Chapter 14. Code Set and Locale Utilities

This chapter describes the code set conversion utilities that help you convert a file from one code set to another and the `localedef` utility that allows you to define the language and cultural conventions used in your environment.

Code Set Conversion Utilities

The code set conversion facilities that you may find useful prior to compiling are:

iconv

Converts a file from one code set encoding to another. It can be used to convert C source code before compilation or to convert input files.

genxlt

Generates a translate table for use by the `iconv` utility and `iconv` functions to perform code set conversion. It can be used to build code set conversions for existing code pages that are not supplied with C, or to build code set conversions for existing code pages. The `iconv_open()`, `iconv()`, and `iconv_close()` functions are called from the `iconv` utility to perform code set translation. These functions can be called from any program requiring code set translation. For more information on these functions, see the [XL C/C++ for z/VM: Runtime Library Reference](#).

iconv Utility

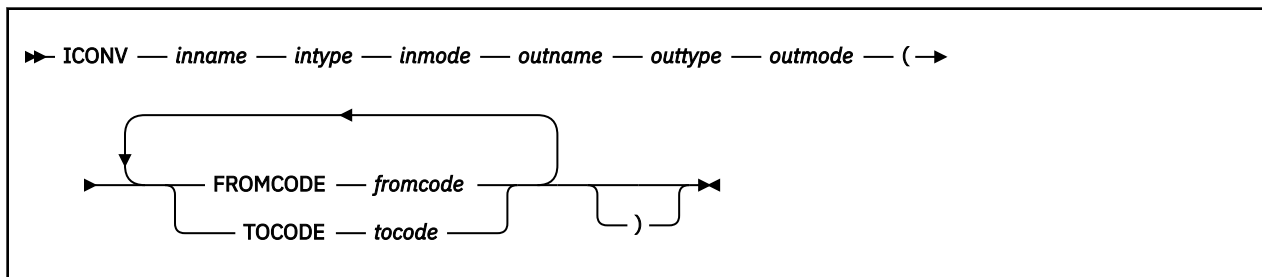
The `iconv` utility converts the characters from the input file from one coded character set (code set) definition to another code set definition, and writes the characters to the output file.

The `iconv` utility uses the `iconv_open()`, `iconv()`, and `iconv_close()` functions to convert the input file records from the coded character set definition for the input code page to the coded character set definition for the output code page. There is one record in the output file for each record in the input file. No padding or truncation of records is performed.

When conversions are performed between single-byte code pages, the output records are the same length as the input records. When conversions are performed between double-byte code pages, the output records may be longer or shorter than the input records because the shift-out and shift-in characters may be added or removed.

The ICONV EXEC is provided to invoke the `iconv` utility to copy the input file to the output file and convert the characters from the input code page to the output code page.

The syntax of the ICONV command is:



inname

is the file name of the input file.

intype

is the file type of the input file.

inmode

is the file mode of the input file.

outname

is the file name of the output file. If = is specified, the output file is the same as the input file.

outtype

is the file type of the output file. If = is specified, the output file type is the same as the input file type.

outmode

is the file mode of the output file. If = is specified, the output file mode is the same as the input file mode.

fromcode

is the name of the codeset in which the input data is encoded.

tocode

is the name of the codeset to which the output data is to be converted.

In the following example, the input file is INPUT FILE A in code page IBM-037 and the output file is OUTPUT FILE A in code page IBM-1047.

```
ICONV INPUT FILE A OUTPUT FILE A (FROMCODE IBM-037 TOCODE IBM-1047
```

Note: If the FROMCODE or TOCODE is specified more than once, the last value specified is used. The output file is created with a record format of V. For more information, see the [z/OS: XL C/C++ Programming Guide \(https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147315/\\$file/cbcpx01_v2r5.pdf\)](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147315/$file/cbcpx01_v2r5.pdf).

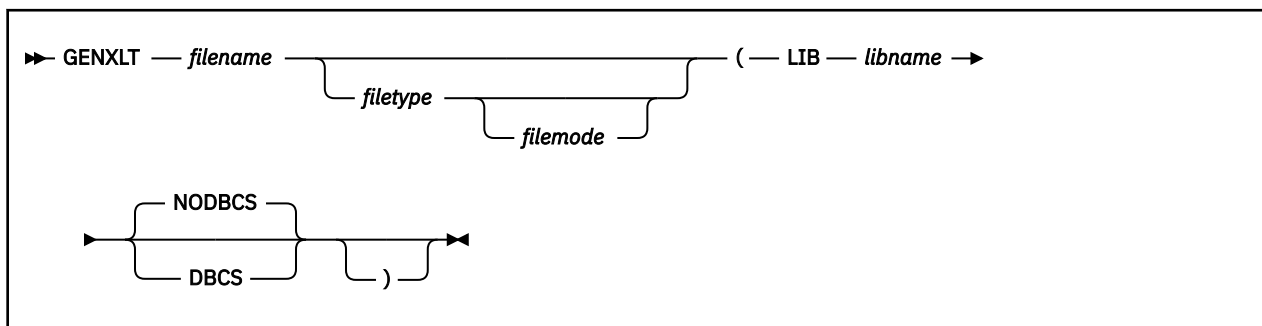
genxlt Utility

The `genxlt` utility creates translation tables, which are used by the `iconv_open()`, `iconv()`, and `iconv_close()` services of the runtime library. These services can be called from both non-XPLINK and XPLINK applications. The non-XPLINK and XPLINK versions have different names. The non-XPLINK version of the GENXLT table should always be generated. If any XPLINK applications will require one of these translation tables, then the XPLINK version should also be generated.

The `genxlt` utility reads character conversion information from the input file and writes the compiled conversion table to the LOADLIB. The input file contains directives that are acted upon by the `genxlt` utility to produce the compiled version of the conversion table. The source input to the `genxlt` utility is assumed to be implicitly specified in code page IBM-1047.

The GENXLT EXEC invokes the `genxlt` utility to read the character conversion information and produces the conversion table. It may be invoked under VM/CMS or VM batch. The `genxlt` utility options can be specified on the command line. If the same option is specified more than once, the last option specified is used.

The syntax of the GENXLT command is:

**filename**

is the file name of the file containing the character conversion information.

filetype

is the file type of the file containing the character conversion information. If it is not specified, it defaults to GENXLT.

filemode

is the file mode of the file containing the character conversion information. If it is not specified, the accessed disks are searched for the first file that matches the file name and file type.

LIB *libname*

specifies the name of the LOADLIB. The member name in the LOADLIB will be the same as *filename*.

NODBCS**DBCS**

specifies whether the DBCS characters within shift-out and shift-in characters will be converted. The DBCS option should be specified only when an EBCDIC code page is being converted 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 the NODBCS option is specified (or by default), all the characters are converted, and no checking of DBCS characters is performed.

For more information, see the *z/OS: XL C/C++ Programming Guide* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147315/\\$file/cbcpx01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147315/$file/cbcpx01_v2r5.pdf)).

The conversion table is built as a member of the loadlib specified. The member name is the same as the *filename* specified.

In the following example, the input file is EDCUEAEY GENXLT A, the library is MYLIB with the DBCS option, and the conversion is from IBM-037 to IBM-1047.

```
GENXLT EDCUEAEY GENXLT A (LIB MYLIB DBCS
```

To make the conversion table available for the `iconv` utility and `iconv_open()` function, issue the GLOBAL LOADLIB command, as follows:

```
GLOBAL LOADLIB MYLIB SCEERUN
```

localedef Utility

The `localedef` utility creates locale objects, which are used by the `setlocale()` service of the runtime library. This service can be called from both non-XPLINK and XPLINK applications. The non-XPLINK and XPLINK locale object versions have different names. The non-XPLINK version of the locale object should always be generated. If any XPLINK applications will use the locale then the XPLINK version should also be generated.

A *locale* is a collection of data that defines language and cultural conventions. Locales consist of various categories, that are identified by name, that characterize specific aspects of your cultural environment.

The `localedef` utility generates locales according to the rules that are defined in the locale definition file. A user can create his own customized locale definition file.

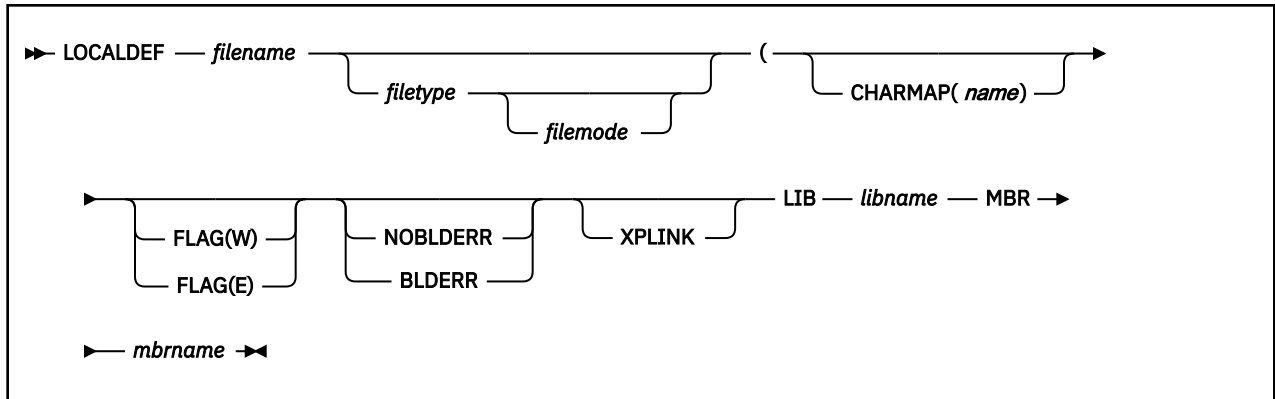
The `localedef` utility reads the locale definition file and produces a locale object that can be used by the locale specific library functions.

The LOCALDEF EXEC invokes the `localedef` utility under VM/CMS and VM batch. It does the following:

1. Invokes the CCNELDEF module to read the locale definition file and produce the C code to build the locale
2. Invokes the XL C/C++ compiler to compile the C source generated
3. Invokes the VM/CMS BIND command to build a loadlib member

The options for the `localedef` utility are specified on the command line. They can be separated by spaces or commas. If the same option is specified more than once, the last option specified is used.

The syntax of the LOCALDEF command is:



filename

is the file name of the file containing the locale definition information.

filetype

is the file type of the file containing the locale definition information. If it is not specified, it defaults to LOCALE.

filemode

is the file mode of the file containing the locale definition information. If it is not specified, the accessed disks are searched for the first file that matches the file name and file type.

CHARMAP(name)

specifies the member name of the file containing the mapping of the character symbols to actual character encodings. If this option is not specified, the `localdef` utility defaults the Charmap to IBM-1047.

The name specified is the file name of the charmap file. The file type is CHARMAP.

FLAG(W)

FLAG(E)

specifies whether warning messages are issued. If FLAG(W) is specified (or by default), warning and error messages are issued. If FLAG(E) is specified, only the error messages are issued.

NOBLDERR

BLDERR

specifies whether the locale is generated if errors are detected. If the BLDERR option is specified, the locale is generated even if errors are detected. If the NOBLDERR option is specified (or by default), the locale is not generated if an error is detected.

XPLINK

specifies that the locale to be built is an XPLINK locale.

libname

is the libname parameter of the LIB option that specifies the name of the LOADLIB.

mbrname

is the mbrname parameter of the MBR option that specifies the member name for the member in the LOADLIB. The member name defaults to the file name of the input file.

The LOADLIB member is created using the BIND command. The member name within the LOADLIB is the member name (if specified) or the file name of the input file. 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.

For more information on locale and code set codes, see the *z/OS: XL C/C++ Programming Guide* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147315/\\$file/cbcpx01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc147315/$file/cbcpx01_v2r5.pdf)).

In the following example, the locale source is EDC\$EUEY LOCALE A, the library name is MYLIB, options are CHARMAP(IBM-297), and the output member name is EDC\$EUEM, for EN_US. IBM-297.

```
LOCALDEF EDC$EUEY LOCALE A (LIB MYLIB CHARMAP(IBM-297) MBR EDC$EUEM
```

Chapter 15. OpenExtensions ar and make Utilities

OpenExtensions provides two utilities that you can use to make the task of creating and managing OpenExtensions C/C++ application programs easier: `ar` and `make`. Use these utilities with the `c89/cxx` utility to build an application program into an easily updated and maintained executable file.

Note: All references to `c89` in the following sections also apply to `cxx` unless otherwise specified.

OpenExtensions Archive Libraries

The `ar` utility allows you to create and maintain a library of OpenExtensions C/C++ application object files. You can specify the `c89` command string so that archive libraries are processed during 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. A `c89`-specified archive library is processed during binding. Any object files in the specified archive library will be bound if they can be used to resolve external symbols. Use of this autocall library mechanism by the `c89` utility is analogous to the use of the `C370LIB` Object Library utility for `z/VM` application program objects. For more information, see [Chapter 11, “Object Library Utility,”](#) on page 73.

The `c89` utility requires that archive libraries obey the following naming convention in the byte file system (BFS):

```
filename.a
```

This assumes that no directory file searching for the archive file takes place when specified on the `c89` command line. For example, to compile the application program source file `dirsum.c` from the `src` subdirectory of your working directory and resolve external symbols from the `symb.a` archive library in your working directory, you would enter:

```
c89 -o ./exfls/dirsum ./src/dirsum.c ./symb.a
```

To use `c89` to search for specified archive files in one or more BFS directories, use the naming convention:

```
liblibname.a
```

On the `c89` command line, specify BFS directories to be searched with the `-L directory` option and an archive library with the `-l libname` operand. For example, to compile the application program source file `entinfo.c` from the `src` subdirectory of your working directory and bind it with the object file `newsroute.o` and the archive file `/mylib/libbrwobjs.a`, enter:

```
c89 -o ./entinfo -L /mylib ./src/entinfo.c newsroute.o -l brwobjs
```

The BFS subdirectory `mylib` of your working directory is searched first for the archive library `libbrwobjs.a`. If it is not found there, `c89` searches for the archive library in the usual places.

Creating Archive Libraries

To create the archive library, use the `ar -r` option. For example, to create an archive library named `bin/libbrobompgm.a` from your working directory and add the member `jkeyadd.o` to it, specify:

```
ar -rc ./bin/libbrobompgm.a jkeyadd.o
```

The `libbrobompgm.a` archive library file is created in the `bin` subdirectory of your BFS working directory. Use of the `-c` option tells `ar` to suppress the message normally sent when an archive library file is created.

To display the object files archived in the `bin/libbrobompgm.a` library from your working directory, specify:

```
ar -t ./bin/libbrobompgm.a
```

For more information about the `ar` utility, see the [z/VM: OpenExtensions Commands Reference](#).

Creating Makefiles

The `make` utility maintains all the parts of and dependencies for your application program. It uses a *makefile*, which you create, to keep your application parts (listed in it) up to date with one another. If one part changes, `make` updates all the other files that depend on the changed part.

A *makefile* is a normal BFS text file. Create the file and edit it using any text editor to describe the application program files, their locations, dependencies on other files, and rules for building the files into an application 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 invokes the `c89` interface to the XL C/C++ compiler and the binder to recompile and bind an updated application program.

For a detailed discussion of the `make` utility and how best take advantage of its function, see the [z/VM: OpenExtensions Commands Reference](#) and [z/VM: OpenExtensions Advanced Application Programming Tools](#).

Appendix A. IBM-Supplied EXECs

This appendix lists the EXECs provided by the XL C/C++ compiler, in conjunction with Language Environment, to call the various utilities. For more information on the EXECs provided by Language Environment, see the *z/OS: Language Environment Programming Guide* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380682/\\$file/ceea200_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380682/$file/ceea200_v2r5.pdf)).

EXEC Name	Task Description
CC	Compile
CMOD	Generate an executable module
CXXFILT	Demangle names
GENXLT	Generate a translate table for use by the ICONV utility and functions
ICONV	Convert a file from one code set encoding to another
LOCALDEF	Produce a locale object that can be used by the locale specific library functions
LINKLOAD	Generate a fetchable module
C370LIB	Maintain an object library TXTLIB
CDSECT	Run the DSECT Conversion Utility

Appendix B. XL C/C++ Compiler Return Codes and Messages

For complete descriptions of XL C/C++ return codes and messages, see *z/OS: XL C/C++ Messages* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5gc147305/\\$file/cbcdg01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5gc147305/$file/cbcdg01_v2r5.pdf)).

Appendix C. EXEC Error Messages

The messages in this section can be returned from the following XL C/C++ EXECs:

- CC
- CDSECT
- LOCALDEF

The message format is:

CCNUTLnnns *text* [&n]

nnn

is the message number.

s

is the message type (severity):

I

Informational

W

Warning

E

Error

text

is the message that appears on the screen.

&n

is a substitution variable, which contains a specific name in the issued message.

CCNUTL001I &1 exec completed with return code &2.

Explanation

The utility completed processing with the return code specified.

User response

No response required.

CCNUTL002E Help is not available.

Explanation

The help file for the requested command is not accessible or does not exist.

User response

Find out from your systems programmer which disk has the help file on it and get access to the disk. If the help file has not been installed, have your systems programmer install it.

CCNUTL003W LE Run-Time library SCEERUN is not in GLOBAL LOADLIB.

Explanation

The runtime library is missing.

User response

Run the z/VM command GLOBAL LOADLIB SCEERUN to add the runtime library.

CCNUTL004W Invalid parameter list.

Explanation

The parameter list is not valid.

User response

Check the syntax of the command you are running and correct it.

CCNUTL005E A-Disk is not accessed.

Explanation

Your A-disk is not accessed in read/write mode.

User response

Link to and access your A-disk in read/write mode.

CCNUTL006E A-Disk is not writable.

Explanation

Your A-disk is not accessed in read/write mode.

User response

Link to and access your A-disk in read/write mode.

**CCNUTL008E A library name must be specified
in suboption LIB.**

Explanation

You must specify a library name when using
LOCALDEF command.

User response

Check the syntax of the LOCALDEF command and
correct it.

**CCNUTL009E Cannot execute program module
&1.**

Explanation

A module cannot be run.

User response

Check with your system programmer.

Appendix D. Runtime Error Messages and Return Codes

This appendix contains information about the runtime messages and should not be used as programming interface information.

These are messages you see while your XL C/C++ program is running. Messages may be displayed in uppercase or in mixed case English format, or in Kanji.

pererror Messages

These messages are only printed when a call to `pererror` or `sterror` is made and the `errno` value does not prefix the message.

Note: For information about these messages, see *z/OS: Language Environment Runtime Messages* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380686/\\$file/ceea900_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380686/$file/ceea900_v2r5.pdf)).

XL C/C++ Runtime Return Codes

The runtime return code value is set in one of the following ways:

- By the initialization and termination routines or the program management routines of Language Environment.
- By the `return` statement in your XL C/C++ program
- By calling the `exit` or `abort` functions from your XL C/C++ program.

It is possible to pass a return code from an XL C/C++ program to the program that invoked it. For example, if the XL C/C++ program is invoked by a REXX exec, it can examine the return code to determine if processing should continue.

The return code generated by an XL C/C++ program consists of two elements. One element is specified if the program calls the `exit` function or if the program specifies a return value when returning from `main`. The other element is specified by the program management routines of the Language Environment library and indicates the way in which your program terminated. Unless an error is detected that prevents the program management routines from operating correctly, the two elements are added together to form a total in which the thousands digit indicates the way in which your program terminated and the hundreds, tens, and units are set by your program.

Valid return codes are -2^{31} to $2^{31}-1$, inclusive.

Note: The CMS Ready(*nnnnn*) prompt displays only the last 5 digits of the return code. For example, 2,000,000 is displayed as Ready (00000). You can write a REXX exec to retrieve the full return code.

For a list of error messages, see *z/OS: Language Environment Runtime Messages* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380686/\\$file/ceea900_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380686/$file/ceea900_v2r5.pdf)).

Appendix E. Utility Messages

This appendix contains information about the DSECT utility messages.

See for messages and return codes for:

- Object Library Utility
- Runtime messages and return codes
- localedef Utility
- genxlt Utility
- iconv Utility

See *z/OS: XL C/C++ Messages* ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5gc147305/\\$file/cbcdg01_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5gc147305/$file/cbcdg01_v2r5.pdf)) for messages and return codes for the CXXFILT utility.

DSECT Utility Messages

Return Codes

Table 9. Return Codes from the DSECT Utility

Return Code	Meaning
0	Successful completion.
4	Successful completion, warnings issued.
8	DSECT Utility failed, error messages issued.
12	DSECT Utility failed, severe error messages issued.
16	DSECT Utility failed, insufficient storage to continue processing.

Messages

The messages issued by the DSECT utility have the format:

```
EDCnnnn ss text [&n]
```

nnnn

is the message number.

ss

is the message type (severity):

00

Informational

10 or E

Error warning

30

Error

40

Severe error

text

is the message that appears on the screen.

&n

is a substitution variable, which contains a specific name in the issued message.

EDC5500 10 Option &1 is not valid and is ignored.

Explanation

The option specified in the message is not valid DSECT Utility option or a valid option has been specified with an invalid value. The specified option is ignored.

User response

Rerun the DSECT Utility with the correct option.

EDC5501 30 No DSECT or CSECT names were found in the SYSADATA file.

Explanation

The SECT option was not specified or SECT(ALL) was specified. The SYSADATA was searched for all DSECTs and CSECTs but no DSECTs or CSECTs were found.

User response

Rerun the DSECT Utility with a SYSADATA file that contains the required DSECT or CSECT definition.

EDC5502 30 Sub option &1 for option &2 is too long.

Explanation

The sub option specified for the option was too long and is ignored.

EDC5503 30 Section name &1 was not found in SYSADATA File.

Explanation

The section name specified with the SECT option was not found in the External Symbol records in the SYSADATA file. The C structure is not produced.

User response

Rerun the DSECT Utility with a SYSADATA file that contains the required DSECT or CSECT definition.

EDC5504 30 Section name &1 is not a DSECT or CSECT.

Explanation

The section name specified with the SECT option is not a DSECT or CSECT. Only a DSECT or CSECT names may be specified. The C structure is not produced.

EDC5505 00 No fields were found for section &1, structure is not produced.

Explanation

No field records were found in the SYSADATA file that matched the ESDID of the specified section name. The C structure is not produced.

EDC5506 30 Record length for file "&1" is too small for the SEQUENCE option, option ignored.

Explanation

The record length for the output file specified is too small to enable the SEQUENCE option to generate the sequence number in columns 73 to 80. The available record length must be greater than or equal to 80 characters. The SEQUENCE option is ignored.

EDC5507 40 Insufficient storage to continue processing.

Explanation

No further storage was available to continue processing.

User response

Rerun the DSECT Utility with a larger virtual machine (CMS).

EDC5508 30 Open failed for file "&1": &2

Explanation

This message is issued if the open fails for any file required by the DSECT Utility. The file name passed to fopen() and the error message returned by strerror(errno) is included in the message.

User response

The message text indicates the cause of the error. If the file name was specified incorrectly on the OUTPUT option, rerun the DSECT Utility with the correct file name.

EDC5509 40 &1 failed for file "&2": &3

Explanation

This message is issued if any error occurs reading, writing or positioning on any file by the DSECT Utility. The name of the function that failed (Read,

Write, fgetpos, fsetpos), file name and text from strerror(errno) is included in the message.

User response

This message may be issued if an error occurs reading or writing to a file. This may be caused by an error within the file, such as an I/O error or insufficient disk space. Correct the error and rerun the DSECT Utility.

EDC5510	40 Internal Logic error in function &1
----------------	---

Explanation

The DSECT Utility has detected that an error has occurred while generating the C structure. Processing is terminated and the C structure is not produced.

User response

This may be caused by an error in the DSECT Utility or by incorrect input in the SYSADATA file. Contact your systems administrator.

EDC5511	10 No matching right parenthesis for &1 option.
----------------	--

Explanation

The option specified had a sub option beginning with a left parenthesis but no right parenthesis was present.

User response

Rerun the DSECT Utility with the parenthesis for the option correctly paired.

EDC5512	10 No matching quote for &1 option.
----------------	--

Explanation

The OUTPUT option has a sub option beginning with a single quote but no matching quote was found.

User response

Rerun the DSECT Utility with the quotes for the option correctly paired.

EDC5513	10 Record length too small for file "&1".
----------------	--

Explanation

The record length for the Output file specified is less than 10 characters in length. The minimum available record length must be at least 10 characters.

User response

Rerun the DSECT Utility with an output file with a available record length of at least 10 characters.

EDC5514	30 Too many sub options were specified for option &1.
----------------	--

Explanation

More than the maximum number of sub options were specified for the particular option. The extra sub options are ignored.

EDC5515	00 HDRSKIP option value greater than length for section &1, structure is not produced.
----------------	---

Explanation

The value specified for the HDRSKIP option was greater than the length of the section. A structure was not produced for the specified section.

User response

Rerun the DSECT Utility with a smaller value for the HDRSKIP option.

EDC5516	10 SECT and OPTFILE options are mutually exclusive, OPTFILE option is ignored
----------------	--

Explanation

Both the SECT and OPTFILE options were specified, but the options are mutually exclusive.

User response

Rerun the DSECT Utility with either the SECT or OPTFILE option.

EDC5517	10 Line &1 from "&2" does not begin with SECT option
----------------	---

Explanation

The line from the file specified on the OPTFILE option did not begin with the SECT option. The line was ignored.

User response

Rerun the DSECT Utility without OPTFILE option, or correct the line in the input file.

EDC5518	10 setlocale() failed for locale name "&1".
----------------	--

Explanation

The `setlocale()` function failed with the locale name specified on the `LOCALE` option. The `LOCALE` option was ignored.

User response

Rerun the DSECT Utility without `LOCALE` option, or correct the locale name specified with the `LOCALE` option.

Appendix F. Layout of the Events File

This appendix specifies the layout of the SYSEVENT file. The SYSEVENT file contains error information and source file statistics. Use the EVENTS compiler option to produce the SYSEVENT file. For more information on the EVENTS compiler option, see [“EVENTS | NOEVENTS”](#) on page 26.

In the following example, the source file SIMPLE C is compiled with the EVENTS (EGEVENT FILE) compiler option. The file ERR H is a header file that is included in SIMPLE C. [Figure 37 on page 115](#) 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 35. SIMPLE C

```
1  add some;
2  errors in the header file;
```

Figure 36. ERR H

```
----- start simple.events -----
FILEID 0 1 0 13 'SIMPLE C A1'
FILEID 0 2 1 8 ERR H A1
ERROR 0 2 1 0 1 1 1 8 CCN3166 E 12 48 Definition of function add requires parentheses.
FILEEND 0 2 2
ERROR 0 2 1 0 1 5 2 8 CCN3276 E 12 35 Syntax error: possible missing '{'?
ERROR 0 1 1 0 3 4 3 27 CCN3045 E 12 26 Undeclared identifier add.
ERROR 0 1 1 0 5 9 5 18 CCN3277 E 12 42 Syntax error: possible missing ';' or ','?
ERROR 0 1 1 0 5 4 5 18 CCN3045 E 12 27 Undeclared identifier here.
FILEEND 0 1 6
----- end simple.events -----
```

Figure 37. Sample SYSEVENT file

There are three different record types generated in the event file:

- FILEID
- FILEEND
- ERROR

FILEID Field

The following is an example of the FILEID field from the sample SYSEVENT file that is shown in [Figure 37 on page 115](#). [Table 10 on page 116](#) describes the FILEID identifiers.

```
FILEID 0 1 0 13 'SIMPLE C A1'
      A B C D E
```

Table 10. 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.

FILEEND Field

The following is an example of the FILEEND field from the sample SYSEVENT file that is shown in [Figure 37](#) on page 115. [Table 11](#) on page 116 describes the FILEEND identifiers.

```
FILEEND 0 1 6
        A B C
```

Table 11. 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.

ERROR Field

The following is an example of the ERROR field from the sample SYSEVENT file that is shown in [Figure 37](#) on page 115. [Table 12](#) on page 116 describes the ERROR identifiers.

```
ERROR 0 1 1 0 3 4 3 27 CCN3045 E 12 26 Undeclared identifier add.
      A B C D E F G H I       J K L M
```

Table 12. 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.
E	Starting line number	The source line number for which the message was issued. A value of 0 indicates the message was not associated with a line number.
F	Starting column number	The column number or position within the source line for which the message was issued. A value of 0 indicates the message is not associated with a line number.

Table 12. Explanation of the ERROR Field Layout (continued)

Column	Identifier	Description
G	Reserved	Do not build a dependency on this identifier. It is reserved for future use.
H	Reserved	Do not build a dependency on this identifier. It is reserved for future use.
I	Message identifier	String Containing the message identifier.
J	Message severity character	I=Informational W=Warning E=Error S=Severe U=Unrecoverable
K	Message severity number	Return code associated with the message.
L	Message length	Length of message text.
M	Message text	String containing message text.

Notices

This information was developed for products and services offered in the US. This material might be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

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 grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, Japan

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 jurisdictions 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 websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing
IBM Corporation
North Castle Drive, MD-NC119
Armonk, NY 10504-1785
US

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 document 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.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information may contain sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Programming Interface Information

This publication documents intended Programming Interfaces that allow the customer to write programs to obtain the services of IBM XL C/C++ for z/VM and IBM z/VM.

Trademarks

IBM, the IBM logo, and ibm.com[®] are trademarks or registered trademarks of International Business Machines Corp., in the United States and/or other countries. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on [IBM Copyright and trademark information](https://www.ibm.com/legal/copytrade) (<https://www.ibm.com/legal/copytrade>).

Adobe is either a registered trademark or a trademark of Adobe Systems Incorporated in the United States, and/or other countries.

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

Terms and Conditions for Product Documentation

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the IBM website.

Personal Use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of IBM.

Commercial Use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of IBM.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

IBM Online Privacy Statement

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see:

- The section entitled **IBM Websites** at [IBM Privacy Statement](https://www.ibm.com/privacy) (<https://www.ibm.com/privacy>)
- [Cookies and Similar Technologies](https://www.ibm.com/privacy#Cookies_and_Similar_Technologies) (https://www.ibm.com/privacy#Cookies_and_Similar_Technologies)

Bibliography

This topic lists the publications in the z/VM library. For abstracts of the z/VM publications, see [z/VM: General Information](#).

Where to Get z/VM Information

The current z/VM product documentation is available in [IBM Documentation - z/VM \(https://www.ibm.com/docs/en/zvm\)](https://www.ibm.com/docs/en/zvm).

z/VM Base Library

Overview

- [z/VM: License Information](#), GI13-4377
- [z/VM: General Information](#), GC24-6286

Installation, Migration, and Service

- [z/VM: Installation Guide](#), GC24-6292
- [z/VM: Migration Guide](#), GC24-6294
- [z/VM: Service Guide](#), GC24-6325
- [z/VM: VMSES/E Introduction and Reference](#), GC24-6336

Planning and Administration

- [z/VM: CMS File Pool Planning, Administration, and Operation](#), SC24-6261
- [z/VM: CMS Planning and Administration](#), SC24-6264
- [z/VM: Connectivity](#), SC24-6267
- [z/VM: CP Planning and Administration](#), SC24-6271
- [z/VM: Getting Started with Linux on IBM Z](#), SC24-6287
- [z/VM: Group Control System](#), SC24-6289
- [z/VM: I/O Configuration](#), SC24-6291
- [z/VM: Running Guest Operating Systems](#), SC24-6321
- [z/VM: Saved Segments Planning and Administration](#), SC24-6322
- [z/VM: Secure Configuration Guide](#), SC24-6323

Customization and Tuning

- [z/VM: CP Exit Customization](#), SC24-6269
- [z/VM: Performance](#), SC24-6301

Operation and Use

- [z/VM: CMS Commands and Utilities Reference](#), SC24-6260
- [z/VM: CMS Primer](#), SC24-6265
- [z/VM: CMS User's Guide](#), SC24-6266
- [z/VM: CP Commands and Utilities Reference](#), SC24-6268

- [*z/VM: System Operation*](#), SC24-6326
- [*z/VM: Virtual Machine Operation*](#), SC24-6334
- [*z/VM: XEDIT Commands and Macros Reference*](#), SC24-6337
- [*z/VM: XEDIT User's Guide*](#), SC24-6338

Application Programming

- [*z/VM: CMS Application Development Guide*](#), SC24-6256
- [*z/VM: CMS Application Development Guide for Assembler*](#), SC24-6257
- [*z/VM: CMS Application Multitasking*](#), SC24-6258
- [*z/VM: CMS Callable Services Reference*](#), SC24-6259
- [*z/VM: CMS Macros and Functions Reference*](#), SC24-6262
- [*z/VM: CMS Pipelines User's Guide and Reference*](#), SC24-6252
- [*z/VM: CP Programming Services*](#), SC24-6272
- [*z/VM: CPI Communications User's Guide*](#), SC24-6273
- [*z/VM: ESA/XC Principles of Operation*](#), SC24-6285
- [*z/VM: Language Environment User's Guide*](#), SC24-6293
- [*z/VM: OpenExtensions Advanced Application Programming Tools*](#), SC24-6295
- [*z/VM: OpenExtensions Callable Services Reference*](#), SC24-6296
- [*z/VM: OpenExtensions Commands Reference*](#), SC24-6297
- [*z/VM: OpenExtensions POSIX Conformance Document*](#), GC24-6298
- [*z/VM: OpenExtensions User's Guide*](#), SC24-6299
- [*z/VM: Program Management Binder for CMS*](#), SC24-6304
- [*z/VM: Reusable Server Kernel Programmer's Guide and Reference*](#), SC24-6313
- [*z/VM: REXX/VM Reference*](#), SC24-6314
- [*z/VM: REXX/VM User's Guide*](#), SC24-6315
- [*z/VM: Systems Management Application Programming*](#), SC24-6327
- [*z/VM: z/Architecture Extended Configuration \(z/XC\) Principles of Operation*](#), SC27-4940

Diagnosis

- [*z/VM: CMS and REXX/VM Messages and Codes*](#), GC24-6255
- [*z/VM: CP Messages and Codes*](#), GC24-6270
- [*z/VM: Diagnosis Guide*](#), GC24-6280
- [*z/VM: Dump Viewing Facility*](#), GC24-6284
- [*z/VM: Other Components Messages and Codes*](#), GC24-6300
- [*z/VM: VM Dump Tool*](#), GC24-6335

z/VM Facilities and Features

Data Facility Storage Management Subsystem for z/VM

- [*z/VM: DFSMS/VM Customization*](#), SC24-6274
- [*z/VM: DFSMS/VM Diagnosis Guide*](#), GC24-6275
- [*z/VM: DFSMS/VM Messages and Codes*](#), GC24-6276
- [*z/VM: DFSMS/VM Planning Guide*](#), SC24-6277

- [*z/VM: DFSMS/VM Removable Media Services*](#), SC24-6278
- [*z/VM: DFSMS/VM Storage Administration*](#), SC24-6279

Directory Maintenance Facility for z/VM

- [*z/VM: Directory Maintenance Facility Commands Reference*](#), SC24-6281
- [*z/VM: Directory Maintenance Facility Messages*](#), GC24-6282
- [*z/VM: Directory Maintenance Facility Tailoring and Administration Guide*](#), SC24-6283

Open Systems Adapter

- [Open Systems Adapter-Express Customer's Guide and Reference \(https://www.ibm.com/support/pages/node/6019492\)](https://www.ibm.com/support/pages/node/6019492), SA22-7935
- [Open Systems Adapter-Express Integrated Console Controller User's Guide \(https://www.ibm.com/support/pages/node/6019810\)](https://www.ibm.com/support/pages/node/6019810), SC27-9003
- [Open Systems Adapter-Express Integrated Console Controller 3215 Support \(https://www.ibm.com/docs/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.ioa/ioa.htm\)](https://www.ibm.com/docs/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.ioa/ioa.htm), SA23-2247
- [Open Systems Adapter/Support Facility on the Hardware Management Console \(https://www.ibm.com/docs/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.ioa/ioa.htm\)](https://www.ibm.com/docs/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.ioa/ioa.htm), SC14-7580

Performance Toolkit for z/VM

- [*z/VM: Performance Toolkit Guide*](#), SC24-6302
- [*z/VM: Performance Toolkit Reference*](#), SC24-6303

RACF® Security Server for z/VM

- [*z/VM: RACF Security Server Auditor's Guide*](#), SC24-6305
- [*z/VM: RACF Security Server Command Language Reference*](#), SC24-6306
- [*z/VM: RACF Security Server Diagnosis Guide*](#), GC24-6307
- [*z/VM: RACF Security Server General User's Guide*](#), SC24-6308
- [*z/VM: RACF Security Server Macros and Interfaces*](#), SC24-6309
- [*z/VM: RACF Security Server Messages and Codes*](#), GC24-6310
- [*z/VM: RACF Security Server Security Administrator's Guide*](#), SC24-6311
- [*z/VM: RACF Security Server System Programmer's Guide*](#), SC24-6312
- [*z/VM: Security Server RACROUTE Macro Reference*](#), SC24-6324

Remote Spooling Communications Subsystem Networking for z/VM

- [*z/VM: RSCS Networking Diagnosis*](#), GC24-6316
- [*z/VM: RSCS Networking Exit Customization*](#), SC24-6317
- [*z/VM: RSCS Networking Messages and Codes*](#), GC24-6318
- [*z/VM: RSCS Networking Operation and Use*](#), SC24-6319
- [*z/VM: RSCS Networking Planning and Configuration*](#), SC24-6320

TCP/IP for z/VM

- [*z/VM: TCP/IP Diagnosis Guide*](#), GC24-6328
- [*z/VM: TCP/IP LDAP Administration Guide*](#), SC24-6329
- [*z/VM: TCP/IP Messages and Codes*](#), GC24-6330

- *z/VM: TCP/IP Planning and Customization*, SC24-6331
- *z/VM: TCP/IP Programmer's Reference*, SC24-6332
- *z/VM: TCP/IP User's Guide*, SC24-6333

Prerequisite Products

Device Support Facilities

- Device Support Facilities (ICKDSF): User's Guide and Reference ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5gc350033/\\$file/ickug00_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5gc350033/$file/ickug00_v2r5.pdf)), GC35-0033

Environmental Record Editing and Printing Program

- Environmental Record Editing and Printing Program (EREP): Reference ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5gc350152/\\$file/ifc2000_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5gc350152/$file/ifc2000_v2r5.pdf)), GC35-0152
- Environmental Record Editing and Printing Program (EREP): User's Guide ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5gc350151/\\$file/ifc1000_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5gc350151/$file/ifc1000_v2r5.pdf)), GC35-0151

Related Products

z/OS

- *Common Programming Interface Communications Reference* (<https://publibfp.dhe.ibm.com/epubs/pdf/c2643999.pdf>), SC26-4399
- z/OS and z/VM: Hardware Configuration Definition Messages ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc342668/\\$file/cbdm100_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc342668/$file/cbdm100_v2r5.pdf)), SC34-2668
- z/OS and z/VM: Hardware Configuration Manager User's Guide ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc342670/\\$file/eequ100_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sc342670/$file/eequ100_v2r5.pdf)), SC34-2670
- z/OS: Network Job Entry (NJE) Formats and Protocols ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa320988/\\$file/hasa600_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa320988/$file/hasa600_v2r5.pdf)), SA32-0988
- z/OS: IBM Tivoli Directory Server Plug-in Reference for z/OS ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa760169/\\$file/glpa300_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa760169/$file/glpa300_v2r5.pdf)), SA76-0169
- z/OS: Language Environment Concepts Guide ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380687/\\$file/ceea800_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380687/$file/ceea800_v2r5.pdf)), SA38-0687
- z/OS: Language Environment Debugging Guide ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5ga320908/\\$file/ceea100_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5ga320908/$file/ceea100_v2r5.pdf)), GA32-0908
- z/OS: Language Environment Programming Guide ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380682/\\$file/ceea200_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380682/$file/ceea200_v2r5.pdf)), SA38-0682
- z/OS: Language Environment Programming Reference ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380683/\\$file/ceea300_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380683/$file/ceea300_v2r5.pdf)), SA38-0683
- z/OS: Language Environment Runtime Messages ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380686/\\$file/ceea900_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380686/$file/ceea900_v2r5.pdf)), SA38-0686
- z/OS: Language Environment Writing Interlanguage Communication Applications ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380684/\\$file/ceea400_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa380684/$file/ceea400_v2r5.pdf)), SA38-0684
- z/OS: MVS Program Management Advanced Facilities ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa231392/\\$file/ieab200_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa231392/$file/ieab200_v2r5.pdf)), SA23-1392
- z/OS: MVS Program Management User's Guide and Reference ([https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa231393/\\$file/ieab100_v2r5.pdf](https://www.ibm.com/servers/resourcelink/svc00100.nsf/pages/zosv2r5sa231393/$file/ieab100_v2r5.pdf)), SA23-1393

XL C++ for z/VM

- [XL C/C++ for z/VM: Runtime Library Reference](#), SC09-7624
- [XL C/C++ for z/VM: User's Guide](#), SC09-7625

Index

A

- absolute file names [49](#)
- application programs
 - using makefiles to maintain [102](#)
- ar utility
 - creating archive libraries [101](#)
 - maintaining program objects [101](#)
- ARCHITECTURE compiler option [23](#)
- archive libraries
 - ar utility [101](#)
 - creating [101](#)
 - displaying the object files in [101](#)
 - file naming convention for c89/cxx use [101](#)
- assembler
 - generation of C structures [90](#)

B

- BFS files
 - definition xi
- BFS input files, compiler [43](#)
- binder, interface to c89/cxx utility [69](#)
- BITFOXL DSECT utility option [84](#)
- BLKSIZE DSECT utility option [90](#)

C

- C++ compiler listing [33](#)
- C370LIB
 - directory [73](#)
 - EXEC
 - FILENAME option [74](#)
 - syntax of [73](#)
- c89/cxx utility [69](#)
- CC EXEC
 - error messages returned by [107](#)
 - specifying BFS input file [43](#)
 - specifying CMS input file [42](#)
 - specifying compiler options [44](#)
 - syntax [41](#)
- CCNnnnn messages [105](#)
- CCNUTL001I [107](#)
- CCNUTL002E [107](#)
- CCNUTL003W [107](#)
- CCNUTL004W [107](#)
- CCNUTL005E [107](#)
- CCNUTL006E [108](#)
- CCNUTL008E [108](#)
- CCNUTL009E [108](#)
- CCNUTLnnnx messages [107](#)
- CDSECT EXEC [83](#)
- CEEBINT HLL user exit, using to set emsg [64](#)
- CEEUOPT CSECT, creating [70](#)
- CEEXOPT macro [70](#)
- CLASSNAME filter utility option [80](#)

- CMOD EXEC
 - examples [59](#)
 - options [57](#)
- CMS files
 - definition xi
- CMS input files, compiler [42](#)
- code set conversion utilities
 - genxlt
 - usage [95](#)
 - iconv
 - usage [95](#)
- command line parameter string [63](#)
- COMMENT DSECT utility option [85](#)
- compiler
 - C compiler listing [33](#)
 - c89/cxx utility interface to [65](#)
 - error messages [105](#)
 - input [42](#)
 - output [45](#)
 - return codes [105](#)
- compiler options
 - compiler options
 - #pragma options [21](#)
 - overriding defaults [21](#)
 - defaults [21](#)
 - operational differences
 - ARCHITECTURE [23](#)
 - CSECT/NOCSECT [23](#)
 - DEBUG/NODEBUG [24](#)
 - ENUMSIZE [25](#)
 - EVENTS/NOEVENTS [26](#)
 - INLPRT/NOINLPRT [26](#)
 - LIST/NOLIST [27](#)
 - LSEARCH/NOLSEARCH [27](#)
 - OBJECT/NOOBJECT [29](#)
 - OPTFILE/NOOPTFILE [30](#)
 - PPONLY/NOPONLY [31](#)
 - SEARCH/NOSEARCH [32](#)
 - SOURCE/NOSOURCE [32](#)
 - unsupported [22](#)
- compiling and binding using c89/cxx [66](#)
- compiling and binding using make [67](#)
- constructed reentrancy [62](#)
- conventions
 - default names in XL C/C++
 - xi
- CSECT (control section)
 - CEEUOPT [70](#)
 - compiler option [23](#)
- CSECT compiler option [23](#)
- CXXFILT EXEC
 - CLASSNAME option [80](#)
 - error messages [111](#)
 - REGULARNAME option [80](#)
 - running under VM/CMS [80](#)

- CXXFILT EXEC (*continued*)
 - SIDEBYSIDE option [80](#)
 - SPECIALNAME option [80](#)
 - SYMMAP option [79](#)
 - unknown type of name [80](#)
 - using [79](#)
 - WIDTH option [80](#)

D

- DCSS (discontiguous saved segment) [41](#)
- ddname
 - definition [xi](#)
 - include files [49](#)
- DEBUG compiler option [24](#)
- default, overriding compiler options [21](#)
- DEFSUB DSECT utility option [85](#)
- disk search sequence
 - include files [52](#)
 - LSEARCH compiler option [27](#)
 - SEARCH compiler option [32](#)
- DSECT utility
 - BITFOXL option [84](#)
 - BLKSIZE option [90](#)
 - COMMENT option [85](#)
 - DEFSUB option [85](#)
 - EQUATE option [86](#)
 - error messages [111](#)
 - HDRSKIP option [88](#)
 - INDENT option [88](#)
 - LOCALE option [88](#)
 - LOWERCASE option [88](#)
 - LRECL option [90](#)
 - OPTFILE option [89](#)
 - OUTPUT option [90](#)
 - PPCOND option [89](#)
 - RECFM option [90](#)
 - SECT option [84](#)
 - SEQUENCE option [89](#)
 - structure produced [90](#)
 - UNNAMED option [89](#)

E

- EDC5500 [112](#)
- EDC5501 [112](#)
- EDC5502 [112](#)
- EDC5503 [112](#)
- EDC5504 [112](#)
- EDC5505 [112](#)
- EDC5506 [112](#)
- EDC5507 [112](#)
- EDC5508 [112](#)
- EDC5509 [112](#)
- EDC5510 [113](#)
- EDC5511 [113](#)
- EDC5512 [113](#)
- EDC5513 [113](#)
- EDC5514 [113](#)
- EDC5515 [113](#)
- EDC5516 [113](#)
- EDC5517 [113](#)
- EDC5518 [113](#)

- emsg messages [64](#)
- ENUMSIZE compiler option [25](#)
- EQUATE DSECT utility option [86](#)
- error messages
 - compiler [105](#)
 - DSECT utility [111](#)
 - EXEC [107](#)
 - filter utility (CXXFILT) [111](#)
 - genxlt utility [111](#)
 - iconv utility [111](#)
 - localedef utility [111](#)
 - Object Library Utility [111](#)
 - redirecting [46](#)
 - runtime [109](#)
- EVENTS compiler option [26](#)
- Events file [45](#), [115](#)
- examples
 - machine-readable [5](#)
 - naming of [5](#)
 - softcopy [5](#)

EXEC

- C370LIB [73](#)
- CC [41](#)
- CDSECT [83](#)
- CMOD [56](#)
- CXXFILT [79](#)
- error messages [107](#)
- GENXLT [96](#)
- ICONV [95](#)
- LOCALDEF [97](#)

EXECs

- supplied by IBM [103](#)

executable

- files
 - placing CMS load modules in the BFS [70](#)
 - running CMS modules from the shell [71](#)
 - running, from the shell [71](#)
- modules, creating [55](#)

F

- feature test macro [37](#)

FILEDEF

- definition [xi](#)

filename

- definition [xi](#)

files

- executable [55](#), [71](#)
- names
 - absolute [49](#)
 - include files [48](#)

filter utility (CXXFILT)

- CLASSNAME option [80](#)
- error messages [111](#)
- REGULARNAME option [80](#)
- running under VM/CMS [80](#)
- SIDEBYSIDE option [80](#)
- SPECIALNAME option [80](#)
- SYMMAP option [79](#)
- unknown type of name [80](#)
- using [79](#)
- WIDTH option [80](#)

functions

functions (*continued*)
code set conversion [95](#)

G

GENMOD command [59](#)
GENXLT EXEC [96](#)
genxlt utility
error messages [111](#)
usage [95](#)
GLOBAL command [41](#)

H

HDRSKIP DSECT utility option [88](#)

I

IBM-supplied EXECs [103](#)
ICONV EXEC [95](#)
iconv utility
error messages [111](#)
usage [95](#)
include files
naming [48](#)
preprocessor directive [47](#)
record format [47](#)
system files and libraries
SEARCH compiler option [32](#)
searching for [52](#)
using [47](#)
user files and libraries
LSEARCH compiler option [27](#)
searching for [52](#)
using [47](#)
INDENT DSECT utility option [88](#)
INLRPT compiler option [26](#)
input
compiler [42](#)
source files [45](#)

L

library
archive
creating [101](#)
displaying the object files in [101](#)
file naming convention for c89/cxx use [101](#)
searching for objects by c89/cxx [101](#)
availability at run time [63](#)
Language Environment
compiler [41](#)
components [55](#)
runtime [41](#)
making available to the compiler [41](#)
search sequence
for include files [52](#)
with LSEARCH compiler option [27](#)
with SEARCH compiler option [32](#)
LINKLOAD EXEC
options [75](#)
LIST compiler option [27](#)

LKED command [61](#)
LOAD command [59](#)
load module, creating [55](#)
LOCALDEF EXEC [97](#)
LOCALE DSECT utility option [88](#)
localedef utility
error messages [97](#), [111](#)
LOWERCASE DSECT utility option [88](#)
LRECL DSECT utility option [90](#)
LSEARCH compiler option [27](#)

M

macros, feature test [37](#)
maintaining objects in an archive library [101](#)
maintaining programs with make using c89/cxx [67](#)
make utility
compiling and binding application programs [67](#)
creating makefiles [102](#)
maintaining C/C++ application programs [102](#)
makefiles
creating [102](#)
maintaining application programs [102](#)
mangled name filter utility [79](#)
map heading [76](#)
math considerations [56](#)
member heading [76](#)
message examples, notation used in [xiv](#)
messages
compiler, list of [105](#)
returned by XL C/C++ EXECs [107](#)
runtime [109](#)

N

naming, object library members [74](#)
natural reentrancy [62](#)
NOCSECT compiler option [23](#)
NODEBUG compiler option [24](#)
NOEVENTS compiler option [26](#)
NOINLRPT compiler option [26](#)
NOLIST compiler option [27](#)
NOLSEARCH compiler option [27](#)
NOOBJECT compiler option [29](#)
NOOPTFILE compiler option [30](#)
NOPPONLY compiler option [31](#)
NOSEARCH compiler option [32](#)
NOSOURCE compiler option [32](#)
notation used in message and response examples [xiv](#)
nucleus extension
compiler location [41](#)
program installation in [62](#)
NUCXLOAD command [62](#)

O

object
code [41](#)
compiler option [29](#)
library
adding object modules [73](#)
deleting object modules [73](#)

- object (*continued*)
 - library (*continued*)
 - example [74](#)
 - listing the contents [73](#)
- OBJECT compiler option [29](#)
- Object Library Utility
 - long name support [73](#)
 - map [75](#)
 - messages [111](#)
- OpenExtensions
 - binding using c89/cxx [69](#)
 - compiling and binding using c89/cxx [66](#)
 - compiling and binding using make [67](#)
 - maintaining objects in an archive library [101](#)
 - maintaining programs through makefiles [102](#)
 - placing CMS load modules in the BFS [70](#)
 - running [70](#)
 - specifying runtime options for [70](#)
- OPTFILE compiler option [30](#)
- OPTFILE DSECT utility option [89](#)
- output
 - compiler [45](#)
- OUTPUT DSECT utility option [90](#)

P

- passing arguments [39](#)
- PATHDEF
 - definition [xi](#)
- perror messages [109](#)
- POSIX
 - function call from non-POSIX function [4](#)
 - making use of [4](#)
- PPCOND DSECT utility option [89](#)
- PPONLY compiler option [31](#)
- Preprocessor output [45](#)
- primary input
 - specifying to the compiler [42](#)
- program module
 - definition [xi](#)
- programs
 - using makefiles to maintain [102](#)

R

- RECFM DSECT utility option [90](#)
- redirecting error messages [46](#)
- reentrancy [61](#)
- REGULARNAME filter utility option [80](#)
- response examples, notation used in [xiv](#)
- return codes
 - compiler [105](#)
 - DSECT utility [111](#)
 - genxlt utility [111](#)
 - iconv utility [111](#)
 - localdef utility [111](#)
 - Object Library Utility [111](#)
- run time
 - error messages [109](#)
 - return codes [109](#)
- running programs
 - from the shell [71](#)
 - OpenExtensions application [71](#)

- running programs (*continued*)
 - VM/CMS
 - example [62](#), [63](#)
 - with the START command [62](#)

S

- sample program
 - C source [7](#)
 - C++ source [11](#)
 - C++ template source [16](#)
- SEARCH compiler option [32](#)
- search sequence
 - include files [52](#)
 - library files [63](#)
- SECT DSECT utility option [84](#)
- SEQUENCE DSECT utility option [89](#)
- shared programs [61](#)
- shell
 - compiling and binding within
 - using the c89/cxx utility [65](#)
 - invoking load modules [71](#)
 - running programs [71](#)
- SIDEBYSIDE filter utility option [80](#)
- SOURCE compiler option [32](#)
- SPECIALNAME filter utility option [80](#)
- START command [62](#)
- stub routine
 - in Language Environment [55](#)
- symbol information [77](#)
- SYMMAP filter utility option [79](#)
- syntax diagrams, how to read [xii](#)

T

- template program example [16](#)
- trademarks [120](#)
- TXTLIB
 - command [73](#)
 - creating [73](#)

U

- UNNAMED DSECT utility option [89](#)
- user
 - include files
 - LSEARCH compiler option [27](#)
 - searching for [52](#)
 - specifying with #include directive [48](#)
- user comments [77](#)
- utilities
 - OpenExtensions [101](#)
 - XL C/C++ [73](#)

V

- VM/CMS
 - executable
 - module [59](#)
 - program [59](#)
 - GENMOD command [60](#)
 - LOAD command [60](#)

VM/CMS (*continued*)
 messages [64](#)
 running a program [62](#)
 VM/CMS
 compiling [41](#)

W

WIDTH filter utility option [80](#)
writable static [62](#)



Product Number: 5654-A22

Printed in USA

SC09-7625-73

